Specification and Verification of Reactive Systems with RSDS

Kalliopi Androutsopoulos

Submitted to the University of London
for the degree of
Doctor of Philosophy

June 2004

King's College of London
Department of Computing

Abstract

Formal methods have been applied to reactive systems in order to capture errors early on in the development life-cycle and reduce redesign costs. The Reactive Systems Development Support (RSDS) method provides support for the analysis and design of reactive systems and generates code from these specifications. An RSDS system is specified by a set of invariants, a set of statemachines and a Data Control Flow Diagram (DCFD), which are then verified using the B theorem-prover. B however requires user interaction and is not capable of proving temporal properties easily. This thesis extends RSDS by integrating model checking so that temporal properties can be verified. The model checker used is the Symbolic Model Verifier (SMV).

There are two distinct semantic views of statemachines in RSDS: the coarse-grain and the fine-grain, with the key difference between them being the granularity of a step. We describe a translation to SMV for each semantic view and we guarantee the quality of the translations by formally proving their correctness. This proof is a vital part in our provision of transparent formal method support for system design. To overcome the state explosion problem of model checking, we propose some natural ways of using the RSDS decomposition techniques for dividing the system into subsystems; these can then be model checked independently as separate SMV programs. We have tested our translations with various case studies.

RSDS/UML is an object-oriented version of RSDS that uses a restricted subset of UML for specification. It aims to bridge the gap between formal methods and mainstream software development techniques. For the same reasons as with RSDS, we integrate model checking with RSDS/UML by defining a translation for the coarse-grain and proving its correctness. The properties verified can reason over the dynamic instantiation of classes. The translation is illustrated on the gas burner system.

Acknowledgments

Firstly, I would like to express my sincere thanks to Kevin Lano, my supervisor, for his interest, patience and help, especially during the most difficult times. I feel lucky to have worked with such a dynamic researcher in the field. Also, I am very grateful to David Clark for his enthusiasm, stimulating discussions and for being a good friend. Thanks are also due to Tom Maibaum, who gave me the opportunity to study at Kings and find out for myself what research is all about.

My time at King's has been thoroughly enjoyable, especially because of the brilliant people that I have met and the close friends that I have made. In particular, I wish to thank Laurie Tratt, Naza Aguirre and Franco Raimondi for providing helpful feedback and lots of chats, tea (mate) and cakes (home-made). Also thanks to Jung, Paul, Ulle and Islam for sharing their thesis and viva experiences and for their comforting advice. My day-to-day life has been spiced up with the highly entertaining lunch time sessions with all the other friends in this department - you know who you are ! I am especially grateful to Arlene Ong (my twin) for cheering me on, spending lots of late nights working together, motivating each other, cooking for me and being there for a chat and a good laugh and of course for getting up to lots of mischief.

My friends from outside King's have been very supportive and I thank them for their patience and words of advice. Special thanks to Katerina, Jenny, Rachel, Sharon, Fuzz and Alexandra.

My most heartfelt thanks go to my parents for their unfailing support and love. I wish to thank my brother, Vassilis, for understanding most of the time what I was going through, for helping me out and for making me laugh.

Finally, I wish to thank Jameel Syed for being there with me every step of the way and for believing in me. Without his constant love and support, I would not have been able to reach my goals.

Declarations

This thesis is being submitted to the University of London in support of my application for admission into the degree of Doctor of Philosophy. The work carried out in this thesis is a result of original research (to the best of my knowledge) and it has not been submitted to another institution of learning for its examination. Some parts of this work are based on joint work produced in collaboration with other researchers and have been appropriately referenced. Specifically, the joint work is described in Chapter 3, and section 7.2.

I declare that any ideas or quotations from the work of other people, published or otherwise, have been acknowledged in accordance with the standard referencing practices of the discipline. I also recognise the valuable guidance and support of my supervisor, Dr Kevin Lano.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Background

With technological advances in computing, the complexity of software and hardware systems has increased as more sophisticated functionality and flexibility is demanded by users. Consequently, there is a high possibility of subtle errors that, if undetected, can lead to fatal accidents or great economic loss. In general, errors must be captured as early on in the development process as possible to reduce redesign costs and faults. In industry, there is a growing demand for software methods and tools that detect these errors in the early stages of development and that guarantee the correctness of the systems.

This thesis is concerned with the development of correct reactive systems and in particular control systems. *Reactive systems*, a term introduced by Harel and Pnueli in [HP85], are systems that continuously interact with their environment by responding to external stimuli. Their response consists of enforcing the desired behaviour on the environment, and is determined by the system's current state and the occurrence of an external event. Control systems are "reactive systems that control the operation of equipment or plant" [Sto96]. They are composed of three types of components: *sensors* that interact with the environment by receiving external events; *controllers* that compute the system's response based on the external events; and *actuators* that carry out the system's response by monitoring and controlling some parameters. Figure 1.1 illustrates the interaction of a control system with its environment.

Reactive systems are considered as safety-critical when a system failure has potentially devastating effects on the environment or to human life. In order to reduce system faults, standards have been defined, such as the defence standard [oD97], the railway industry standard CELENEC EN 50128 [501] and [Com99], a more general safety standard. These describe procedures and practices that must be complied with by industries to ensure that safety functions of systems are implemented correctly. They insist on the use of formal methods as a primary development method for the safety functions and properties, and recommend it for the rest of the system functions.

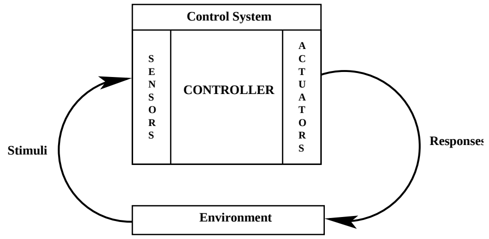


Figure 1.1: The interaction of a control system with its environment.

Formal methods use a mathematical framework for describing specifications of computer systems precisely and unambiguously, and ensure their correctness with verification. The benefit of formal methods is that they capture errors early on in the development process by verification, thus reducing faults and redesign costs. They are composed of two main parts: a formal notation and a verification system.

A formal notation has a formal syntax and semantics and is used to describe a system and its properties (invariants). Properties vary from system to system. For reactive systems, the properties are classified as [BBF$^+$01]:

Definition 1 (Safety Properties). Safety properties express that under certain conditions something never occurs.

Definition 2 (Liveness Properties). Liveness properties state that under certain conditions something will eventually occur.

Definition 3 (Reachability Properties). Reachability properties declare that some particular situation is reached.

Definition 4 (Fairness Properties). Fairness properties state that something will or will not infinitely often occur.

Definition 5 (Deadlock-freeness Properties). Deadlock-freeness properties express that the system cannot be in a situation where no progress is possible. It can be treated as a safety property.

Some of these properties can only be expressed using temporal logic and are known as temporal properties. Temporal logic is an extension of propositional logic that contains temporal operators that describe changes in time. It is usually used for reasoning about the dynamic behaviour of systems.

A verification system is a system that supports verification. Verification is the process of using a proof method to determine that the system satisfies its desired properties. The most commonly used methods for performing verification are theorem proving and model checking.

Theorem proving expresses the system and its properties as formulae in a particular mathematical logic. The logic is defined by a set of axioms and a set of inference rules that constitute a formal system. The process of verifying with theorem proving consists of showing that the system

satisfies its properties by constructing a proof in that logic for each of its properties. Axioms and rules, and possibly lemmas or derived definitions, are applied where necessary during a step in a proof. Software tools, collectively called theorem provers, have been developed to support this verification process. These tools can handle systems with infinite state spaces, making theorem proving a powerful verification technique. They often rely, however, on user interaction for determining the proofs and as a consequence the verification process is slow and often error-prone. Moreover, users are dissuaded from using this verification method as the notation is unfamiliar and the verification process requires some expertise.

Alternatively, *model checking* carries out the verification process automatically. The system is described as a model based on transition systems, with the desired properties expressed as logical formulae and correctness found by determining the truth of the formulae in the model. The algorithm that executes the verification process, exhaustively searches the state space to determine whether or not a property is satisfied. A counter-example is produced if a property is not satisfied by the model showing the steps that lead to the violation. The state space of a system model must be finite in order for the algorithm to terminate. Although the size of systems verifiable by model checking are limited compared to those handled by theorem provers, this approach is appealing to software developers as it is fully automatic.

A further limitation arises with the size of the state space due to the length of time required to model check a system. It is not feasible to model check large systems in a reasonable amount of time. Therefore, only systems with a “suitable” state space size are model checked. Since system sizes vary, it is difficult to determine what the “suitable” size is. There are two approaches that have been applied to solve this problem. The first attempts to optimise the model by reducing the size of the state space. Abstraction is one of the most popular techniques for this. The second approach focuses on optimising the model checking algorithm.

Neither verification approach presented is ideal for ensuring the correctness of all aspects of safety-critical systems [CW96]. A combination of both approaches is commonly used [JS93, Sha96, HTZ96] to counterbalance the disadvantages of each approach [RSG99]. Model checking can be first used to verify as many properties as possible automatically. If model checking fails to prove any of the systems properties due to its limitations, then a theorem prover can be used.

For many years, verification has been criticised as being the most difficult aspect of formal methods that deters potential users. The difficulty of verification has been alleviated by sophisticated theorem provers, such as Prototype Verification System (PVS) [COR$^+$95], that provide standard proof techniques that can be automatically applied and by improving the efficiency of model checking algorithms. In spite of these advances, users are still deterred from using formal methods for development, because of the difficulties involved in formulating specifications in the formal notation from requirements [BS00b], a process which often requires expert knowledge and experience. Formal methods usually provide unfamiliar modelling notations with no guidelines or templates to guide specification. As a consequence, users experience a steep learning curve as few software engineers have expert knowledge of both the application domain and of the formal notation. This leads to an increase in the length of projects as well as in the development costs. Thus, formal methods are not used in industry as widely as expected. However, many users are still interested in using formal methods for system development, because in some cases (e.g. for safety critical systems) the benefits surpass the limitations. Besides the benefits of provability, which provides a measure of system integrity, the use of a formal notation for the description of a specification aims to eliminate any ambiguity [Kli96] and provides a clear description of the systems goals and behaviour. In addition, the development of formal system forces the user to think more thoroughly about the design which helps identify faulty reasoning early on [BH95]. One way of improving the usability of formal methods is to integrate more widely used notations with existing verification systems.

An approach to developing system specifications that is widely used in industry is to use a collection of notations for expressing the requirements instead of just one. The different aspects of a system are described using different notations. A set of such notations used in a specification is termed a language. These notations can vary in formality. The most well-known language is the Unified Modelling Language (UML) [UMLa] that consists of twelve different notations.

The use of such languages can improve the applicability of formal methods. The languages used must be formal, in order to be able to prove correctness of a safety-critical system specification. Since UML does not have formal semantics, it cannot be used. The Requirements State Machine Language (RSML) [LHHR94, LHR99] and the Software Cost Reduction (SCR) [HBGL95] language are examples of formal languages that are composed of a collection of notations for specifying reactive systems. They were developed based on experience gained from developing actual systems. The Reactive Systems Development Support (RSDS) method also has such a formal language and it describes systems by using multiple notations i.e. statemachines, invariants and Data Control Flow diagrams (DCFD). It focuses on specifying control systems in a simple way that is readable by control engineers as well as by software engineers and to automate as much of the development process as possible. Verification of RSDS specifications is performed with the B method. The most recent version of RSDS (renamed as RSDS/UML) has a language that is considered as a subset of UML with precise semantics. In this thesis we are concerned with systems described using RSDS and RSDS/UML. We discuss these in more detail in section 1.2.

Formal languages are usually integrated with existing verification systems to ensure the correctness of system specifications. They are either integrated with theorem provers or with model checkers or with both. One approach that is commonly used to integrate the notations is translation. This involves mapping every element in the source notation to elements in the target notation. A specification is translated into the input notation of the verification system. Since a specification can be described using multiple notations (i.e. a language), it is often required that elements ranging across notations be translated as well. In some cases, not all of the elements in the source notation can be translated (or directly translated) into the target notation. Some way must be found to translate these in order to preserve the semantics of the source notation.

The translation approach is criticised [Day98] as having the following three disadvantages:

1. There is no guarantee that the translation preserves the semantics of the source language.
2. The results produced from some processing performed (in this case verification) on the target notation is not represented in the source notation.

3. The translation is notation-specific. Therefore, the specification is translated to a particular input notation of a verification system, and is limited to the verification capabilities of that system.

In this thesis, we use the translation approach to extend the verification support provided by RSDS. In this way, we are able to rigorously check that some desired properties hold in a system specification developed using RSDS, that we could not check with the existing verification support. We respond to the criticisms on the translation approach as follows:

1. We formally show that the translation preserves the semantics of the RSDS language. Thus, we are able to automate this translation and be confident that the model produced in the target notation correctly maps to the model of the RSDS specification.
2. We show how the results produced when applying the chosen verification system on the target notation are interpreted in an RSDS specification.
3. We want to adopt a notation-specific approach, i.e. the translation approach, because we want to make full use of the capabilities provided by a particular notation. In this thesis, the particular notation is a verification system that we want to integrate with RSDS in order to extend the verification support provided.

1.2 Problem Statement

In this thesis, we are concerned with extending the verification support provided for control systems that were developed using the RSDS approach. Since there are two main versions of the RSDS approach, one which is object-oriented and the other that is not, we present the research problem separately for each version.

1.2.1 RSDS

RSDS emerged from earlier work [LK98, LA99, LBA99] on the development of control systems that used a combination of techniques from control engineering and formal methods. The behaviour of a system controller was described by finite state machines (FSM), a notation familiar to control engineers. No timed or parameterised transitions were required. The system structure was visualised with Data Control Flow Diagrams (DCFDs). DCFDs illustrate the flow of events between the system components. From these visual descriptions, specifications in the notation of the B formal method [Abr96, LH96, Sch01] were produced for the purpose of verification and for the automatic generation of code.

B is a formal method for developing systems from formal specification to code. It expresses specifications using the B Abstract Machine Notation (AMN) that is based on first order predicate logic and set theory. It is a modular notation, where the basic modules are known as machines. Each machine consists of data, operations for manipulating the data and invariants that the data must always maintain. Once a system is specified using a set of machines, it must be formally checked by proofs to ensure that the invariants are maintained. Stepwise refinement techniques are applied on the machines for deriving code, such as C or C++. For each refinement, it is required to formally show (by proofs) that it satisfies its previous specification. The intermediate modules produced are known as refinements and implementations. There are two tools available to support development with the B method: the B-Toolkit [BTo] and Atelier B [Ate]. However, the proofs required to show that the invariants are maintained and those for refinement usually require user interaction.

Thus, the primary concerns of the earlier work were how to best describe FSMs of reactive systems as B specifications so that a translation between the two notations can be defined that can be automated. Consequently, B specifications were automatically generated from FSMs. This aimed to bridge the communication gap between control engineers and software engineers as control engineers need not have any knowledge about formal methods. Moreover, errors that may be introduced by users when formulating specifications are reduced.

The RSDS method [LAK00, LAC00, LCAK00, LCA01, AL01] has similar objectives. However, as a result of the earlier work, it has the following additional objectives: to provide a modelling language that is as simple as possible for describing reactive systems; to provide tool support for automating as much of the development process as possible and to ensure that all the system's properties expressed in a specification are verified.

The RSDS language is comprised of the following notations as illustrated in Figure 1.2: invariants, a modular statemachine notation (variation of FSM) and a DCFD. Invariants are logical formulae that must always be true in a specification. They are used to precisely describe the system requirements in a simple and abstract way. Statemachines describe the dynamic behaviour of the system and have two semantic views concerning the granularity of a computation step: the coarse-grain and the fine-grain. A single step in the coarse-grain corresponds to several steps in the fine-grain. In the fine-grain view, the specific order in which transitions of actuator components occur is described, while in the coarse-grain they occur simultaneously. DCFDs are used in the design stage for visualising the system structure. Various decomposition techniques can be applied to decompose the system into smaller components. A control algorithm is automatically synthesised from the invariants and used to generate specification in B and code such as Java.

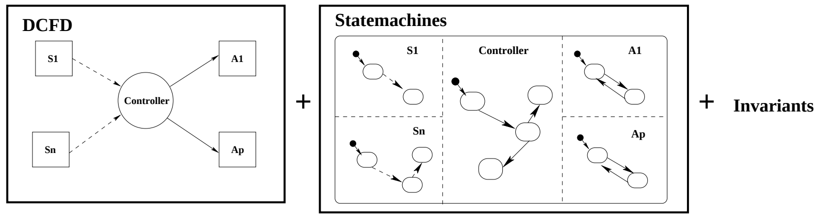


Figure 1.2: The constituents of an RSDS specification.

RSDS uses B to ensure, by proof, that the invariants are maintained by the specification. Invariants at both semantic views of statemachines are verified: the coarse-grain corresponds to execution semantics of B machines, while the fine-grain corresponds to B implementations where

operations are executed sequentially. However, the verification support provided by the B method is not sufficient.

The proofs in B are not completely automated. Most non-trivial proofs will require human interaction. These are challenging, requiring expertise, and can be erroneous if incorrect lemmas are provided by the user.

The structure mechanisms of B impose some restrictions that can make it difficult to specify and structure specifications of reactive systems in B. For example, the INCLUDES structuring mechanism allows no cycles, that is, no two machines are allowed to change the state of a shared machine at the same time (single writer and multiple readers only). This restriction arises to enforce non-interference with compositionality and to preserve independent refinement of machines. In [BPR96, BB99, DBMM00, Lec02, Rod] ways of overcoming these limitations are presented but unfortunately are not yet implemented in the B tools. INCLUDES is used to structure B machines that represent system components. Its impossible to specify a reactive system with two controllers that share an actuator. In practise, there is a way to overcome this (described in detail in section 3.4) by defining dummy specifications and then replacing the code generated accordingly. However, this is not a good solution as we change the code generated after we have proven the correctness of the implementation. Therefore, we believe that one cannot verify a system structured in this way easily with B.

Temporal properties cannot be proven easily in B. In [AM98], new clauses are introduced in the B notation for expressing temporal properties of event-driven systems. However, the manner in which these properties are expressed is complex and does not resemble their logical form. For example, the temporal property $\Box(P \Rightarrow \Diamond Q)$ (which defines "leads to" $\leadsto$) for predicates P and Q is expressed in B with a loop and the proof required must show that it terminates. It is defined in B in five parts under a MODALITIES clause as follows.

```
MODALITIES
  SELECT
    P
  LEADSTO
    Q
  WHILE
    F_1 OR ··· OR F_n
  INVARIANT
    J
  VARIANT
    V
  END
```

P is the predicate that must be true initially and Q is the predicate that must be eventually true. A non-empty list of event names F_1 OR $\cdots$ OR F_n are required to identify those that may be taken in the loop in order for P to progress to Q (if none is given then all the events in the system are considered). J is an optional invariant predicate that must be true during the loop (i.e. the invariant must be true between the activation of all events). Finally, V is a natural number expression that represents the decreasing variant of the loop. The new clauses used in this example are LEADSTO, INVARIANT and VARIANT.

Expressing temporal properties in this way could lead to the additional introduction of errors as further information is required from the user, that is, a variant, a loop invariant and a list of events. Besides the modality "leads to", there are two other patterns used for describing temporal properties: the modality "until" ($\Box(P \Rightarrow (P \mathcal{U} Q))$ and the dynamic invariant. These patterns correspond to only a fragment of Propositional Linear Time Logic (PLTL), meaning that the range of properties that can be expressed in this way is limited compared to the expressive power of LTL or Computational Tree Logic (CTL), for example. Moreover, these new constructs have not been implemented by the B tools and therefore cannot be used in practice.

An alternative solution is provided in [BCJ02, BDJK00, JMM99] that uses a combination of model checking (SPIN) and B for verifying temporal properties. The temporal properties are expressed in Propositional Linear Time Logic (PLTL) and model checked on a finite-state transition system of the abstract B specification. The static properties are proved in B as normal. As the B specification is refined, new PLTL properties are described for the new events that are introduced and the old PLTL properties are redefined. The temporal properties of the refinements are not verified by model checking but are proven using B. This approach seems to be more suitable for verifying the temporal properties of RSDS specifications. However, since RSDS specifications are described using statemachines that are similar to transition systems on which model checking is performed, it seems unnecessary to translate RSDS specifications into B and then into transition systems for model checking. A much simpler solution would be to translate RSDS specifications directly into the input language of a model checker, that also allows for verification, independent from B. The standards [Com99] and related railway industry standards such as CELENEC EN 50128 [501] recommend that several verification tools should be used to provide independent verification. Moreover, RSDS does not use the refinement techniques of B during system development.

Recently, the B method has been extended with a model checker, ProB [LB03], but it only verifies standard B invariants and not temporal properties as yet.

Therefore, we propose integrating a model checker with RSDS for solving the verification problem of temporal properties. To integrate the model checker with RSDS, we define translation rules for deriving a model of the system in the language of the model checker. Since the performance of model checking is handicapped by the state space explosion problem, we investigate how the structure of RSDS specifications can be exploited in a natural way to reduce the size of the model generated.

1.2.2 RSDS/UML

RSDS/UML [LCA04, LAC03, LCA02c, LCA02d, LCA02a] came about in response to finding a way of improving the applicability of formal methods and bridging the gap between mainstream software development techniques and formal methods. Object-oriented design is mainly used in mainstream software development. By integrating object-oriented techniques and formal methods, the former obtains the formality and verification support that object-oriented notations often lack, while the latter obtains familiar modelling techniques for formulating modular specifications that are flexible, maintainable and reusable. Examples of such integrations include Object-Z [Smi00], VDM++ [Gro00] and U2B [Sno02].

RSDS is extended with a subset of UML to form RSDS/UML that specifies systems in an object-oriented way. UML was chosen over other object-oriented languages because of its close resemblance with the RSDS language and because it is an international standard. RSDS/UML specifications are described using a restricted subset of the following UML notations: a class diagram, object diagrams and a statemachine notation. A class diagram is used in the place of the DCFD to model the relationship of system components with classes and associations. The instances of classes and associations are visualised using object diagrams. Invariants are defined in the class diagram as OCL constraints and are attached to classes (local properties) or to associations (properties describing the interaction of numerous classes). The dynamic behaviour of instances is described using statemachines. These statemachines also have a coarse-grain and fine-grain semantic views. Figure 1.3 illustrates the different UML notations used to specify reactive systems with RSDS/UML.

RSDS/UML is still under development. In this thesis, we use one of the more stable versions [LCA02a, LCA02c] whose semantics are given. The future version of RSDS aims to reduce the number of a classes used to specify a system, by defining a class for each subsystem with attributes for the sensor and actuator components.

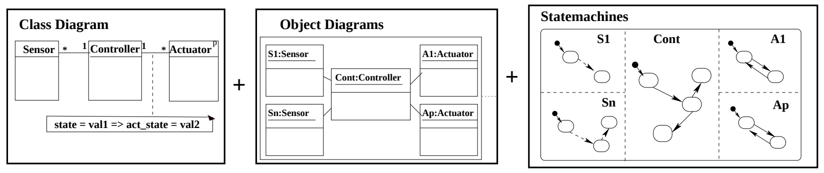


Figure 1.3: The constituents of an RSDS/UML specification.

There are many benefits in integrating UML with RSDS in this way. Firstly, RSDS/UML is suitable for the development of critical systems as it has precise semantics, whereas UML has some semi-formal semantics [UMLa], but in conjunction with the informal semantics are incomplete. Compared to RSDS, RSDS/UML can represent several similar components as a single class that can create and delete objects as needed. This is particularly useful for control systems as these systems usually contain a large number of similar components, such as switches or valves, and template classes for these components can be defined and reused. Furthermore, in the RSDS/UML language it is possible to define temporal properties that specify how the relationships dynamically change (or don't change) between objects of classes representing components.

As with RSDS, we need to provide verification support for RSDS/UML. There are two types of properties that must be verified: properties local to a class that only refer to local attributes and properties that describe behaviour with associated classes that can refer to attributes from all classes involved in the association. In addition, we need to ensure that with the creation and deletion of instances in an RSDS/UML specification, the relationships among them remain correct. Some of these properties are temporal and therefore the verification support provided must also support the verification of temporal properties.

Again, the B method is used to provide verification support and translations have been published in [LAC03, LCA04]. However, we still encounter the problems in B that we discussed in section 1.2.1 with verifying temporal properties. Therefore, we choose to apply model checking to RSDS/UML specifications as well, for verifying the temporal properties. Hence, we integrate model checking with RSDS/UML by defining a translation and proving its correctness. In this thesis, we only consider the coarse-grain semantic view of the RSDS/UML statemachines as we want to investigate to what extent model checking is suitable for verifying these object-oriented systems. If the results of this investigation are positive, model checking can be applied to a larger set of UML (newer versions of RSDS/UML) and a wider range of applications could be specified and verified in this way.

1.3 Contributions

The main focus of this thesis is to find a way to verify the temporal properties of reactive systems developed with RSDS and also with RSDS/UML. Several verification systems (proof and model based) exist that support the verification of temporal properties. To verify the temporal properties of RSDS and RSDS/UML specifications, we choose to integrate an existing verification system with RSDS and RSDS/UML instead of defining a new one. The main criterion for choosing which verification system to integrate is its degree of automation. We prefer a verification system that carries out the verification process automatically and thus adheres to the objectives of RSDS and RSDS/UML. Model checking is such an approach that can be integrated. In particular, we have chosen to use the Symbolic Model Verifier (SMV) [McM93] which has its own tool support. We justify this choice by relating SMV with other tools in section 2.7.

We use the translation approach to integrate model checking with RSDS and RSDS/UML. The translations that we define map elements in the source language (RSDS or RSDS/UML) to elements in the model checking language (SMV). These mappings can later be automated to prevent any user interaction that could introduce errors. Since we are using the translation approach, it is important that we are able to overcome the problems associated with the translation approach discussed in [Day98]. Therefore, we formally show that the translations preserve the semantics of RSDS and RSDS/UML respectively. These proofs of correctness are very important for safety critical systems as we can be sure that the model being model checked does in fact represent the system specified. We emphasise that the proof of correctness is what makes our approach unique as many similar translations [Kwo00, CAB$^+$98, LHHR94, HN96] have not provided such a proof. In the chapters were our translations are defined, we compare and contrast our translation with other similar translations. Also, we show how the results from the model checker are interpreted on the original specification. In the following sections, we discuss the specific contributions made for RSDS and RSDS/UML respectively.

1.3.1 RSDS

Our main contribution is that we have provided a way of verifying the temporal properties of RSDS specifications by translating into the notation of a model checker. A translation is defined by a set of translation rules that can be used to automate the translation. Since RSDS specifications have

two different semantic views of statemachines (the coarse-grain and the fine-grain), we define a translation for each view. Only the coarse-grain translation has been implemented by Kevin Lano in the RSDS tool based on the translation rules defined in this thesis.

We prove the correctness of both the coarse-grain and fine-grain translations with respect to the RSDS semantics, to ensure that we do not verify a misrepresented system. This proof requires that the RSDS language has formal semantics. Therefore, we have consolidated a number of versions of semantics of RSDS. Moreover, the proof of correctness requires the semantics of the model checking language, in this case SMV, to be formally defined and preferably in the same way as the RSDS semantics (axiomatically). Thus, we define the semantics axiomatically for the subset of SMV that we use in the translations. Figure 1.4 outlines the translations that are required for model checking RSDS specifications and the respective proofs of correctness.

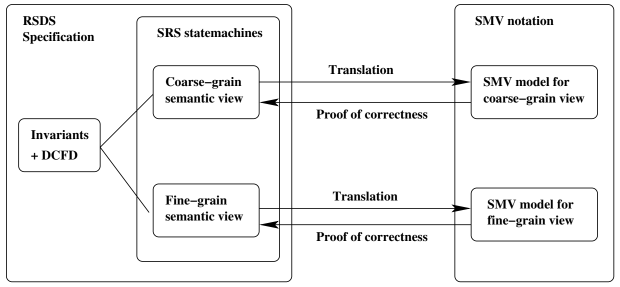


Figure 1.4: An outline of the translations and proof of correctness.

The size of the SMV model produced for large systems can be so large, because of the state space explosion problem, that it is impossible to verify. In RSDS, large systems are structured by applying decomposition techniques. Therefore, we discuss natural ways in which the structure of the systems can be exploited to reduce the size of the SMV model.

Finally, the translations have been tested on several systems specified in RSDS and in RSDS/UML such as the gas burner system, the production cell and the autopilot. In particular, we use the autopilot system to compare the entire RSDS method (with the integration of the SMV) against two popular approaches: the SCR (a method) and PVS (a verification system).

1.3.2 RSDS/UML

Our contribution for RSDS/UML specifications is that we have found a way of verifying their temporal properties, again by translation. In this case, we have only defined translation rules for the coarse-grain view of RSDS/UML because RSDS/UML is still under development and we want to first investigate to what extent model checking is suitable for verifying object-oriented systems. If the results are positive then we can pursue to define both translations with a stable version of the RSDS/UML. We also formally show that the coarse-grain translation preserves the RSDS/UML semantics. Figure 1.5 indicates what we have achieved for RSDS/UML specifications, that is, we have defined a translation for the coarse-grain view and proved its correctness. The gas burner system and part of a railway system have been implemented in RSDS/UML and the translations have been applied to generate SMV code.

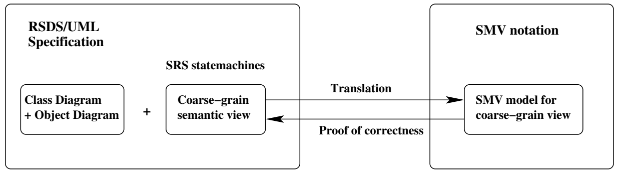

RSDS/UML Specification
SRS statemachines
Class Diagram + Object Diagram
+
Coarse–grain semantic view
Translation
Proof of correctness
SMV notation
SMV model for coarse–grain view

Figure 1.5: The translation and proof of correctness that we define for RSDS/UML.

1.3.3 Contribution relative to co-authors

The development of RSDS and RSDS/UML is based on work produced in collaboration with the researchers: Kevin Lano and David Clark. My role in this work is clearly defined. I have been primarily concerned with the verification of temporal properties of RSDS specifications by model checking. I have used the translation approach to integrate RSDS and the SMV model checker. However, for the translation I defined and for its proof of correctness, I had to have a consistent definition of the RSDS language. Kevin Lano and David Clark are working on the (formal) definition of the RSDS and RSDS/UML language. Therefore, the joint work effort mainly involved the definition of RSDS and RSDS/UML. Kevin Lano has also been working on defining translations into B and has been implementing the RSDS tool[1] and recently the RSDS/UML tool. The current implementation of the RSDS tool allows a user to define a coarse-grain RSDS specification, automatically check its consistency, generate SMV code (for the coarse-grain) based on the rules defined in this thesis, and also generate B code.

In this thesis, joint work is discussed in Chapter 3 where the RSDS method is introduced and its language is formally presented. Since the RSDS language is always evolving, I have consolidated the existing semantics and have extended it to formally define controllers as amalgamations of sensor modules. Also, joint work is discussed in sections 7.1 and 7.2, which define the language and semantics of RSDS/UML.

1.4 Field of Modelling and Verification of Reactive Systems

In this section, we discuss methods and languages comparable to RSDS (and RSDS/UML) for specifying and verifying reactive systems. These languages are based on one the following three formalisms that are used for specifying reactive systems: *transition systems*, *Petri nets* and *process algebras*. We informally introduce the three formalisms and since RSDS is based on transition systems (in particular finite-state machines), we review in more detail languages based on this formalism. Each language needs some procedure for checking the validity of a specification. The level of difficulty of performing this checking depends on the language that is used and the level

[1] I have been involved in the earlier developments of the RSDS tool.

of validity that is required (for example, safety critical systems require a high level of validity). We also discuss the verification support provided for each language.

Languages can be synchronous or asynchronous, depending on how concurrency is viewed. Synchronous languages implicitly assume the presence of a global clock. At each clock tick, all inputs are considered, the outputs and new states are calculated and then transitions are made. It thus has the advantage of guaranteeing deterministic behaviour. On the other hand, asynchronous languages describe several processes that are non-deterministic since the order in which tasks are executed is not specified. This is a more realistic view of concurrency, however, synchronous languages provide high level modular constructs that aid with the design of these systems. RSDS and RSDS/UML are synchronous languages and therefore we only discuss synchronous languages in the related work. Also, we only consider languages that specify discrete reactive systems with no timing (i.e. timing constraints, delays).

1.4.1 Transition Systems

The most widely used formalism for specifying reactive systems is transition systems, which have different forms depending on their use. An example of a transition system is illustrated in Figure 1.6. Basically, a transition system is a directed graph where the nodes represent the *states* of a system and the edges represent *transitions*. There is a designated state that represents the initial state of a system. Usually, transition systems that have labels on states are known as Moore machines, and those that have labels on transitions are known as Mealy machines. Transition systems can have labels on both states and transitions. Consequently, transition systems can be state-based or event-based. Some systems are modelled more naturally using one rather than the other. The reactive behaviour of a transition system is given by a sequence of transitions or states starting from the initial state. For example, in Figure 1.6 one such behaviour is: $a1, a2, b1, c1$.

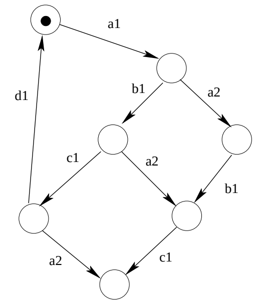


Figure 1.6: An example of a transition system.

A more expressive form of transition systems was informally introduced by Harel in [HN96] known as statecharts, that introduces hierarchy and modularity with the definition of AND and OR states. If a state with inner states is active and its inner states are OR states, then exactly one of the states will be active. If its inner states are AND states, then all of its states will be active. Many successful attempts have been made to formalise the statechart semantics [PS91, HN96, LMM99b], which resulted in some many variants of the language. One such variant (but not entirely formal) is part of the UML [UMLa].

There are two ways in which transition systems are verified: model checking which is the most natural choice as their models are based on transition systems (Kripke models), and formal verification systems such as PVS, or the B prover. For the latter, the transition systems used to describe reactive systems must be represented in the specification language of the formal method. In some cases, this is not straight forward as the formal methods were not developed with these particular systems in mind. For example, event-driven B is introduced in [Lec02] for defining reactive systems as B specifications, and in [Led91], the Vienna Development Method (VDM) [DN87, BR91] is extended in order to specify and verify reactive systems.

The languages based on transition systems can have a tabular form, be expressed as a state diagram, or textual object code format. The language of RSDS and RSDS/UML includes statemachines that are expressed as diagrams.

Tabular form

Transition systems can be expressed using a tabular notation that lists the current state and inputs, transitions and next state or outputs. The problem with this approach is that the number of rows in the table grows exponentially with the number of inputs. However, consistency of the table can be simply determined as there must be exactly one row for each state/input and each output or next state must be an allowed output or state.

The SCR method specifies reactive systems using a tabular notation based on transition systems. It consists of two formal models [Hei02]: the Four Variable Model and the SCR requirements model. In [HBGL95] a set of software tools were developed to analyse documents with SCR requirements. The SCR requirements model [HJL96] is a special case of the Four Variable Model that provides precise semantics by representing the system as a statemachine. Three types of tables are used for describing specifications with SCR: mode transition tables, event tables and condition tables. The mode transition table describes the system modes in terms of the system states and how they are changed when transitions occur. Event tables are used to describe the changes to the variables depending on the events that occur and also which mode the system is in. Condition tables describe how the values of the output variables or terms change as a function of a mode and condition. Besides the size of the tables that can grow exponentially, it is difficult for one to follow a particular reactive behaviour as it can range over a number of tables. A variable dependency graph is introduced to help reduce this problem. SCR provides both model checking (translations into SPIN and SMV) and the use of a theorem prover (PVS) for verification. Their tool hides the verification process from the user as it provides a friendly user interface to PVS and interprets the results of the model checkers. In Chapter 6 we compare SCR with RSDS in detail.

State Diagrams

State diagrams refer to the collective graphical representation of transition systems. The particular form of each element can change from one diagram to the next, depending on which transition

system is illustrated. These are slightly better than the tabular form because transitions are visualised easily and it is easier to understand the reactive behaviour of the system. However, verification is more difficult than in the tabular form, because of the use of predicates on the labels of the transitions instead of the explicit representation of the tabular form.

STATEMATE [Sta] is a tool for specifying and analysing systems. Its modelling language is Harel's statecharts [HN96]. Compared to RSDS statemachines, statecharts are more expressive as they allow nested OR and AND states. STATEMATE, by performing checks, ensures that good design practices are adhered to when describing specifications. For example, it checks that every part of the model should be reachable, i.e. that there is no "dead code". Model checking is used to perform these checks. Model checking is also used for testing the behaviour of the model in extreme or critical configurations, as well as for verifying desired properties of a system. However, model checkers suffer from the state space explosion problem and for large systems they are unfeasible. The RSDS method aims to use model checking as well as the B prover for verification.

Requirements State Machine Language (RSML) [LHHR94] is a requirements specification language that uses a modified notation of statemachines ([HN96], [PS91]) as well as other techniques like AND/OR tables, to specify each component. As with STATEMATE, RSML statecharts are more expressive than RSDS statemachines and the notion of a step is different. Moreover, RSDS presents two semantic views. RSML is limited with respect to the structuring mechanisms that it provides: like SCR, it only decomposes the controller by modes. A variety of analyses can be performed such as: completeness and consistency analysis, fault tree generation, model checking, code generation and test data coverage. These range over the entire development life cycle and together form a complete method. RSDS also aims to be a complete method, however, it does not provide any support for performing hazard analysis.

DOVE (Design Oriented Verification and Evaluation) [Dov] is a tool with a graphical editor for drawing statemachines, an animator for simulating the execution and a prover for verifying the critical properties. Unlike RSDS, the tool does not provide structuring mechanisms and guidelines and does not check the liveness property because it represents only finite configurations. The only property it can check is the progress property i.e. a state can be reached after a number of transitions.

Roscoe shows in [Ros03] how to verify statecharts using the model checker FDR. The novelty in his approach is that statecharts are translated first into CSP (see section 1.4.3) and CSP methods such as compression, abstraction, data independence and symmetry are used to improve the performance of FDR checks. Therefore, the state space explosion problem inherent in all model checkers is reduced in this case via CSP.

Object Code (OC)

The Object Code format was developed as an intermediate language for the synchronous languages: Lustre [CPHP87, HR99, Hal93], Esterel [Ber98] and Argos [Mar91, MR01]. It is the easiest to execute on a sequential processor. Unlike other high-level languages, it does not have any concurrency as it only describes a single finite-state machine. It is only suited to describing sequential control processes. Therefore, an OC program describes a single finite state machine.

Each state has a decision tree attached to it. The nodes in the decision tree are indices of an action table and the leaves of the tree are pointers to the next states. An action table contains a list of atomic behaviours which include testing a variable or signal, computing the new value of a variable, calling an external function or sending out a signal. The main drawback of this approach is that all OC languages suffer from is that OC forces two events to happen in a strict order, when they could happen simultaneously. This causes some artificial dependencies to form that can lead to system deadlock.

Lustre is a declarative, textual synchronous language that mainly concerns itself with dataflows. Lustre programs consist of expressions that define flows. Flows describe a possibly infinite sequence of values of a particular type along with a clock (sequence of times for a sequence of values). A restriction imposed by the compiler is that the operators work pointwise on flows, for example, if x and y are flows $(x_1, x_2, ..., x_n)$ and $(y_1, y_2, ..., y_n)$ with identical clocks, then $x + y = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n)$. Sequential behaviour can be defined using delay and initialisation operators. It is fairly easy to check consistency of Lustre programs. This is because feedback loops without a *pre* operator (it delays the flow by one clock cycle) are prohibited.

Esterel is a textual and imperative synchronous programming language and also a compiler which translates Esterel programs into finite-state machines. It is well-suited for specifying sequential control-dominated tasks. It is deterministic and concurrent, and supports preemption as well as an exception mechanism that is compatible with concurrency. An Esterel program consists of a group of modules that are executed concurrently. These modules communicate through signals that at each clock tick are either absent or present with a value. Sequential behaviour can be described succinctly with Esterel. Consistency checking of Esteral programs is more challenging than for the other OC languages. This is because it is easy to write paradoxes in the language and in order to look for them, every possible execution of the program must be explored. Recent compilers can do this symbolically.

Argos is a language that describes specifications as hierarchical finite-state machines, a synchronous derivative of Harel's statecharts. Therefore, it allows the definition of inner states as OR and AND states. Consistency checking of an Argos specification is more difficult than that of a state diagram. For example, a possible check might consider which states are reachable from a particular sequence of inputs.

Argos has been extended into the SyncCharts [And02, And96] formalism. SyncCharts support hierarchy (OR states), concurrency (AND states), communication (instantaneous broadcasting of signals) and preemption (abortion and suspension). A SyncChart model can be automatically translated into an Esterel program in order to use the software environment of Esterel, such as checking for consistency and generating code.

1.4.2 Petri Nets

Petri nets is a class of models of concurrent systems that were introduced as an extension of standard finite automata models with communication. There is a vast amount of literature on Petri nets and its variations: in [PR91] more than four thousand entries have been given. A Petri

net model is a directed bipartite[2] graph as illustrated in Figure 1.7. The circles in the graph represent *conditions*, while the boxes represent *events*. If a circle contains a black dot, then a condition is marked with a *token*. The arcs in the model are of two kinds: input arcs that connect conditions to events, and output arcs that connect events to conditions. The dynamics of the model are given by the *token game* rules that define how the tokens float between conditions. The token game rules state: an event fires by taking a token from each of its input conditions and putting a token in each of output conditions. The intuition behind the meaning of events, conditions and tokens for modelling systems is that events represent the transitions of a system, conditions represent local resources and tokens represent the availability of local resources. For example, the Petri net given in Figure 1.7 that is taken from [Pet] models a queuing system .

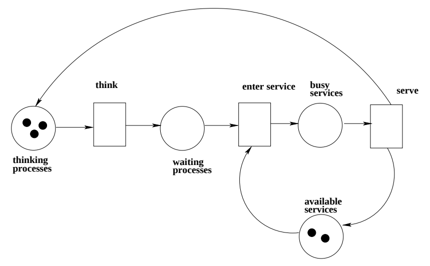


Figure 1.7: A Petri net that model a queuing system.

A way in which Petri nets can be analysed is to compute all the tokens reachable from the initial token. In this way, one can explain how the tokens move through the conditions around the model. Transition systems can be derived from these reachable tokens by labelling the arrows with e from a token T to token T' where e is the event that when fired in T moves the token to T'. This procedure is known as *state space enumeration*. Since Petri nets can be described as transition systems, the same notation of behaviour for transition systems apply to Petri nets.

1.4.3 Process Algebras

Process algebras (or process calculi) are a family of languages that are used for describing concurrent reactive systems. The most eminent of these are: CCS (Calculus of Communicating Systems) [Mil80, Mil89], CSP (Communicating Sequential Systems) [Hoa85] and ACP (Algebra of Communicating Processes) [BK84]. A process algebra is a language that describes processes that can be built from subprocesses by using a number of combinators and also a facility for spawning (generating) subprocesses. A process refers to the behaviour of system. Some examples of processes are:

inaction 0 is the simplest process, i.e. the process that cannot perform any actions

prefixing $a.P$ denotes a process that can perform an action a and become P

[2]A bipartite graph is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.

sequential composition $P;Q$ denotes a process that behaves as P and then Q

non-deterministic choice $P+Q$ denotes a process that can become either P or Q

parallel composition $P||Q$ denotes a process whose subprocess P and Q are running concurrently

Milner was the first to realise that such combinators exhibit or should exhibit algebraic properties. The semantics of process algebras are usually given using structural operational semantics (SOS) [Bae04]. The structure of the syntax guides the derivation of the operational or stepwise behaviour of processes by means of SOS rules.

Verification of process algebras is most successful when verification techniques that come from process algebras are used, i.e. equational reasoning [Bae04]. Model checking and theorem proving have also been applied. The use of a combination of these verification techniques is favourable.

[GR]

Jack (Just Another Concurrency Kit) [GLM99] is a support environment for the specification and verification of reactive system by means of process algebras. It provides a graphical view of the specification and translates properties described in natural language into ACTL formulae. In this way, it assist in the formalisation of informal system requirement which can eventually be model checked.

1.5 Model checking Statecharts

The RSDS statemachine language is a restricted variation of statecharts, i.e. it is more expressive than finite-state machines as it contains AND states and OR states (these cannot be nested). This thesis is about model checking RSDS statemachines. Therefore, we relate our work with other approaches that model check statechart models, and in particular with the SMV model checker. There are two key translations that closely relate to our work: translation of RSML statecharts [CAB$^+$98] to SMV and the translation of STATEMATE to SMV [CH00]. We compare and contrast these with our translations at the end of Chapter 3 and 4, where our translations are defined. In this section, we discuss briefly the problems encountered with verification of statecharts as presented in [BR04] and we give an indication of the future directions of this area of research.

1.5.1 Problems encountered with model checking statecharts

Statecharts are very expressive and contain features that prove difficult to translate into SMV because it is based on transition systems that are more primitive. RSDS statemachine have only a few features of statecharts and therefore does not face that many challenges. These features are:

- **State Hierarchy:** usually the hierarchy of statecharts is flattened and then translated into SMV. However, this could lead to a SMV model with an enormous state space which if it is greater than 10^{20} state cannot be model checked. Also, the SMV model generated will be very complicated to read. RSDS statemachine also have a hierarchy. However, its not very deep, as AND states cannot be nested with OR states and so on. Therefore, the modular structure of RSDS specification is preserved in the translations.

- **Inter-level transitions:** these are rich control transfers between state hierarchies. RSDS statemachines do not model inter-level transitions.
- **Conflicting transitions:** transition are conflicting if they have the same source state. RSDS statemachines do not allow the occurrence of these transitions.
- **Concurrency:** the view of concurrency that RSDS adopts is that of synchronous execution which can be easily represented in SMV. RSDS has two semantic views: the coarse-grain and the fine-grain. The first denotes that the reaction to a sensor event must occur in a single step (all actuator events happen concurrently), while the second denotes that the specific order of actuator events must be defined. The second proves to be challenging when modelling in SMV as it is difficult to specify this order.
- **History: statechart model have history connectors for recalling states that have been visited. RSDS statemachines do no have such connectors.**

1.5.2 Future directions

According to [BR04], the future work for verifying statecharts, that is based on identifying the gaps in the current approaches, is the following.

1. **Preservation of statechart hierarchy in model checking** Alur in [AGM00, AY01, AKY99] presented a way in which the hierarchy of hierarchical reactive modules was preserved when translating. Although the semantics of reactive modules is different from statecharts, in [BR04], they suggest that it would be very useful if the approach in it is adopted for statecharts.
2. **Compositional verification** Compositional verification consists of verifying properties of sub-components that guarantee properties of the entire system. A compositional verification technique for verifying Argos programs has been presented in [Ram00], that could be extended and used for statecharts. The sub-component can be either concurrent automaton or top-level automaton with some hierarchical states being reduced to basic states. By verifying the sub-component only, the state space of the model being verified is reduced. A limitation of this approach is that not all properties can be proven in this way, at least not easily. The result from verification holds, if the input formula satisfies certain conditions such as that it is local (referring to localised events) and that it does not state anything negative about states or events not in the sub-component.
3. **Refinement** Statecharts can be used to define high level abstract specification that can be refined to into a detailed specification, that is also modelled by statecharts. Therefore, verification would consists of checking the detailed specification implements the abstract specification by defining refinement maps between the two specifications.

Another point that was made in [BR04] is that the tools that perform the translations are not at a commercial standard and they do not manage to translate all of the features of statecharts. Moreover, the results produced by the model checker are not interpreted on the statecharts, hence there is no traceability between the models. Traceability would improve the use of model checkers as debugging tools.

1.6 Thesis Outline

Chapter 2 introduces the field of model checking. We briefly discuss the different approaches for implementing model checking algorithms and several techniques that were developed for reducing the state space explosion problem. For model checking RSDS specifications, we choose to use the SMV model checker that is developed according to the temporal logic (branching time) approach. Therefore, we present the temporal logic model checking approach in more detail. We give an overview of some model checking tools available and we provide justification for our choice.

Chapter 3 presents the RSDS method and how it can be used to develop reactive systems. In addition, the semantic foundations of RSDS are formally defined, including the definitions of the two semantic views of statemachines: the coarse-grain and fine-grain. In order to demonstrate how the RSDS method can be used in practice, the gas burner system is developed using RSDS. This system is referred to throughout the thesis.

Chapter 4 describes how coarse-grain specifications are model checked by translating them into the input language of SMV. The translation is given as translation rules and is illustrated with the gas burner system. In order to prove the correctness of the translation with respect to the RSDS semantics, we define the SMV semantics formally using the same logic as that used to define the semantics of RSDS. Moreover, we show some natural ways in which the modular structure of decomposed systems can be used to reduce the state space of the SMV model produced.

Chapter 5 describes how fine-grain RSDS specifications are model checked, also by translating them into the input language of the SMV model checker. This chapter complements the work in Chapter 4. Therefore, we present a translation by translation rules and a proof of correctness with respect to the semantics of RSDS. In addition, we show some natural ways in which the modular structure of decomposed systems can be used to reduce the state space of the SMV model produced.

Chapter 6 compares RSDS against SCR and PVS, two other prominent methods used for developing reactive systems. The autopilot case study is used as a basis of this comparison.

Chapter 7 describes how RSDS/UML specifications are verified by model checking. A translation is defined from RSDS/UML to the language of SMV. We only model check coarse-grain RSDS/UML specifications and provide a proof of correctness with respect to the RSDS/UML semantics. Also, it gives a flavour of how future versions of RSDS/UML can be model checked. The chapter concludes with a discussion of related work.

Chapter 8 summarises and evaluates the translations presented for model checking RSDS specifications, and explores directions for future work.

Appendix A provides the SMV code for the RSDS specification of part of the fault-tolerant production cell.

Appendix B lists the SMV code for the RSDS specification of the gas burner system. Also, it lists the SMV code for the gas burner system generated from a comparable translation.

Appendix C provides the SMV and B source code for the autopilot system generated using the RSDS method.

Appendix D gives the SMV code for the RSDS/UML specification of the gas burner system and a part of a railway system.

CHAPTER 2

Model Checking: An Overview

Model checking consists of representing a system as a finite model in an appropriate logic and checking whether the model satisfies some desired properties. If the model does not satisfy a property, a counter-example is produced, that is, a trace that outlines the system behaviour that led to that contradiction. Model checking is fully algorithmic, that is, it is an entirely automatic approach that requires no user interaction. The system model is expressed as a transition system. The properties to be verified are expressed as either temporal logic formulas or as automata. There are three distinct approaches for implementing model checking algorithms: the temporal logic approach based on fixed point-based algorithms, the automata-theoretic approach that is based on proving language containment and the tableau approach that constructs a proof-tree.

Initially, the system model was restricted to finite systems in order for the algorithm to terminate. Although this restriction is no longer absolute, it still requires systems to have "medium-sized" state spaces and simple descriptions of behaviour due to the *state space explosion problem*. The term "medium-sized" refers to systems with at most 10^8 reachable states [BCM90]. The state space explosion problem is the tendency of the state space to become very large as there is an exponential relation between the number of variables or the number of components that constitute a system, and the size of the state space. For example, the complexity of verifying a property of a system is doubled when a new variable is added to the model. A system with a "large" state space is one with greater than 10^8 reachable states which cannot be model checked by tools implemented using explicit state enumeration methods [BCM90]. Researchers in the field of model checking are mainly concerned with developing methods and data structures for overcoming this problem. These are approximately grouped into the following categories: symbolic model checking (it can model check systems with in excess of 10^{20} states [BCM90]), on-the-fly model checking, reduction and compositional reasoning.

In this thesis we are interested in verifying the temporal logic formulas expressed for RSDS specifications. RSDS specifications model reactive systems by using finite statemachines and invariants. Model checking is an appropriate verification approach for reactive systems for the

following reasons. These systems have states whose descriptions are short (i.e. in RSDS states just have a name) and easily manipulated because the intricacies lie more in the control rather than in the data [Wol95]. The behaviour of these systems is described in terms of their possible interaction with their environment rather than by transformations on complex data. We discuss which logic and hence which model checking approach we have chosen for this work.

In this chapter, we start by presenting the model checking problem and outline several possible implementations for solving it. Before we discuss the details of the model checking algorithms, we describe the notation used to represent system models and the different temporal logics that can be used to express the properties to be verified as formulas. The performance of these algorithms can be improved by applying techniques for reducing the state space explosion problem. From these, we discuss some of the most commonly used. We conclude with a brief overview of some of the model checking tools available and discuss the reasons for using the SMV model checker for this work.

2.1 The Model Checking Problem

In [MOSS99], the model checking problem is shown to be specified in two ways:

The global model checking problem: For a given finite model M and a formula ϕ, establish the set of states in M that satisfy ϕ.

The local model checking problem: For a given finite model M, a formula ϕ and a specific state s, establish whether s satisfies ϕ.

These definitions are closely related [MOSS99]: the solution generated by the global model checking problem includes the solution generated by the local model checking problem. However, the global and local model checkers are used for different applications. For example, applications that are disabled by the state space explosion problem might prefer the use of local model checking as it inspects and constructs only a small part of the model. Global model checking is required by some applications that want to verify global properties such as liveness, that usually requires access to all the states in the model.

There are several ways in which model checking is implemented. The three most common approaches are:

Temporal logic approach: In this approach the formulas are expressed using temporal logic and their semantics are inductively computed on the finite model. The model is expressed as some transition system. This results in a global model checker that works best for branching-time temporal logics.

Automata-theoretic approach [MV86, VW94, CPS93, VBW94]: The formulas, as well as the model, are expressed as automata in this approach. The model checking problem is reduced to an inclusion problem between automata. It is best suited to model checking linear-time temporal logic formulas and solves the global model checking problem.

Tableau Approach [SW91, Cle90]: This approach solves the local model checking problem using sub-goals. Its aim is to find a proof tree that determines whether a given state has the given property. If no proof tree can be found, then the given state does not satisfy the given property. Only a small part of the state space is inspected.

Figure 2.1 gives an overview of the model checking approaches and indicates which temporal logics they are best suited to and which model checking problems they solve. Some researchers [VBW94, CGH97] have shown that it is possible for the model checking approaches to model check formulas of other logics that are not indicated in the figure. However, these are less common.

Model checking approaches	Global	Local	Branching-Time	Linear-Time
Temporal logic	√		√	
Automata-theoretic	√	√		√
Tableau		√	√	√

Figure 2.1: Properties of model checking approaches as given in [MOSS99].

In this thesis we want to verify the temporal logic properties of RSDS specifications by model checking. Since these are expressed using branching-time temporal logic (see Section 3.1.1) and include global properties, the temporal logic model checking approach is the most obvious choice to adopt. Therefore, the RSDS method uses a model checker that implements the temporal model checking approach for branching-time logic. We present, in the following sections, how models are described and what temporal logics are used for expressing properties for model checking, and the temporal model checking approach for branching-time properties is provided.

2.2 Models

The behaviour of a system is usually described as a discrete model for the purpose of model checking. The models use graph structures to represent the system behaviour, where the nodes represent the system states and the arcs between the nodes represent the possible transitions. Additional information is included in the definition of some graph structures to provide a more detailed description of the model. There are three types of augmented graph structures typically used [MOSS99]: *Kripke structures* (also known as transition systems), whose nodes are annotated with atomic propositions; *labelled transition systems* (LTS) whose arcs are annotated by actions and the *Kripke transition systems* that combine Kripke and LTS. For reasons of simplicity, we choose to describe only Kripke structures in this chapter.

2.2.1 Kripke Structures

A Kripke structure over a set of atomic propositions P is a triple (S, R, L), where S is a set of states, $R \subseteq S \times S$ is a transition relation and $L : S \to 2^P$ is a function that labels each state with the set of atomic propositions that hold at that state. S and P are finite for model checking. Although atomic propositions are formally just symbols, they represent some basic local properties of states. The propositions `true` and `false` are always contained in the set P and for any state s, `true` $\in$ `L(s)` and `false` $\notin$ `L(s)`. For model checking, the execution of a system starts at the

initial state, that is defined as $s_0 \in S$, and the graph is said to be *rooted.* A Kripke model for a system is *total* if for all $s \in S$ there is a $t \in S$ such that $(s, t) \in R$, otherwise it is *partial.* Figure 2.2 illustrates a Kripke structure for a model of a system where the atomic propositions p, q, r are true in the states which they are contained.

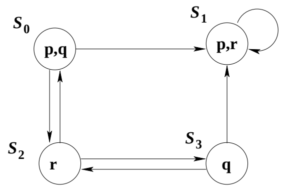


Figure 2.2: A Kripke structure for a model of a system with propositional atoms p, q, r.

2.3 Logics Used for Properties

Pnueli [Pnu77, Pnu86, MP92] proposed that temporal logic be used for reasoning about reactive systems. A temporal logic is a formal system, where the propositional operators have been augmented with tense operators that describe changes in time. We can express properties that cannot be expressed using propositional logic, such as, the formula Fq that is true in the present if at some moment in the future, q is true. Another example is the formula Pq that is true in the present if q is true at some moment in the past. In this thesis, we only consider formulas with future operators because RSDS invariants are described using future operators only and there are very few efficient model checkers [PPSM03] available for verifying formulas with past operators. Moreover, it has been shown [LS95] that most of the formulas expressed with past operators can be translated into formulas with only future operators.

Numerous temporal logics have been defined for various uses. These are usually grouped according to two distinct views of time: *linear-time logics* that consider time as a sequence of time instances and *branching-time logics* that for a given point in time offer several alternative future states. An additional characteristic of time concerns the view of time as being continuous or discrete. In this thesis, we are concerned with model checking discrete systems. Computational Tree Logic (CTL) is a subset of modal branching-time logic that is discrete. We also present Linear-Time Temporal Logic (LTL) that is a linear-time logic and CTL*. CTL* is a uniform logical framework that was introduced by [EH86] for comparing the expressive power of both CTL and LTL as both CTL and LTL properties can be expressed in CTL*.

2.3.1 CTL

Time is modelled by CTL as an infinite tree-like structure with many future paths. The Kripke structure in Figure 2.2 can be unwound into a computational tree that is visualised in Figure 2.3. Thus, the future is nondeterministic as any of these paths can be considered as the future path. Temporal operators are used to reason over the states and paths of the structure and can only occur in pairs. The first of the pair reason over paths, where A means "along all paths" and E means "along at least one path". The second temporal operator of the pair reason over states, where F means "some future state", G means "all future states", U means "until" and X means "next state". Figure 2.4 illustrates the precise meaning of pUq, where p holds continuously until the next occurrence of q, that is from S_2 to S_7. E is dual to A as $\exists$ is to $\forall$ in predicate logic.

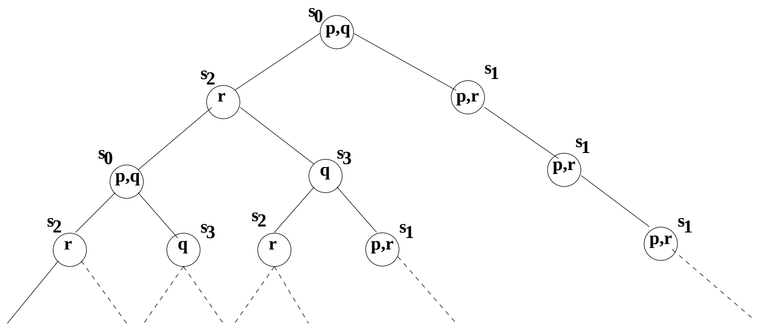


Figure 2.3: Unwinding the system in Figure 2.2 into an infinite tree to show all the computation paths starting from s_0.

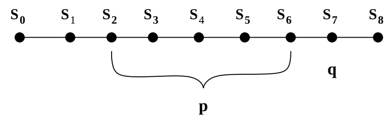


Figure 2.4: The meaning of p U q in the CTL semantics.

Syntax

The syntax of CTL formulas is defined inductively via a Backus-Naur form as [HR00]:

$$\begin{aligned}\phi \ ::= \ & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid AX\phi \mid \\ & EX\phi \mid A(\phi U \phi) \mid E(\phi U \phi) \mid AG\phi \mid EG\phi \mid AF\phi \mid EF\phi.\end{aligned}$$

where p ranges over a countable set of atomic formulas. The symbols $\bot$ and $\top$ denote false and true respectively. If ϕ is a CTL formula, then so is $\neg\phi$, $(\phi \wedge \phi)$ and so on.

Semantics

The semantics of CTL define the truth and falsehood of formulas with respect to a model expressed as a Kripke structure. Let $\mathcal{M} = (S, R, L)$ be a model for CTL, defined as the Kripke structure in section 2.2.1. A *path* π of a model $\mathcal{M}$ is an infinite sequence of states $(s_0, s_1, s_2, ..) \in S^\omega$ such that each successive pair (s_i, s_{i+1}) is an element of R. The notation π^i is used to denote the suffix of π starting at s_i. Every path is a maximal linearly ordered subset of the tree structure unwound

from s_0. For any state s, $\mathcal{M}, s \vDash \phi$ denotes that ϕ holds in state s. Structural induction is used on all CTL formulas in order to define the satisfaction relation $\vDash$ as given in [HR00].

1. $\mathcal{M}, s \vDash \top$ and $\mathcal{M}, s \nvDash \bot$ for all $s \in S$.
2. $\mathcal{M}, s \vDash p$ iff $p \in L(s)$.
3. $\mathcal{M}, s \vDash \neg\phi$ iff $\mathcal{M}, s \nvDash \phi$.
4. $\mathcal{M}, s \vDash \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ and $\mathcal{M}, s \vDash \phi_2$.
5. $\mathcal{M}, s \vDash \phi_1 \vee \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$.
6. $\mathcal{M}, s \vDash \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, s \nvDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$.
7. $\mathcal{M}, s \vDash AX\phi$ iff for all s_1 such that $(s, s_1) \in R$ we have $\mathcal{M}, s_1 \vDash \phi$.
8. $\mathcal{M}, s \vDash EX\phi$ iff for some s_1 such that $(s, s_1) \in R$ we have $\mathcal{M}, s_1 \vDash \phi$.
9. $\mathcal{M}, s \vDash AG\phi$ holds iff for all paths π, where s_0 equals s, and all s_i along the path, we have $\mathcal{M}, s_i \vDash \phi$.
10. $\mathcal{M}, s \vDash EG\phi$ holds iff there is a path π, where s_0 equals s, and all s_i along the path, we have $\mathcal{M}, s_i \vDash \phi$.
11. $\mathcal{M}, s \vDash AF\phi$ holds iff for all paths π, where s_0 equals s, there is some s_i such that $\mathcal{M}, s_i \vDash \phi$.
12. $\mathcal{M}, s \vDash EF\phi$ holds iff there is a path π, where s_0 equals s, there is some s_i such that $\mathcal{M}, s_i \vDash \phi$.
13. $\mathcal{M}, s \vDash A(\phi_1 U \phi_2)$ holds iff for all paths π where s_0 equals s, π satisfies $\phi_1 U \phi_2$.
14. $\mathcal{M}, s \vDash E(\phi_1 U \phi_2)$ holds iff there is a path π, where s_0 equals s, and that path satisfies $\phi_1 U \phi_2$.

Two CTL formulas ϕ and ψ are semantically equivalent if any state in the model satisfies ϕ, precisely when it satisfies ψ. We write this as $\phi \equiv \psi$.

2.3.2 LTL

Time is modelled in LTL as a single infinite future path. Therefore, LTL has no path quantifiers such as A and E of CTL. However, the temporal operators concerning the states of the model are the same as those of CTL. It may seem that LTL is less expressive than CTL. However, this is not true as LTL allows the nesting of boolean connectives and modalities in a way that is not allowed in CTL.

Syntax

The syntax for LTL is expressed in Backus Naur form as follows [HR00]:

$$\phi \ ::= \ p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid X\phi \mid (\phi \ U \ \phi) \mid G\phi \mid F\phi$$

where p is any propositional atom.

An LTL formula is evaluated on a single path, or on a set of paths. A formula ϕ holds on a set of paths if ϕ holds on every path in the set.

Semantics

For a CTL model $\mathcal{M} = (S, R, L)$, a path $\pi = (s_0, s_1, s_2, ...)$ satisfies an LTL formula via the satisfaction relation $\vDash$ for LTL formulas as follows:

1. $\pi \vDash \top$.
2. $\pi \vDash p$ iff $p \in L(s_1)$.
3. $\pi \vDash \neg\phi$ iff $\pi \nvDash \phi$.

4. $\pi \vDash \phi_1 \wedge \phi_2$ iff $\pi \vDash \phi_1$ and $\pi \vDash \phi_2$.
5. $\pi \vDash X\phi$ iff $\pi^2 \vDash \phi$.
6. $\pi \vDash G\phi$ holds iff, for all $i \geq 0,\ \pi^i \vDash \phi$.
7. $\pi \vDash F\phi$ holds iff, for some $i \geq 0,\ \pi^i \vDash \phi$.
8. $\pi \vDash \phi_1\ U\ \phi_2$ holds iff there is some $i \geq 0,\ \pi^i \vDash \phi_2$ and for all $j = 0, ..., i-1$ we have $\pi^j \vDash \phi_1$.

Two LTL formulas are semantically equivalent, that is $\phi \equiv \psi$, if they are true for the same paths.

2.3.3 CTL*

CTL* is a logic that embodies CTL and LTL. It includes the nesting of modalities and boolean connectives (allowed by LTL) before applying the path quantifiers E and A (allowed by CTL).

Syntax

The syntax of CTL* is defined inductively in the Backus Naur form. The definition of CTL* formulas is divided into two types of formulas: state formulas, that are true in a particular state and path formulas that are true along a particular path. The definition of each class depends on the definition of the other.

- State formulas:

  $$\phi \ ::= \ \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid A[\alpha] \mid E[\alpha]$$

  where p is any atomic formula and α is any path formula. If α is a path formula then $A[\alpha]$ is a state formula, and similarly with $E[\alpha]$.

- Path formulas:

  $$\alpha \ ::= \ \phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha\ U\ \alpha) \mid G(\alpha) \mid F(\alpha) \mid X(\alpha)$$

  where ϕ is any state formula. If ϕ is a state formula, then ϕ is also a path formula.

Semantics

For a Kripke model $\mathcal{M}$, we describe the semantics [CGP99] of CTL* by defining inductively the relation $\vDash$ as follows. We assume that ϕ_1 and ϕ_2 are state formulas and ψ_1 and ψ_2 are path formulas.

1. $\mathcal{M}, s \vDash p$ iff $p \in L(s)$.
2. $\mathcal{M}, s \vDash \neg\phi_1$ iff $\mathcal{M}, s \nvDash \phi_1$.
3. $\mathcal{M}, s \vDash \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ and $\mathcal{M}, s \vDash \phi_2$.
4. $\mathcal{M}, s \vDash \phi_1 \vee \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$.
5. $\mathcal{M}, s \vDash E\phi_1$ iff there is a path π from s such that $\mathcal{M}, \pi \vDash \phi_1$.
6. $\mathcal{M}, s \vDash A\phi_1$ iff for every path π starting from s, $\mathcal{M}, \pi \vDash \phi_1$.
7. $\mathcal{M}, \pi \vDash \phi_1$ iff s is the first state of π and $\mathcal{M}, s \vDash \phi_1$.
8. $\mathcal{M}, \pi \vDash \neg\psi_1$ iff $\mathcal{M}, \pi \nvDash \psi_1$.
9. $\mathcal{M}, \pi \vDash \psi_1 \wedge \psi_2$ iff $\mathcal{M}, \pi \vDash \psi_1$ and $\mathcal{M}, \pi \vDash \psi_2$.
10. $\mathcal{M}, \pi \vDash \psi_1 \vee \psi_2$ iff $\mathcal{M}, \pi \vDash \psi_1$ or $\mathcal{M}, \pi \vDash \psi_2$.

11. $\mathcal{M}, \pi \vDash X\psi_1$ iff $\mathcal{M}, \pi^1 \vDash \psi_1$.
12. $\mathcal{M}, \pi \vDash F\psi_1$ iff there exists a $k \geq 0$ such that $\mathcal{M}, \pi^k \vDash \psi_1$.
13. $\mathcal{M}, \pi \vDash G\psi_1$ iff for all $i \geq 0,\ \mathcal{M}, \pi^i \vDash \psi_1$.
14. $\mathcal{M}, \pi \vDash \psi_1 U \psi_2$ iff there exists a $k \geq 0$ such that $\mathcal{M}, \pi^k \vDash \psi_2$ and for all $0 \leq j < k,\ \mathcal{M}, \pi^j \vDash \psi_1$.

The minimum set of operators required to express any other CTL* formula is: $\vee$, $\neg$, X, U and E.

2.3.4 Expressivity of CTL*, CTL and LTL

CTL* was introduced as a logical framework for comparing the expressiveness of CTL and LTL. Both CTL and LTL are considered as subsets of CTL*. This is true for LTL, even though the syntax of LTL does not include path quantifiers, the semantics of LTL is described in terms of all paths. Therefore, an LTL formula α can be expressed in CTL* as $A[\alpha]$. CTL is also a subset of CTL* because of the CTL* fragment that restricts the form of path formulas to:

$$\alpha \ ::= \ (\alpha\ U\ \alpha) \mid G(\alpha) \mid F(\alpha) \mid X(\alpha)$$

We discuss the expressivity of CTL and LTL in general for model checking with respect to the model and the properties (formulas). Then, we discuss the suitability of CTL and LTL for describing models and properties of reactive systems.

A logic must be able to handle non-determinism in models, where states can have a number of successors [Wol95, Gia99]. One way in which non-determinism arises is by underspecification of some system component or of the environment. CTL handles non-determinism explicitly, where the model of time is a tree. CTL properties contain operators that express that something holds on some or all of the possible futures. This selectivity is what makes branching time logic suitable for expressing properties of reactive systems [MOSS99, BBF$^+$01]. Formulas are interpreted on the computational tree defined by the finite-state model of the program. In contrast, LTL handles non-determinism implicitly, where the model of time is linear. Thus, a program is seen as a set of possible executions. Formulas are interpreted on the program executions. According to [Wol95], the linear model checking approach is better adapted to systems whose properties reason about the execution of a program. On the other hand, the branching approach is better adapted to systems whose properties reason about the structure of the program.

For property specification in model checking, the expressiveness of the logics CTL and LTL must be compared. Figure 2.5 illustrates which formulas can be expressed only in CTL, LTL and CTL*. In CTL, the formula $AGEFp$, means that regardless of what state the program enters, there exists a computation that leads to a state where p holds. This formula can be used to express reachability properties. The negation of a reachability property is a safety property [BBF$^+$01]. For reactive systems, reachability and safety properties are significant for determining their correctness. However, the formula $AGEFp$ and its negation cannot be expressed in LTL.

A CTL* formula that can be expressed in LTL but not in CTL is $A[GFp \rightarrow Fq]$, meaning that if there are infinitely many p along the path, then there is an occurrence of q. Fairness constraints are usually expressed in this form. Since CTL cannot express this property (it does not allow the nesting of G and F without a path quantifier), it must be extended in order to do so. These extensions are easily implemented, however, the complexity of these model checkers

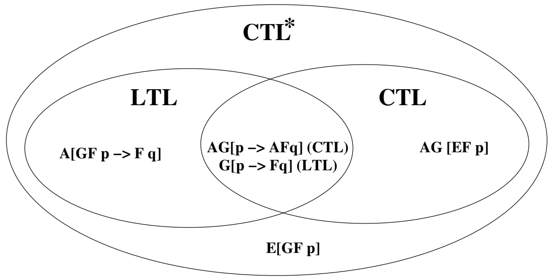


Figure 2.5: The expressiveness of CTL, LTL and CTL*

increases [BBF$^+$01]. Several tools, such as the one that we have decided to use for this thesis (SMV), consider fairness property to be part of the model, rather than expressing the property as a formula in an extended CTL. This approach has no effect on the model but [BBF$^+$01] argue that there are losses with respect to expressivity while gaining only slightly in simplicity.

Overall, CTL* seems to be the best choice for describing properties of reactive systems. However, even though CTL* is more expressive, the algorithm for model checking CTL* is less efficient than that of model checking CTL. Therefore, we choose to use CTL instead because the model checking algorithm for CTL is more efficient than that of LTL or CTL* (see Table 2.1). Also, we prefer CTL over LTL because it handles non-determinism explicitly.

2.4 Temporal Model Checking Approach

The temporal model checking approach was proposed independently by Clarke and Emerson in [CE81] and by Quielle and Sifakis in [QS82]. The algorithms that they implemented were for verifying branching-time logic properties. In this section we describe only the temporal model checking approach for verifying CTL formulas on the transitional behaviour of the model. In [CGH97], the authors show how LTL model checking can be reduced to CTL model checking.

2.4.1 Verifying Branching-Time Logic Properties

The most effective algorithm [HR00] for model checking CTL formulas is based on a fixed point characterisation of CTL operators. Fixed point theory can be exploited for CTL model checking as CTL formulas are satisfied by a set of states (instead of paths). The fixed points can be easily computed and they ensure that the model checking algorithm terminates after a maximum of $|S|$ rounds.

The algorithm used is recursive. It determines all the states of a given structure that are reachable by applying the transition relation and that also satisfy the given CTL formula. Initially, the set of states is computed that satisfy the atomic sub-formulas that have no path or state quantifiers. Next, fixed points of monotonic functionals for the CTL quantifiers are applied in rounds until a fixed point is obtained. The algorithm must show that the initial state of the

structure is also an element of the set of reachable states computed. If this is so, then the structure is a model of the CTL formula.

Preliminaries

A function $F : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ on the power set of S, where S is a set of states, is *monotone*, if $x \subseteq y$ implies $F(x) \subseteq F(y)$ for all subsets x and y of S. A subset x of S is considered a *fixed point* of F if $F(x) = x$. The Knaster-Tarski Theorem proves that for a finite set $S = \{s_0, s_1, ..., s_n\}$ with $n + 1$ elements, if $F : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ is a monotone function, then:

- $F^{n+1}(\emptyset)$ is the least fixed point of F, where F^{n+1} means that the function is applied $n + 1$ times and,
- $F^{n+1}(S)$ is the greatest fixed point of F.

This theorem asserts the existence of the least and greatest fixed points for monotonic functions and also provides a way for computing them. For example, to compute the least fixed point, an algorithm repeatedly applies the function F to the empty set until the result becomes invariant. Therefore, this theorem guarantees that the algorithm terminates as there is always a fixed point for monotone functions. Furthermore, the worst-case number of iterations is $n + 1$ for a set with $n + 1$ elements.

The algorithm

The model checking algorithm computes the set of states $s \in S$ satisfying ϕ, given a CTL formula ϕ and a CTL model $M = (S, R, L)$. The set of states computed is written as $[\![\phi]\!]$. The computation starts with the states that satisfy the atomic sub-formulas of ϕ, which contain no path or state quantifiers. These formulas ϕ are either $\bot$, $\top$ or p and $[\![\phi]\!]$ is computed directly, or are smaller sub-formulas combined with CTL operators. An example of such a formula is $\psi_1 \wedge \psi_2$ where the algorithm computes the sets $[\![\psi_1]\!]$ and $[\![\psi_2]\!]$ and then combines them in this case by taking the intersection to obtain $[\![\psi_1 \wedge \psi_2]\!]$. For formulas that contain path or state quantifiers, each formula is computed in concurrence with the semantics of the CTL operators used. For example, $EX\psi$ is computed by first obtaining the set $[\![\psi]\!]$ and then by computing the set of states that have a transition to a state in $[\![\psi]\!]$. The meaning of the temporal operators are expressed by fixed points as there is no direct computations of these. We will discuss in detail how the fixed points are used in the algorithm to characterise the EG operator.

The pseudo-code for the model checking algorithm as described in [HR00] is as follows.

```
function $SAT(\phi)$
begin
  case
    $\phi$ is $\top$ : return $S$
    $\phi$ is $\bot$ : return $\emptyset$
    $\phi$ is atomic : return $\{s \in S | \phi \in L(s)\}$
    $\phi$ is $\neg\phi_1$ : return $S - SAT(\phi_1)$
    $\phi$ is $\phi_1 \wedge \phi_2$ : return $SAT(\phi_1) \cap SAT(\phi_2)$
    $\phi$ is $\phi_1 \vee \phi_2$ : return $SAT(\phi_1) \cup SAT(\phi_2)$
    $\phi$ is $\phi_1 \rightarrow \phi_2$ : return $SAT(\neg\phi_1 \vee \phi_2)$
    $\phi$ is $AX\phi_1$ : return $SAT(\neg EX \neg\phi_1)$
    $\phi$ is $EX\phi_1$ : return $SAT_{EX}(\phi_1)$
    $\phi$ is $A(\phi_1 U \phi_2)$ : return $SAT(\neg(E[\neg\phi_2 U(\neg\phi_1 \wedge \neg\phi_2)] \vee EG\neg\phi_2))$
    $\phi$ is $E(\phi_1 U \phi_2)$ : return $SAT_{EU}(\phi_1, \phi_2)$
    $\phi$ is $EF\phi_1$ : return $SAT(E(\top U \phi_1))$
    $\phi$ is $EG\phi_1$ : return $SAT_{EG}(\phi_1)$
    $\phi$ is $AF\phi_1$ : return $SAT(\neg EG \neg\phi_1)$
    $\phi$ is $AG\phi_1$ : return $SAT(\neg EF \neg\phi_1)$
  end case
end begin
```

The algorithm only deals with an adequate[1] set of temporal operators: EU, EG and EX. The main function SAT (which stands for satisfies) takes as input a CTL formula and returns the set of states that satisfy the formula. The algorithm calls functions SAT_{EU}, SAT_{EG} and SAT_{EX} each time it encounters EU, EG and EX respectively in the parse tree for the given formula. It is assumed that SAT can access any part of the model.

The semantics of $EG\phi$ assert that $s_0 \vDash EG\phi$ holds iff there exists a computational path such that $s_i \vDash \phi$ holds *for all* $i \geq 0$. This could also be expressed by the equivalence:

$$EG\phi = \phi \wedge EXEG\phi$$

that expresses that $EG\phi$ holds if ϕ holds and $EG\phi$ in one of the next states to the current state. This equivalence can be proved from the semantic definition of $EG\phi$. The semantic definition of $EX\psi$ can be used to rewrite the equivalence in the following way:

$$[\![EG\phi]\!] = [\![\phi]\!] \cap \{s | \text{ exists } s' \text{ such that } (s, s') \in R \text{ and } s' \in [\![EG\phi]\!]\}$$

This does not seem like a good way for computing $EG\phi$ as it is a circular definition. However, it can be shown that

$$F(X) = [\![\phi]\!] \cap \{s | \text{ exists } s' \text{ such that } (s, s') \in R \text{ and } s' \in X\}$$

is a monotone function, and that $[\![EG\phi]\!]$ is the greatest fixed point of $F(X)$. The pseudo code for SAT_{EG} computes $EG\phi$ as follows.

[1]A set of temporal operators is adequate if any CTL formula can be converted into a semantically equivalent formula that uses only those operators.

```
function $SAT_{EG}(\phi)$
local var X, Y
begin
  Y := $SAT(\phi)$;
  X := 0;
  repeat until X = Y
  begin
    X := Y;
    Y := Y $\cap$ $\{s|$ exists $s'$ such that $(s, s') \in R$ and $s' \in Y\}$
  end
  return Y
end
```

Functions for SAT_{EU} and SAT_{EX} are derived similarly.

Example

Let us consider the Kripke model illustrated in Figure 2.2 and show how to compute the set $[\![EGq]\!]$ using the fixed point functions. This set is the greatest fixpoint of the function F, i.e. $[\![EGq]\!] = F^{n+1}(S)$ where $S = \{S_0, S_1, S_2, S_3\}$. From Figure 2.2, it is clear that $[\![q]\!] = \{S_0, S_3\}$ and therefore:

$$\begin{aligned} F(X) &= [\![q]\!] \ \cap \ \{s \in S | \text{ such that } (s, s') \in R \text{ for some } s \in X\} \\ &= \{S_0, S_3\} \ \cap \ \{s \in S | \text{ such that } (s, s') \in R \text{ for some } s \in X\}. \end{aligned}$$

Since $[\![EGq]\!]$ is the greatest fixed point of F, F must be iterated on S until the process stabilises. Therefore, the first iteration consists of:

$$\begin{aligned} F^1(X) &= \{S_0, S_3\} \ \cap \ \{s \in S | \text{ such that } (s, s') \in R \text{ for some } s \in S\} \\ &= \{S_0, S_3\} \ \cap \ S \text{ since every } s \text{ has some } swhere(s, s') \in R \\ &= \{S_0, S_3\}. \end{aligned}$$

The second iteration:

$$\begin{aligned} F^2(X) &= F(F^1(S)) \\ &= \{S_0, S_3\} \ \cap \ \{s \in S | \text{ such that } (s, s') \in R \text{ for some } s \in \{S_0, S_3\}\} \\ &= \{S_0, S_3\}. \end{aligned}$$

Since the process stabilises, the greatest fixed point of F is $\{S_0, S_3\}$.

2.5 Complexity of Model Checking Algorithms

The complexity of model checking algorithms [Sch02a] is measured in order to evaluate their cost, that is, their running time, and to try to show that the algorithms are *optimal*. Optimality of algorithms is a specific term used in the theory of computational complexity for showing that no better algorithm exists at the cost of some simplifying abstractions, such as asymptotic measures and comparing performance on the worst case. In the model checking field, the complexity measure is used to compare the computational capabilities of algorithms for different temporal logics (i.e. LTL versus CTL) and for identifying any efficiency limitations.

The cost of model checking algorithms, for example, for the one we presented that solves the problem of $M \vDash \phi$ for some Kripke structure M and for a given formula ϕ, is given by the sizes

Table 2.1: Summary of complexity measures for LTL, CTL and CTL*

Temporal Logic	Program-complexity	Formula-complexity	Total Complexity
LTL	NLOGSPACE-complete	PSPACE-complete	PSPACE-complete
CTL	NLOGSPACE-complete	LOGSPACE	P-complete
CTL*	NLOGSPACE-complete	PSPACE-complete	PSPACE-complete

$|M|$ and $|\phi|$. In [Sch02a], they assume that $|\phi|$ is the number of symbols (strings) in ϕ and $|M|$ is the sum of the number of nodes and edges $|S| + |R|$.

In the early days of model checking, it was believed that model checking CTL formulas (P-complete) was easier than model checking LTL formulas (PSPACE-complete). The complexity of model checking CTL* is PSPACE-complete. In [Sch02a], the author presents an argument against this belief and shows a different approach for determining the complexity of model checking algorithms. This approach considers two parameters that contribute separately to the complexity of model checking. For the Kripke structure M and the formula ϕ, the size of M is usually large and the size of ϕ is usually small. Therefore, the size of M is seen to be more significant with respect to cost of model checking. In the complexity-theoretic framework, these parameters lead to the determination of: *program-complexity* and *formula-complexity*. Program-complexity is measured as a function of the Kripke structure M only, whereby the formula ϕ is fixed. Conversely, formula-complexity is measured as a function of ϕ only and M is fixed. Therefore, we can measure the impact of each input to the overall cost. We summarise in the table below the complexity measures for LTL, CTL and CTL* for model checking as presented in [Sch02a], where more details can be found on the results.

In [Var98], they also point out that the complexities and expressiveness of the logics do not simply indicate which logic is easier to model check. Therefore, this factor should not entirely influence our choice of model checking approach to use for the work in this thesis.

2.6 Techniques Addressing State Explosion

Although the complexity of the model checking algorithms seems tractable, in practice, the algorithms still suffer from the state space explosion problem. This is because when the model checker is implemented, the program that represents the system model can become very large as there is an exponential relation between the number of variables and the number of components. Several techniques have been proposed for reducing this problem by improving the efficiency of model checking algorithms.

2.6.1 Symbolic model checking

One way of improving the efficiency of the algorithm is to apply symbolic model checking. A *symbolic representation* is used in symbolic model checking for implicitly representing the state variables and transitions of a transition system as well as the CTL formulas. The issue of concern is no longer the size of the state space but rather the size of the symbolic representation, which may be much smaller.

One such symbolic representation used is a canonical representation of boolean functions known as Ordered Binary Decision Diagrams (OBDDs) [Bry86, Bry92]. This representation is a more efficient extension of Binary Decision Diagrams (BDDs) [Bry85] as reductions have been applied (for example, the removal of duplicate nodes) and a strict ordering has been imposed on the variables, as BDDs were not a great improvement from truth-table representations. BDDs and hence OBDDs are similar to binary decision trees, with the exception that they are directed acyclic graphs rather than trees.

OBDDs are composed of non-terminal nodes (denoted by circles) labelled with boolean variables and terminal nodes (denoted by squares) labelled with boolean values, i.e. 1 or 0. The edges that connect the nodes are either represented by dashed lines if the values of the variables are 0 or by solid lines if the values of the variables are 1. Therefore, to obtain the truth values of the boolean function, one traverses the OBDD starting from the root node, and follows the edges depending on the values of the variables to reach the values expressed by the terminal nodes. Figure 2.6 illustrates the BDD and OBDD for the function $f(a,b,c) = (a \cdot b) + c$ where $\cdot$ denotes $\wedge$, $+$ denotes $\vee$, 1 denotes $\top$ and 0 denotes $\bot$. The truth-table is also given as it corresponds to the boolean variables and values in the diagrams. The OBDD has no duplicate terminal or non-terminal nodes and imposes an order on variables that in Figure 2.6 increases along any path.

a	b	c	f(a,b,c)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

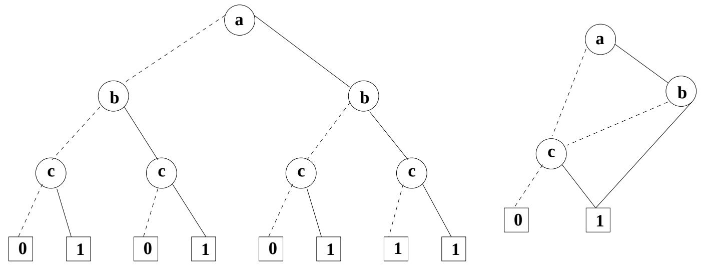


Figure 2.6: The truth table, BDD and OBDD (with ordering $a < b < c$) for $f(a,b,c) = (a \cdot b) + c$

Symbolic model checking [BCM90] uses OBDDs to represent the finite-state model of the system and checks temporal formulas directly on them. A finite-state model $\mathcal{M} = (S, R, L)$ is expressed as OBDDs in the following way. Each element of S is assigned a vector of boolean values $(v_1, v_2, ..., v_n)$, where $n = \lceil \log_2 |S| \rceil$. A set of states $\{s_1, s_2, ..., s_k\}$ is represented as the OBDD of the boolean function

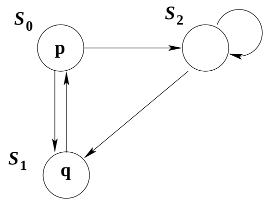
$$(v_{11} \cdot v_{12} \cdot ... \cdot v_{1n}) + (v_{21} \cdot v_{22} \cdot ... \cdot v_{2n}) + ... + (v_{k1} \cdot v_{k2} \cdot ... \cdot v_{kn})$$

The transition relation R (subset of $S \times S$) is represented by a pair of boolean vectors, one encoding the current state and the other encoding the next state. It is represented by the OBDD for the boolean function

$$(v_1 \cdot v_2 \cdot ... \cdot v_n) \cdot (v'_1 \cdot v'_2 \cdot ... \cdot v'_n)$$

and the entire transition relation is the OBDD resulting from the disjunction of such formulas.

Figure 2.7 shows how the substates of states of a simple Kripke model are represented by boolean values and functions.



set of states	representation by boolean values	representation by boolean functions
$\emptyset$		0
$\{S_0\}$	(1,0)	$p \cdot \bar{q}$
$\{S_1\}$	(0,1)	$\bar{p} \cdot q$
$\{S_2\}$	(0,0)	$\bar{p} \cdot \bar{q}$
$\{S_0, S_1\}$	(1,0),(0,1)	$p \cdot \bar{q} + \bar{p} \cdot q$
$\{S_0, S_2\}$	(1,0),(0,0)	$p \cdot \bar{q} + \bar{p} \cdot \bar{q}$
$\{S_1, S_2\}$	(0,1),(0,0)	$\bar{p} \cdot q + \bar{p} \cdot \bar{q}$
S	(1,0),(0,1),(0,0)	$p \cdot \bar{q} + \bar{p} \cdot q + \bar{p} \cdot \bar{q}$

Figure 2.7: A simple Kripke model and the representation of its substates of states.

The algorithm presented in 2.4.1 for model checking CTL formulas is suitable to be used with OBDDs. That is, OBDDs are used to represent the atomic propositions and the transition relation of the model and the algorithm returns the OBDD for all the system states that are true for the given formula. Fixed point characterisation of temporal operators also forms the basis of the symbolic model checking approach. There are a number of operations that are defined on OBDDs and these are used in order to implement the operations defined in the model checking algorithm. Relational mu-calculus has been used as a syntax for referring to fixed points in the context of boolean formulas and for describing the dependencies and interactions of fixed point invariants.

Symbolic model checking has been successfully used for model checking CTL formulas and has been shown [CGP99] to benefit LTL-model checking as well as automata-theoretic approaches. This approach has allowed several systems [BCM90] to be model checked that were previously rejected because of the size of the state space.

However, OBDD algorithms do not perform better in all cases because the size of the OBDD depends greatly on the chosen ordering of the variables. The problem of finding an ordering that generates a minimal OBDD is NP-complete. Moreover, it has been shown that for some boolean functions the size is always exponential no matter which ordering is applied [CGL93]. Nevertheless, several heuristics have been developed for choosing a suitable variable ordering, if such an ordering exists.

The complexity of the optimal symbolic model checking algorithm [Sch02a] for LTL, CTL and CTL* is PSPACE-complete and the program-complexity is also PSPACE-complete. For symbolic model checking algorithms of branching-time μ-calculus[2], the complexity of the algorithm is EXPTIME-complete [Sch02a].

2.6.2 On-the-fly techniques

On-the-fly techniques improve on the efficiency of performing reachability analysis. Reachability

[2]We have not presented this algorithm for simplicity.

analysis is a verification technique that exhaustively explores all the reachable states and transitions in the system which usually requires the entire state graph (reachability graph) of the system to be stored. This is made impossible by the state space explosion problem for many industrial systems. On-the-fly techniques describe ways of performing reachability analysis without storing the entire state graph. Instead, all possible transition sequences performed by the system are simulated in a depth first traversal of the system graph, without storing any of the states. This reduces the amount of memory required to perform verification [JJFM92].

There are two extreme strategies for handling the storage requirements of the depth-first search. The first handles the minimum storage requirement of storing the current path being explored. This reduces the memory required and still ensures that reachability analysis is performed successfully. However, the time needed to complete the search increases immensely. The second strategy handles the maximum storage requirement where states are stored as they are visited. This improves the execution time but increases the memory requirements which can lead to the impossibility of all states of some systems being stored. Several methods have been proposed that can be seen as a compromise between these two strategies.

Two further techniques:

1. *State-space caching* [GHP92] creates a restricted cache for storing chosen visited states. The cache initially stores all the states that it visits until there is no more space. After, it applies one of many replacement strategies for replacing the old states with new ones. The effectiveness of state-space caching depends on the size of the cache and the structure of the state space. Since the latter is unforeseeable, it is very difficult to find the optimal cache setup that can lead to a dramatic increase in execution time.

2. *Bit-state hashing* or *supertrace* technique [Hol95] is used when the size of system does not allow for exhaustive verification. It performs a partial search of the state space and stores the visited states in a hash table whose size depends on the memory available. A sequential bit-state technique can be applied to increase the coverage of the algorithm by performing multiple runs with statistically independent hashing functions, until the coverage level is reached. In this case, there is no time limitation, only space.

The applicability of reachability analysis is extended with the development of automata-theoretic approaches [CVWY92]. That is, it is no longer limited to detecting errors such as static deadlock. For example, LTL model checking can be reduced to reachability analysis and therefore algorithms can be provided for performing model checking on-the-fly. Examples of such algorithms are found in [CVWY92]. However, the use of on-the-fly techniques is not restricted to LTL model checking. They have also been applied to CTL.

The main advantage of on-the-fly techniques is that the algorithms execute until an error is found. Thus, the entire state space need not be explored and errors, if any, are detected quickly. In the worst case, if the system model is correct, the entire state space is explored. Therefore, on-the-fly techniques are suitable for systems detecting errors in the initial stages of design. If errors are found, counter-examples are generated to help the user with correcting them.

2.6.3 Partial-order reduction

Traditionally model checking was used for verifying hardware as the state explosion problem is especially large for software [CGP99]. The reason for this is that software tends to have less regularity of structure than hardware. In particular, the state space of asynchronous concurrent software is extremely large. These systems have distinct processes that perform the activities independently without a global clock. *Partial order reduction* techniques were invented as ways to successfully reduce the state space of asynchronous concurrent software, while still managing to prove the system properties.

An asynchronous concurrent system is usually represented by an interleaving model, that arranges all the events in a single execution into a linear order known as an interleaving sequence. Events can thus be ordered in any way with respect to one another. Properties of such systems are expressed using logics that can differentiate between interleaving sequences in which events are executed with different orders. The state space of these systems grows extremely large because all the possible interleaving of events must be considered. This is not a problem for synchronous concurrent systems, as they execute events simultaneously.

Partial order reduction techniques exploit the notion of independence of concurrently executed events, that is, that events can be executed in any order and the results are still the same. They are able to reduce the number of interleaving sequences to be considered and consequently the state space is reduced. If two or more indistinguishable interleaving sequences that are only differentiated by the order in which they are executed, then only one of these needs to be considered. This approach is based on the "partial order model of program execution" [CGP99].

Many researchers have proposed techniques for reducing the state space in this way. The first proposal of such a reduction technique was presented in [Ove81]. However, it considered a restricted concurrent system, without loops and nondeterministic choice [CGP99]. There are several partial order reduction techniques that have been implemented by model checking algorithms. They perform a selective search of the system state space, where for each state s reached during the search, a subset T is processed that contains the set of transitions enabled at s and explores only those. Techniques for identifying these subsets that are described in the literature include stubborn sets [Val90], persistent sets [God90], ample sets [Pel94], sleep sets [Val90] and the unfolding technique [McM95].

2.6.4 Other techniques

The following techniques can be used in conjunction with the techniques that we have already presented, that is, with symbolic model checking, on-the-fly verification and partial order reduction. We only give a brief description of these.

Compositional reasoning

Compositional reasoning consists of decomposing a complex system into smaller, more manageable components, and identifying local properties of these components that guarantee desired properties of the global system. The local properties must be verified first and are then combined to deduce the global properties of the system. Since only smaller substates of the system are considered at

any time, the entire system state space does not need to be explored. Therefore, this approach does not suffer from the state space explosion problem.

Numerous researchers [McM98b, BCC98, PT98, Sha98, KV98, PDH99, BCY02, LFK02] have devoted their attention to this approach as it seems to be the most promising for reducing the state space. However, there are some issues that prevent it from being so. One of the main issues that arises is knowing that local properties of components remain true when they are combined in parallel. Some specific assumptions about their environment must be made. An approach is presented in [CLM89] that models the environment of a component by another component, known as the interface process. An interface rule is used to deduce the properties of a composition by inspecting the local properties of each component.

The task of decomposing the global properties into local properties is very hard. Furthermore, it must be shown that the decomposition of these properties is correct. Automating this process is also very difficult but it is required in order for it to be used commercially. Certain heuristics exist and are still required [CW96] for efficiently decomposing systems.

More recently, [BCY02] show ways of exploiting a distributed environment for model checking, where parts of a system are model checked across a network (or grid) of computers (called node) that provide extra resources for handling the state space explosion problem. The authors present an algorithm that first partitions the given state space among the nodes, and then each node model checks its partition. They use *border states* for representing the missing parts of the state space[3]. Therefore, whenever a model checker reaches a border state, it obtains information of other nodes about the truth of formulas in that state - assumptions. These assumption might change and require a re-computation. These ideas are similar to those in [LG98].

Abstraction

Abstraction techniques [CGL94, DGG97] are used to reduce the complexity of model checking by producing a smaller representation (abstract system) of the actual system which is easier to verify. The states and transitions of the actual system are mapped to states and transitions in the abstract system. This approach appears to be suitable for reactive systems as they contain data paths, that is, simple relationships between data values. Two such techniques are: data abstraction and cone of influence reduction [CGP99].

Data abstraction consists of finding a mapping between actual data values in a system and a small set of abstract data values. By applying this mapping to states and transitions, a smaller representation of states and transitions can be produced that emulates the actual system. Therefore, it is simpler to model check the abstract model as it is smaller in size.

The cone of influence reduction technique focuses only on the variables that are referred or that influence properties to be verified. All the other variables are eliminated, which results in the generation of a smaller state transition graph. Therefore, the properties are verified on a smaller model of the system.

[3]This idea is similar to the *virtual sensors* that is introduced in Chapter 3 and 4.

Symmetry

Software systems usually have some replicated components or structure, for example, concurrent systems have identical processes, and reactive systems have similar sensors (switches) or actuators. *Symmetry* [ES96] can be used to generate reduced models of such systems. If a system has some symmetry, then there exists a nontrivial permutation group that preserves both the state labeling and the transition relation. These groups can then be used to reduce the state space of a system by defining an equivalence relation on the state space [CGP99]. The smaller models can be used to verify the temporal properties of the original model.

Induction

Induction is used for model checking a network of processes of finite-state concurrent systems [CGB86], since model checking of networks is limited due to state space explosion, lack of inference and lack of generalisation [CM02, MC03]. Inference refers to the ability to infer properties of a system with n processes from the properties of a system with less than n processes. Generalisation refers to the ability to generalise properties to components of an infinite group of processes. The problem of model checking parameterised systems is undecidable [AK86]. Nevertheless, it is sometimes possible to find an *invariant process* that represents the behaviour of a number of systems in a family. This invariant can be used to check properties that should hold true for all systems that belong to that family. Then, to ensure that the invariant is a suitable representation, an inductive argument is used.

Summary

Different techniques were presented that address the state space explosion problem when implementing the model checkers. There are three main techniques for improving the efficiency of the model checking algorithms (symbolic model checking, on-the-fly techniques and partial-order reduction) that can be applied independently, and a number of other techniques (compositional reasoning, abstraction, symmetry and induction) that can be applied in conjunction with the main techniques. We summarise the main techniques in the following table with respect to two main issues: how these techniques improve the efficiency of the model checking algorithms and to what extend they achieve this.

2.7 Overview of Model Checking Tools

Numerous tools have been developed for solving the model checking problem. These tools apply to different application domains, implement different model checking approaches, present ways of overcoming the state space explosion problem. Some of these are academic tools while others are used in industry. Some model checkers are standalone, while others are embedded within development methods. In this thesis, we want to use an existing model checker to verify the temporal properties of RSDS specifications. Therefore, in this section we provide a brief description of a selection of these tools and provide references for others. The factors that have influenced our

Table 2.2: Summary of main techniques addressing the state space explosion problem

	How efficiency is improved	**To what extend**
Symbolic model checking	Uses OBDDs for implicit representation of states and variables	Can model check 10^{20} states, however size of OBDDs depends on ordering
On-the-fly techniques	Performs reachability analysis without storing the entire state graph	Best case: execute until error is found. Worst case: if system model is correct, entire state space is explored.
Partial order reduction	Exploits the notion of independence of concurrently executed events	Depends on how many processes are used. It gaurantees low message complexity $O(N \times log(N))$

choice for this selection are: the popularity of a tool and its relevance to our work, that is how suitable it is for model checking RSDS specifications.

2.7.1 Tools of the temporal logic approach

There are a few such model checkers that include SMV, VIS, Murϕ [D. 96].

SMV

Symbolic Model Verifier (SMV) [McM92a] is a tool for symbolically model checking CTL formulas of finite-state systems. It provides a language for describing a system as a number of modules that can be composed synchronously or asynchronously. Asynchronous composition means that the modules are interleaved arbitrarily which is useful for describing communication protocols and asynchronous systems. Also, the SMV language allows the use of non-deterministic expressions for modelling the environment or for the purpose of abstraction. When SMV model checks a system against a set of CTL formulas, it produces as output: either the word “true” if all the properties hold in the system model, or the word “false” with a counter-example that outlines the steps leading to the violation of the property. SMV’s model checking algorithm was the first to use BDDs and showed that it could exhaustively check systems with large state spaces as in [BCM90]. Moreover, it offers several strategies for ordering the boolean variables in order to improve the efficiency of the algorithm. SMV has been used for verifying both hardware [MS91, CGH$^+$93, CGL94] and software systems [WVF95, CAB$^+$98].

VIS

Verification Interacting with Synthesis (VIS) [RGA$^+$96] is a verification and synthesis tool that is applicable to finite-state hardware systems. Verilog [Ver] is used as a front end of the tool with the extended features of a non-deterministic construct and symbolic variables, that is, representing and analysing the value of variables symbolically. VIS uses the efficient algorithms in [CGMZ95] for model checking, including one for improving the efficiency of checking invariants. Also, it provides further techniques to improve the efficiency of verification such as abstraction, equivalence checking of two designs and structural pruning techniques (that is, techniques for eliminating part of the network that are not affected by the invariant being checked). Moreover, VIS can be considered as a platform for developing new verification algorithms as it consists of 18 packages that supply a set of routines for manipulating data structures and a set of related functions that can be performed. Some industrial designs and sequential circuits have been verified with VIS in order to test its capabilities.

2.7.2 Tools of the automata-theoretic approach

Tools for the automata-theoretic approach include: SPIN, COSPAN, FDR [Ros98].

SPIN

SPIN [HP96, Hol97] is a model checking tool that was mainly developed for simulating and verifying distributed systems. It has an imperative modelling language called PROMELA that resembles C. The semantic model of PROMELA is based on finite automata. The properties to be verified are expressed as LTL formulas (known as "correctness claims") that are converted to negative claims and converted into Büchi automata. The intersection of a claim and the automaton representing the system model is computed and if it is empty, then the positive claim is satisfied for the given system. PROMELA assigns to each state a label of the possible two: "progress" and "accept", which are hidden from the user. These labels are required for SPIN to ensure that a cycle in the system contains at least one progress state and no accept states. The progress state ensures that a useful step will be performed in any infinite execution of a system. The accepting states are used in the verification procedure of LTL formulas and assist the generation of a counterexample when a system does not satisfy a claim. Also, SPIN checks automatically some basic properties such as absence of deadlock.

An optimised nested depth-first search algorithm is used to compute SPIN's verification procedure. It is compatible with all modes of verification available with the tool for dealing with large state spaces such as exhaustive search, bit-state hashing and partial order reduction techniques. The conversion of LTL formulas to Büchi automata is based on on-the-fly construction as described in [GPVW95].

COSPAN

COSPAN [AK95, HHK96] adopts the automata-theoretic approach for model checking. It has a modelling language called S/R (selective/resolution). Several widely used notations for hardware description, such as Verilog, have developed front-end interfaces for COSPAN as a way of integrating the model checker. Its semantic model is based on ω-automata. A system to be verified is modelled as a collection of such automata. Properties are expressed as ω-automata and to facilitate their construction a library of parameterised automata is provided. COSPAN provides a top-down development process through successive refinements. The performance of the algorithm is improved by either symbolic or explicit state-enumeration algorithms. The former uses BDDs, while the latter uses caching and bit-state hashing options in addition to minimisation algorithms. Moreover, COSPAN supports other reduction strategies such as automated localisation reduction (computes the cone of influence of a property) and symmetry reduction.

2.7.3 Model checking real-time and hybrid systems

Model checking has been used to verify real-time and hybrid systems. Real-time systems are systems that must react to stimuli from the environment within a strict time limit. UPPAAL [LPY97, BDL$^+$01] is a tool for modelling and verifying real-time systems and is based on constraint-solving and on-the-fly techniques. Recently, it has been extended with more efficient algorithms and its modelling notation describes hierarchical structures, similarly to those in UML statechart diagrams, for better system design. Kronos [DOTY96, Yov97] is another tool for model checking real-time systems. It implements the symbolic model checking algorithm and verifies properties expressed in the real-time temporal logic TCTL [ACD93]. Both tools model the real-time systems as timed automata, that is finite-state machines extended with real variables (clocks) for expressing the timing constraints on the delays between events.

Hybrid systems are digital systems that express both discrete and continuous change. They interact with each other as well as with an analog environment. The hybrid automaton is a finite automaton with a finite number of real-valued variables that change continuously. It is commonly used for modelling the mixed discrete-continuous behaviour. HyTech [HHWT97] is a symbolic model checker for verifying linear hybrid automata. It takes as input a set of linear hybrid automata given in a textual form and then computes subsets of the global state space. As a result of this computation, HyTech provides information on the behaviour of the system and provides error diagnostics. It can also handle parametric analysis i.e. the analysis can give the parameter values for which the property holds.

According to the survey of tools described in [BBF$^+$01], Kronos allows the verification of liveness properties for TCTL formulas and is not restricted, as UPPAAL and HyTech are, to the verification of reachability properties. HyTech in particular, does not apply model checking to temporal logic. Instead, subsets of states are built by the user to be computed by combinations of basic constraints. UPPAAL has a very friendly graphical user interface, unlike Kronos, but its main weakness concerns its specification language as it allows only binary synchronisations to be described.

2.7.4 Applying a combination of model checking approaches

Model-checking Kit

There are a number of modelling languages and model checkers available that are geared towards certain applications and have certain strengths and weaknesses. In [SSE03], a number of model checkers are applied to the same system to determine which model checker is best suited, as there is no single model checker that is ideal. This resulted in the development of the Model-checking Kit that provides a variety of petri net modelling notations for specifying finite-state systems, and a variety of model checkers (such as PEP tool [Pep], PROD [Pro], SMV and SPIN) for verification. The basic language of the Model-checking Kit is 1-Safe Place/Transition Petri nets as they are simple, with no variants, and can be dealt with by a number of verification techniques. Potentially, the performance of the different model checkers can be compared. However, there are losses in performance as the optimisations of the model checkers in the Kit are based on their specific modelling language and are usually impaired during translation.

2.7.5 Model checking programs

Some tools model check programs written in a given programming language. These include: Bandera [CDH$^+$00, HD01] and SLAM [BR01]. They do not require that the system model be represented in their specific modelling language. They might extract a model from the code, and this is usually done automatically.

BLAST

Berkeley Lazy Abstraction Software (BLAST) [HJM$^+$02] is a tool that uses lazy abstraction to construct correctness proofs for temporal-safety properties of proof-carrying code (PCC). PCC is a certification mechanism that ensures the correctness of the code. This consists of the code being annotated with loop invariants, pre- and post-conditions, and proof of correctness of a verification condition. From these annotations and code, the user can construct a verification condition and checks the supplied proof of correctness. Model checking is used to guide the generation of proofs.

VeriSoft

VeriSoft [God97a, God97b] is a tool for model checking systems made up of several concurrent processes that are implemented in programming language such as C or C++. Every process in the system is mapped to a UNIX process and the execution of these processes is controlled by a scheduler, which is an external process. The scheduler selects to view the visible operations of a process and can explore one transition in the state space. The depth of the search is limited to avoid state-less searches lost in cycles. If a violation is detected for the deadlock or/and assertions, the search is stopped and a trace is displayed. VeriSoft checks for deadlock, assertions, divergences and livelocks. A divergence occurs when no visible operation is executed within a time limit. A livelock occurs when no transition of a process is enabled (i.e. a process is blocked) during a sequence of successive states where the number of successive states in this sequence is greater than that defined by the user.

2.7.6 Methods that embed existing model checkers

Many development methods [BH99, Win01] and verification systems [Sha96, LB03] have realised the importance of model checking for debugging and verification purposes. Thus they also want to exploit the "push button" (hiding the details from the user) approach to detecting errors. We provide a brief overview of some of the key methods that are comparable to RSDS and that use model checking. These either use existing model checkers or have developed new ones.

Model checking with STeP

The Stanford Temporal Prover (STeP) [MtSg96, BBC$^+$96] is a tool for verifying the temporal properties of reactive systems as well as real-time and hybrid systems. It combines model checking with deductive methods to provide verification support to a wide variety of systems such as parameterised programs and circuit design, but also systems with infinite data domains. Model checking is restricted to finite-state systems of a reasonable size (that is, not too large as to make model checking infeasible). Therefore, to model check an infinite-state system it must be

defined as a finite-system by limiting the number of instances, as well as constraining the size of the array and parameters. The deductive methods consist of verification rules that reduce the temporal properties to first-order conditions, and verification diagrams that visually assist with the organisation and management of the proofs. STeP also provides techniques for automatically generating invariants. These invariants are used to facilitate the verification process with an interactive theorem-prover.

Model checking with STATEMATE

I-Logix' Statemate MAGNUM [HLN$^+$90, Sta] is a graphical modelling and simulation tool for developing large and complex reactive systems. The modelling language includes the following standard engineering diagrams: data and control flow diagrams, structure diagrams, statecharts, truth tables and control law block diagrams. It informally ensures correctness of the specification's behaviour by exploring "what if" scenarios.

Statemate has recently been extended [Sta, BDK$^+$02] with model checking based techniques that are grouped into two parts: the ModelChecker$^{\text{TM}}$ component for robustness checks and the ModelCertifier$^{\text{TM}}$ component for certification. Robustness checks are performed at the early stages of development, ensuring that states are reachable or not, depending on the requirements, and that there are no non-deterministic situations. The ModelCertifier has primarily two roles. The first role is to assist the user with expressing properties by selecting and substituting a pattern from a pattern library. These patterns are invariants of typical requirements from the automotive, rail and aerospace application domains. The second role of the ModelCertifier is to check these properties by applying if necessary advance techniques such as abstraction and cone of influence. The model checking techniques use the VIS model checker as its core engine and automatically translate the Statemate specification into the VIS input language.

ProB: a model checker for the B method

ProB [LB03] is a model checker that has been recently developed to provide additional support for the verification of B specifications. B specifications are made up of B machines. When developing specifications with the B method, there are two types of proof checks that must be made: consistency checking that consists of formally showing that the operations of the machine preserve the invariants, and refinement checking that ensures the refinements produced are valid. ProB performs consistency checking of B machines automatically by exploring the state space of the B machines exhaustively or non-exhaustively to detect property violations. Counter-examples are displayed graphically.

The restriction of finite-state space still applies for ProB and thus the range of values for variables must be limited as well as that of sets. Also, ProB cannot model check temporal properties. However, it is a useful visual tool that assists debugging of B specifications.

MOCHA

MOCHA [AHM$^+$98, dAAG$^+$00] is an environment for interactively developing concurrent systems. It provides a language (Reactive Modules [AH99]) for defining system specifications in a modular way and supports hierarchical structuring and reasoning. A specification can contain synchronous modules, asynchronous modules, some modules specifying hardware and others software, and some modules can be time-critical. The simulator visualises the behaviour of the specification. MOCHA's main aim is to use the modular design structure to enhance model checking. It supports symbolic model checking (based on VIS) and enumerative search. The logic used to express the properties is Alternating Temporal Logic (ATL) [AHK02] that makes it possible to describe that a module can achieve a goal regardless of changes to its environment. Model checking is complemented with automated refinement checking for supporting hierarchical design and for decomposing verification tasks. Compositional and assume-guarantee rules are used for this.

2.8 Discussion

We have already explained that a plethora of model checkers and methods integrated with model checkers are available to use for this work. We have also briefly described some of those model checkers that could possibly be used for model checking RSDS specifications. From these, we have decided to use SMV. In this section, we present a justification for our choice by considering the model checkers presented in turn.

The model checkers for verifying the properties of real-time and hybrid systems are too specialised to be used for RSDS systems as RSDS only considers discrete control systems without any timing constraints. Furthermore, Kronos expresses properties using real-time temporal logic TCTL, HyTech does not apply model checking to temporal logic at all and UPPAAL only considers reachability properties and not a full temporal logic [BBF$^+$01]. COSPAN [AJKV97] also model checks properties of real-time systems. RSDS specifications do not currently define real-time constraints.

In order to apply the Model-Checking Kit to RSDS specifications, translations must be defined for each model checker included and its correctness shown. This is an enormous task and the benefits are not obvious as applying several model checkers does not necessarily mean that more errors will be found [ME03]. It is important to focus on capturing errors that are of interest. With the addition of extra steps in the verification process, more errors can be unintentionally introduced and it could be difficult to map back to the original model (for example, the RSDS specification).

Tools that model check programs cannot be used to model check RSDS specifications as there is a wide gap between the level of the representations. RSDS specifications describe the behaviour of systems in an abstract manner, omitting implementation detail, while programs provide specific implementation detail based on the particular programming language used.

Methods that embed existing model checkers usually do so by defining translations from their modelling notation to the input language of the chosen model checker. This process is hidden from the user. In order for these methods to support the verification of RSDS specifications via model checking, they must be able to model RSDS specifications according to the semantics of RSDS. The semantics of the modelling notations of methods differ from that of RSDS, for example, the semantics of the statechart notation of Statemate is different from that used in RSDS. Therefore, a translation from RSDS to the modelling language of the method must be defined if possible.

However, it is better to translate from RSDS directly to the input language of the model checker as the two sets of translations that would occur could lead to the loss of precision of the description of the RSDS specification. We believe that there are advantages in developing systems using the RSDS method, the main one being that RSDS provides a simple and abstract language for specifying systems with invariants and statemachines (where the statemachine notation is simpler with respect to the other statemachine/statechart notations) and automatically translates input lanugages for theorem-proving, model checking and execution (code). Therefore, there is some kind of platform independence at the specification level.

Some verification systems provide model checking as well as theorem proving. Since we want to complement the verification support of the B method for RSDS specifications, the most natural choice would be to use ProB for model checking. However, ProB has only been recently developed and is currently not as sophisticated as existing model checkers. Also, it does not model check temporal properties.

Another tool that combines model checking and thoerem proving is STeP, that has been used for verifying reactive systems. Its main contribution is a deductive tool that can verify infinite-state and parameterised systems. Its model checker uses symbolic model checking on finite-state systems only and uses explicit-state model checking on infinite-state systems but provides no guarantee for its termination. Since it allows for the description of parameterised systems, it seemed initially to be preferable for verifying RSDS/UML specifications. However, STeP model checks only a finite number of instances of a system that cannot be determined at run-time, meaning that the parameterisation can be compared to module definitions in SMV (i.e. no better than SMV). Also, it cannot model check all programs or properties accepted by STeP, for example, a property that has quantifiers with temporal operators in their scope cannot be model checked. The model checking tool is not robust or very powerful [MtSg96], some unexpected anomalies can occur. We prefer to use SMV as it has better tool support and it is actively maintained.

MOCHA is also a verification system for analysing models of reactive systems that seems suitable to be used for RSDS specifications. It is capable of modelling the environment in detail; for example, variables are defined according to their visibility by the environment or system, or both. RSDS specifications are presented at a more abstract level of detail, especially with respect to the environment which is modeled by underspecification. Therefore, we prefer closed model checking that views the complete system in isolation. Moreover, MOCHA provides automated refinement checking and we choose to stick to the RSDS semantic views rather than introduce refinement steps. MOCHA is based on the VIS model checker that has a Verilog (a hardware description language) front end. VIS is therefore more geared towards hardware systems and thus we prefer the input language of SMV.

SPIN is a good candidate for model checking RSDS specifications. Although it model checks LTL formulas as opposed to CTL (used to define RSDS properties), it is easy to use and provides many reduction techniques for handling the state space explosion problem. Nevertheless, we prefer SMV's input language rather than Promela as it is simpler for describing the behaviour of statemachines. Moreover, the counter-example produced by SPIN when a property is violated is often longer than that produced by SMV [BH99]. This is because of the former uses a depth-first algorithm, while the latter uses a breath-first algorithm. Long counter-examples can be difficult for users to read and can slow down the process of correcting the model.

Table 2.3 summarises the model checking tools discussed with respect to the issues concerning: the the logic used to describe the properties to be model checked; the techniques used for ensuring the efficiency of the tool; the input lanugage used for describing the model, and the level of abstraction with which the model is described.

2.9 Summary

In this chapter, we introduced the model checking problem that is classified in two ways. Global model checking checks whether a given formula is satisfied in a set of states. Local model checking checks whether a given state of a model satisfies a given formula. There are three common approaches for implementing model checking. The first two approaches are chosen depending on how the algorithm expresses the properties to be verified. If the properties are expressed using temporal logic, then the temporal logic approach is used for model checking. Alternatively, if the properties are expressed as automata, then the automata-theoretic approach is used for model checking. The third approach is the tableau approach used for implementing local model checking. In this work, we use a model checker that implements the temporal logic approach for global model checking. The properties we want to verify are expressed in CTL, but the model checker we use also checks LTL properties.

A major limitation inherent in all model checking approaches is the state space explosion problem. We have presented numerous ways that have been developed in order to minimise it: symbolic representation, on-the-fly verification, reduction and compositional reasoning. Particular emphasis was given to symbolic model checking as this is the technique that is used by the model checker that we have chosen to use. Symbolic model checking uses OBDDs, a symbolic method, for representing the state variables and transitions of a transition system. The desired properties can be model checked directly on the OBDDs rather than on the state space. The issue of concern is no longer the size of the state space but rather the size of the OBDDs.

Finally, we gave a brief overview of the model checking tools available and the recent implementation trends. A popular strategy is to integrate model checkers with verification systems and with development methods for debugging purposes or for further verification. We adopt this strategy for improving the verification support of RSDS specifications. The SMV model checker is integrated into the RSDS method by translating the RSDS specifications into the SMV input language. However, in order to present this work, we need to introduce the RSDS method and provide details of the syntax and semantics of an RSDS specification, which is explained in the next chapter.

Table 2.3: Summary of some model checking tools

	Logic Used	**Tool Efficiency**	**Input Language**	**Level of Abstraction**
SMV	CTL	Symbolic model checking	Modular, similar to finite state machines	Low level
VIS	CTL	Symbolic model checking with further techniques e.g. abstraction, equivalence checking, structural pruning	Verilog	Low level
SPIN	LTL	On-the-fly techniques	PROMELA that resembles C	Low level
COPSAN	LTL	On-the-fly techniques	Selective/resolution language	Low level
UPPAAL	Only reachability properties	On-the-fly techniques	Supports hierarchical structures like in UML statecharts and has friendly GUI	Low level
Kronos	TCTL	Symbolic model checking	Timed-automata	Low level
HyTech	No temporal logic. Only reachability properties	Symbolic model checking	Expressed both discrete and continuous change	Low level
Model-checking Kit	CTL, LTL	Various techniques	1-Safe Place/ Transition Petri nets	Low level
BLAST	Temporal-safety properties	Lazy abstraction	Proof-carrying code	High level
VeriSoft	No logic. Uses assertions	Depends on the amount of nondeterminism in the system being analyzed. They claim that they can reduce this problem with some ingenuity.	C or C++	High level
STeP	Temporal logic. Formulas have no temporal operators in the scope of binding operators.	Symbolic model checking	Parameterised finite -state systems. Not all properties accepted can be model checked.	Low level
I-logix Statemate MAGNUM	CTL	VIS used as core engine	STATEMATE	Low level
ProB	No temporal properties	Explicit state model checking - uses hashing function	B machines	Fomal method High-level model (less states)
MOCHA	CTL	Based on VIS	Reactive Modules	Low level

CHAPTER 3

Modelling of Reactive Systems with RSDS

RSDS [LCA02b] is a method for supporting the systematic development of reactive systems starting from the formulation of abstract requirements and resulting in code generation and verification. Its main objectives are to define a language that expresses the requirements in the simplest terms possible and to provide as much automated support as possible for analysis, design and code generation. The emphasis in development is shifted to the earlier stages where more attention can be paid to the correctness and safety properties of the system. The RSDS development process consists of the following phases, as illustrated in Figure 3.1:

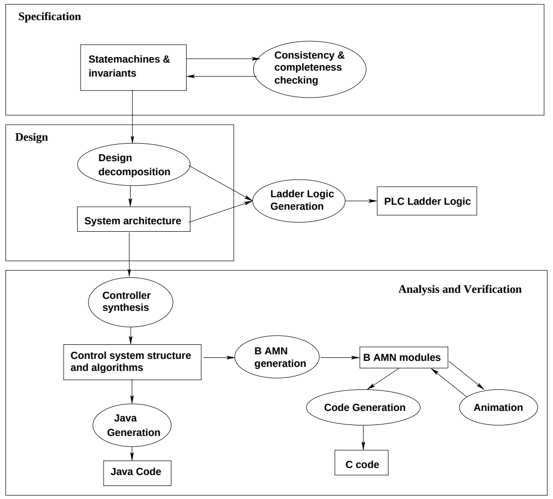


Figure 3.1: The RSDS development steps

1. **Requirements analysis and specification development:** An RSDS specification is composed of a set of invariants and a set of Structured Reactive Systems (SRS) statemachines. Invariants are used to concisely and precisely characterise the system behaviour. These are derived as a result of the requirements analysis. The SRS statemachine notation is a variation of classical statemachines, designed to take advantage of characteristic structures of reactive systems and to eliminate problems associated with traditional statechart semantics. An SRS statemachine is produced for each system component that describes its dynamic behaviour. The dynamic behaviour of the system shows how components move between states. The specification is validated by the tool automatically checking completeness and consistency of the invariants given.

2. **Design:** Data control flow diagrams (DCFD) [Wie03] are used to devise the system structure. Large systems can be decomposed into smaller subsystems, that can be independently developed and tested. A number of decomposition approaches have been defined [LAC00] that can be applied automatically to define the system architecture. The user, however, needs to decide which decomposition approach to apply.

3. **Analysis and Verification:** An explicit control algorithm is produced by using the invariants as input to the code synthesis process. The control algorithm is used to generate: ladder logic [Com93, LCA$^+$02e] for application to process control systems, B [LCA02b] for verifying the static properties and for animating the specification and Java [Jav, LFA02] for automatic code generation.

This chapter presents the details of what is described in each phase of the RSDS development process. Also, the syntax and semantics of RSDS specifications are defined. An example is used to illustrate how RSDS develops reactive systems.

3.1 Specification in RSDS

From a given set of requirements, the specification for an RSDS system is developed using invariants and SRS statemachines.

Definition 6 (RSDS system). An RSDS system Sys is a tuple $(Sens, Invs, Conts, Acts)$ where:

1. $Sens$ is a set of named statemachines representing the sensor components,
2. $Invs$ is a set of invariants,
3. $Conts$ is a set of named statemachines representing controllers, and
4. $Acts$ is a set of named statemachines representing actuator components.

We consider the notation used for the invariants and statemachines separately.

3.1.1 Invariants

The RSDS method uses invariants mainly for specifying reactive systems but they also play a key role in formal synthesis of programs [LBA99], formal design decomposition strategies [LAC00] and in critical applications of formal methods.

There are two main types of invariants used for specifying systems : *system constraints* that describe the behaviour of the system and *environmental constraints* that describe assumptions that are true about the environment. In RSDS, system constraints are used by the tool to synthesise the appropriate control code as these are required to be true of the system. Moreover, the developer needs to show formally that the invariants are properties of the system. Environmental assumptions can be used by the developer as hypotheses for verification of system constraints. When developing fault-tolerant systems, the environmental assumptions are regularly checked by a dedicated controller in the system architecture to ensure that the system is put in a safe state if these fail.

For safety critical systems, invariants are classified as being *critical* and *non-critical*. Critical invariants are necessary for safe behaviour of the equipment under control (EUC). The RSDS tool adds comments in the generated code in order for the user to trace the critical invariants. Both critical and non-critical invariants are interpreted formally in the same way.

Syntax

Definition 7 (Invariant). An invariant $I \in Invs$ of an RSDS system Sys, is any formula in the following CTL sub-language:

1. Atomic formulae:
   (a) propositional constants **true** and **false**;
   (b) action symbols α for each event of an element $Sm \in Sens \cup Conts \cup Acts$ where Sm is a statemachine;
   (c) equality formulae $sm = v$ where sm is an attribute symbol representing the state variable of an element $Sm \in Sens \cup Conts \cup Acts$, and v is a constant representing an element of the state set of Sm.
2. If ϕ and ψ are formulae, so are $\phi \Rightarrow \psi$, $\phi \vee \psi$, $\phi \wedge \psi$.
3. If ϕ is a formula, so are: AXϕ, EXϕ, AGϕ, EGϕ, AFϕ, EFϕ.

Semantics

The semantics for CTL are given in Section 2.3.3 and also apply to this CTL sub-language used for RSDS invariants. The meaning of each temporal operator used in this sub-language is:

If ϕ is a formula, then:

1. AXϕ, ϕ is true for the next state along all paths;
2. EXϕ, there exists a path where ϕ is true for the next state;

3. AGϕ, ϕ is true for all future states along all paths;
4. EGϕ, there exists a path where ϕ is true for all future states;
5. AFϕ, ϕ is true for some future state along all paths; and
6. EFϕ, there exists a path where ϕ is true for some future state

The RSDS tool currently processes formulae of the form $A \Rightarrow S$ where A is a disjunction of atomic formulae, and S is a conjunction of atomic formulae or S is of the form $Ops\ P$ where Ops is a non-empty sequence of temporal operators and P is a non-temporal formula not involving $\Rightarrow$.

Form of invariants

The invariants of RSDS specifications of discrete systems can be of the forms:

1. **Static (or single state) invariants:** $P \Rightarrow Q$ where P and Q consist only of constraints of current states of sensors and actuators in the system. No temporal operators or event names are used. This form of invariant is used to describe a system's normal and safety behaviour. An example of such an invariant is: $swstate = on \Rightarrow gvstate = open \wedge avstate = open$, which is part of the description of the behaviour of the gas burner system that is described in section 3.5. It means: if the state of the switch sensor is *on* then the states for the actuators gas valve and air valve are *open*.

2. **Operational (or action) invariants:** $\alpha\ \&\ P \Rightarrow AX(Q)$ where P and Q are state constraints on sensors and actuators, α is a sensor event, and AX is the temporal operator that denotes "next state on every path". These invariants are usually generated automatically by the RSDS tool, by converting the static invariants into this form. For example, the static invariant for the gas burner $swstate = on \Rightarrow gvstate = open \wedge avstate = open$ is converted to:
$$swstate = off \wedge swon \Rightarrow AX(gvstate = open \wedge avstate = open)$$
where *swon* is the sensor event for the switch sensor. Invariants of this form are used to synthesise the control algorithm. It is also possible to use this form of invariants to describe the system behaviour, however this is not recommended as it is very easy to miss out cases in the specification (see the Autopilot specification in Chapter 6).

3. **Temporal invariants:** $P \Rightarrow M(Q)$ where M is some temporal operator such as AF and AG (except for U), and P and Q are state constraints on sensors and actuators. These invariants are used to describe the following properties : reachability, safety and liveness, which should be checked to be true in the system specification. Each of these types of properties can be expressed in different ways. The form that occurs commonly in RSDS for each is:

   - Reachability: $AG(EF\phi)$ states that from any state, ϕ is reachable, where ϕ consists of constraints of current states of sensors or actuators (usually actuators). An example of a reachability property for the gas burner example is: $AG(EF(fdstate = present))$, which states that eventually a state is reached where the flame is present.

- Safety: $AG(P \Rightarrow Q)$ which is a also a static invariant. Violation of safety invariant can lead to a hazard. For example, in the gas burner system, a safety invariant is $AG(\mathit{gvstate} = \mathit{open} \Rightarrow \mathit{avstate} = \mathit{open})$, which states that the gas valve cannot be opened unless the air valve is. This reduces the chance of a large build up of gas that could lead to a potentially dangerous initial flame.
- Liveness: $AG(P \Rightarrow AF(Q))$ states that for any path, if P holds then eventually Q will hold. P and Q are constraints on sensor or actuator states. An example of a liveness property for the gas burner is: $AG(\mathit{swon} \Rightarrow EF(\mathit{swstate} = \mathit{on} \wedge \mathit{fdstate} = \mathit{present}))$, which states that if event swon occurs at any state, then eventually some state will be reached where the switch is on and the flame is present.

Two other property types that are often checked for reactive systems are: fairness properties and deadlock-freeness. These properties are not checked on RSDS specifications for the following reasons. Communication is unidirectional, therefore there is no possibility of deadlock. Also, the assumption that each event terminates and that the reaction to each event terminates means that the system is always live, so there won't be issues of fairness because of the simplicity of the architecture.

The RSDS tool carries out consistency checks on the invariants. These check whether invariants $A \Rightarrow B$ and $C \Rightarrow D$ are consistent by checking if B and D are contradictory when A and C are not. B and D are contradictory if for a particular component a, $B \Rightarrow a = v$ and $D \Rightarrow a = v'$ where v and v' are distinct.

3.1.2 SRS Statemachines

The SRS statemachine notation is a modular specification notation for reactive systems. It is based on finite state machines (FSMs) [HU79] with the additional capability of modularisation adopted from statecharts [PS91, HN96], which gains the advantage of structuring.

The main features of SRS statemachines are illustrated in Figure 3.2. There are three basic ingredients that are used to construct statemachines: *states*, that are represented by (rounded) rectangles; *transitions*, that are represented by labelled arrows, and *modules*, that are AND compositions of OR states separated by a dashed line. Each state has a name and can be of three types: AND, OR (nested), or basic. Basic states are states with no internal structure. OR states are states that have an enclosed statemachine, drawn as a rectangle, for example state $B2$ in Figure 3.2. AND states are pair or tuple of OR states in a parallel (AND) composition (each OR state in parallel is called a module). In Figure 3.2, the AND states are A and B.

A transition consists of two parts: a *trigger* and a sequence of *generated action(s)*. The trigger is a single event and a logical guard whose default expression is true. The guard is a logical condition on component states that is combined using the logical connectives $\wedge, \vee, \neg$. Negation of events and logical combinations of events are not allowed. Usually when specifying reactive systems in RSDS, sensor transitions have no guards, controller transitions have guards that can refer to both sensor and actuator states, while actuator transitions have guards that refer to other actuator states.

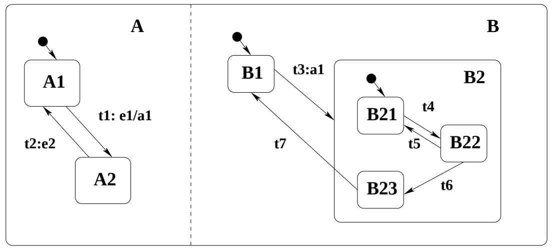


Figure 3.2: An example of SRS statemachines.

The sequence of generated actions correspond to events. Some transitions may not generate any events. Visually, transitions are labelled arrows connecting two states: the *source state* where the transition starts from, and the *target state* where the transition ends (the state where the arrow head is found). In Figure 3.2, transition $t1$ is triggered by event $e1$ and generates event $a1$ and its source state is $A1$ and its target state is $A2$ which is written as: $tr1 : A1 \rightarrow_{e1} A2$ (the generalised form is $tr : s[G] \rightarrow_{\alpha} t$ to indicate that **tr** is a transition with the source **s**, guard **G**, target **t** and event α). The initial state of a statemachine is depicted as the target of a virtual transition arrow with no source, for example, in Figure 3.2 the initial states for $A|B$ are $\{(A1, B1)\}$ and the initial state for the OR state $B2$ is $B21$.

There are two types of events: *external* events that are sensor events and *internal* events that are generated events. Transitions are classified as: *uncontrollable* transitions that are triggered by external events and are represented by dashed lines. *Controllable* transitions are triggered by internal events and are represented by solid lines. The RSDS system processes only one external event at a time, that is at each step.

A reactive system developed using RSDS, is modelled by an SRS statemachine that is an AND composition of a set of *modules*, where each module represents a system component: that is, one module for each sensor, controller, subcontroller and actuator. In Figure 3.3, a SRS statemachine describing a reactive system is illustrated that consists of a module for each sensor (SwitchA and SwitchB) , controller (Controller) and actuator (Actuator). SRS modules are organised in a strict hierarchy of receiving modules where a module transition may only generate events that trigger transitions in modules that are lower in the hierarchy. This strict hierarchy prevents cycles of receiving and sending. In Figure 3.3, the Controller receives events from the sensors (SwitchA and SwitchB), and the Actuator receives events from the Controller, i.e. the actuator cannot receive events directly from the sensors nor send events to any other module. The DCFD illustrates the flow of events between the system components (discussed in detail in section 3.3) which consequently visualises the hierarchy of receiving modules. Therefore, in the DCFD illustrated in Figure 3.4 it is clear which system components will be receiving events.

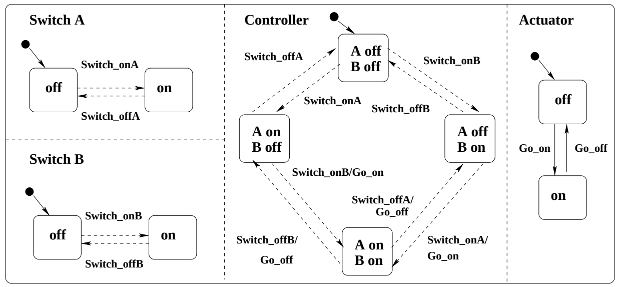


Figure 3.3: An example of a SRS statemachine for a reactive system.

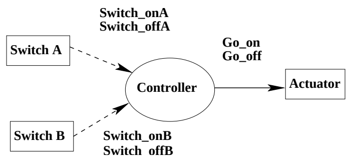


Figure 3.4: The DCFD for the reactive system in Figure 3.3.

Syntax

Definition 8 (SRS statemachine). A SRS statemachine for each system component in RSDS is defined as a tuple $A = (init_A, States_A, Trans_A, Events_A, source_A, target_A, event_A, guard_A, generations_A)$ where:

1. $init_A \in States_A$ is the initial state,
2. $States_A$ is the non-empty set of states of A,
3. $Trans_A$ is the set of transitions of A and $Events_A$ is the set of (input/sensor) events of A.
4. $source_A : Trans_A \rightarrow States_A$ gives the source states of transitions,
5. $target_A : Trans_A \rightarrow States_A$ gives the target states of transitions,
6. $event_A : Trans_A \rightarrow Events_A$ gives the triggering events of transitions,
7. $guard_A : Trans_A \rightarrow BooleanExpression$ gives the guard (which is optional) of a transition, and
8. $generations_A : Trans_A \rightarrow seq(Events_{B1} \cup Events_{B2} \cup ... \cup Events_{Bn})$ gives the generations (output events) of a transition, where $Events_{B1}, Events_{B2}, ..., Events_{Bn}$ refer to the events of statemachine $B1, B2, ..., Bn$ which are in parallel with A.

Remark 1. We assume that only one transition can occur from a source state in order to avoid non-determinism. Let

$tr1 \in Trans_A$,
$tr2 \in Trans_A$

then, the source of $tr1$ cannot be equal to the source of $tr2$, unless the events of $tr1$ and $tr2$ are different. Similarly for all transition of statemachine A.

Definition 9 (OR state). An OR state s of a statechart A has an enclosed statemachine $smach_A(s)$ and otherwise has the same properties as a statemachine state. $States_{smach_{A(S)}}$ are included in $States_A$ and similarly $Trans_{smach_{A(S)}}$ are included in $Trans_A$ and $Events_{smach_{A(S)}}$ in $Events_A$.

Consider the example in Figure 3.5. S2 is an OR state with nested states $\{S21, S22, S23\}$. When the transition with source $S1$ and event α occurs, the target state is the initial state in $S2$, that is, $S21$.

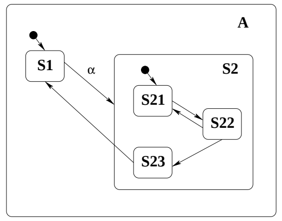


Figure 3.5: An example of an OR state S2.

Definition 10 (AND state). The AND state s is a parallel (AND) composition $A|B$ of two or more OR states. The states of $A|B$ are effectively pairs or tuples (a, b) of states a of A and b of B.

If $s = A|B$ then transitions $t : Trans_A$ in A may refer to the state of B via guard conditions of the form *in x* or *not(in x)* and logical combinations of these, where x is a state of B. The condition of t is given by $condition_A(t) : \mathcal{L}_B$, where $\mathcal{L}_B$ is the language of B. They may also possess generations of the form $e_1 \frown ... \frown e_n$, where the e_i are events of B and $\frown$ means sequential invocation. The generations of t are given by $generations_A(t) : seq(Events_B)$ $States_{A|B} = States_A \times States_B$ and similarly for $Trans_{A|B}$ and $Events_{A|B}$. The collection of all generated events of a statechart M is $Gen_M = \bigcup_{t:Trans_M ran(generations_M(t))}$. Figure 3.6 illustrates an example of AND composition $A|B$ with basic parallel states

$$(A1, B1), (A1, B2), (A1, B2), (A2, B1), (A2, B2), (A2, B3)$$

.

Definition 11 (Modules). A module M is an OR state containing only basic states.

Definition 12 (A module system). A *module system* is a system description S specified by the AND composition $M_1|...|M_m$ of all the modules contained in it, $modules(s) = \{M_1, ..., M_m\}$.

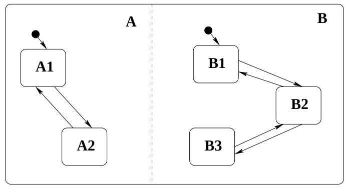


Figure 3.6: An example of AND composition $A|B$.

Definition 13 (Module receivers). Each module M in S has a set of receivers $receivers_S(M)$ of modules in S. It can only send events to $receivers_S(M)$ modules or refer to them in the conditions of its transitions. $receivers_S$ is acyclic: $M \notin receivers_S^*[\{M\}]$ where $receivers_S^*$ is the transitive closure of $receivers_S$ considered as a relation. For each module, M, the set of $receivers_S^*[\{M\}]$ is termed the *subsystem* S' defined by M. M is the *outer module* of S'.

Additional constraints may be placed on the form of $receivers_S$, for example, requiring that it is a tree: no two modules have a common receiver. This corresponds to the purely hierarchical structure of subsystems within a B development.

Definition 14 (Controllers set of receivers). In addition, the controller statemachine C has a set of receivers: $receivers_{Sys}(C) \subseteq Conts \cup Acts$; and, generations on its transitions:

$$generations_C : Trans_C \to seq(Events_{R_1} \cup ... \cup Events_{R_P})$$

where $receivers_{Sys}(C) = \{R_1, ..., R_P\}$. No event of any sensor component can occur in any generation.

The receivers hierarchy is illustrated by the DCFD diagram. For example, the DCFD diagram in Figure 3.4 depicts the receivers hierarchy for the SRS statemachines of the system illustrated in Figure 3.3. The actuator modules are the set of receivers for the controller, while the controller is the set of receivers for the sensor modules.

Definition 15 (Consistency). A statemachine Sm is *consistent* if there is at most one transition for each event from each state of the component:

$$\forall tr, tr' : Trans_{Sm}$$
$$event_{Sm}(tr) = event_{Sm}(tr') \wedge source_{Sm}(tr) = source_{Sm}(tr') \Rightarrow tr = tr'$$

Components in an RSDS system are required to be consistent.

Definition 16 (Completeness). Sm is *complete* if there is a transition for each event from each state of the component:

$$\forall \alpha : Events_{Sm}; s : States_{Sm} \cdot$$
$$\exists tr : Trans_{Sm} \cdot source_{Sm}(tr) = s \wedge event_{Sm}(tr) = \alpha$$

These properties of SRS statemachines are checked by the RSDS tool.

3.2 Semantics of RSDS Specifications

The semantic foundations of RSDS specifications are given in terms of Object Calculus theories. First we will describe what Object Calculus theories are and provide enough background to understand how we have used them. Then, we present the semantics of RSDS as theories.

3.2.1 Describing components as temporal theories

In [FM91, FM92, FSMS92], a formal framework is presented for describing component based systems as temporal theories that are combined using categorical constructs. This consists of defining temporal theories for components as modularisation units for specification, and structuring them by defining categorical morphisms [Pie91] for connecting the theories. A (static) configuration of a system is then characterised by a categorical diagram. It is possible to collapse the interconnected components into a theory through the colimit of the corresponding diagram, which represents the joint behaviour of the components.

Each theory consists of a signature and a collection of formulae (axioms) of the language generated from the signature. A signature is used to define the distinct vocabulary symbols that are used by the axioms to describe the object behaviour. It consists of at least three different parts: the *universe*, the *attribute structures* and the *action structures*. The universe provides information on the data context of the object, that is, its sets of types. The attribute structures are symbols that are used to record data and are state dependent. The action structures are symbols denoting atomic operations that define transformations on attributes and for interacting with other components. The axioms are usually defined in LTL, but any logic can be used. It is assumed that time begins at a certain point and the predicate **BEG** is true at that point in time. Encapsulation is logically enforced by an implicit axiom that is built into the logic, called the locality axiom, as part of each component definition.

Theories can be used to describe more complex systems by arranging the instances of the theories as nodes in a diagram, and using morphisms as edges between the nodes to express their interconnections. Morphisms are the structural mechanisms that define the relationship [1] that must exist between two object descriptions to ensure that one of them is considered as a component of another. The definition of morphism for these temporal theories, is that in order for two objects to interact, they must have a common sub-component in which they synchronise (pushout). This sub-component can be an action or an attribute. If the sub-component is an attribute, the interconnected objects must also share all the actions that update the shared attribute, to preserve encapsulation.

A theory that describes the joint behaviour of two interconnected objects is produced by taking the colimit of the corresponding diagram. The colimit is a generalised operation based on the pushout operation that is a "minimal" combination of the components that respects the morphisms (also called "amalgamation sum"). A pushout of an arbitrary category diagram, as described in [FM92], is the colimit of a diagram as expressed in Figure 3.7 that consists of another object $b \amalg_a c$ together with two morphisms ($h : b \to b \amalg_a c$ and $k : c \to b \amalg_a c$) such that the diagram

[1]"...this relationship consists of a translation between the languages of the description (a signature morphism) such that the theorems of one of them are translated to theorems of the second one." [FM91]

commutes[2] $h \circ f = k \circ g$ as in Figure 3.8; and, for every other commutative diagram (see Figure 3.9 there is a unique morphism (illustrated in Figure 3.10) $j : b \amalg_a c \rightarrow d$ such that $j \circ h = h$' and $j \circ k = k$'.

$$\begin{tikzcd}
\mathbf{b} & \\
\mathbf{a} \arrow[u, "\mathbf{f}"] \arrow[r, "\mathbf{g}"'] & \mathbf{c}
\end{tikzcd}$$

Figure 3.7: The form of an arbitrary categorical diagram that the colimit is applied to.

$$\begin{tikzcd}
\mathbf{b} \arrow[r, "\mathbf{h}"] & \mathbf{b} \amalg_{\mathbf{a}} \mathbf{c} \\
\mathbf{a} \arrow[u, "\mathbf{f}"] \arrow[r, "\mathbf{g}"'] & \mathbf{c} \arrow[u, "\mathbf{k}"']
\end{tikzcd}$$

Figure 3.8: The morphisms h and k are added to show how the diagram commutes.

$$\begin{tikzcd}
\mathbf{b} \arrow[r, "\mathbf{h'}"] & \mathbf{d} \\
\mathbf{a} \arrow[u, "\mathbf{f}"] \arrow[r, "\mathbf{g}"'] & \mathbf{c} \arrow[u, "\mathbf{k'}"']
\end{tikzcd}$$

Figure 3.9: Every other commutative diagram.

$$\begin{tikzcd}
 & & \mathbf{d} \\
\mathbf{b} \arrow[r, "\mathbf{h}"'] \arrow[urr, "\mathbf{h'}"] & \mathbf{b} \amalg_{\mathbf{a}} \mathbf{c} \arrow[ur, dashed] & \\
\mathbf{a} \arrow[u, "\mathbf{f}"] \arrow[r, "\mathbf{g}"'] & \mathbf{c} \arrow[u, "\mathbf{k}"] \arrow[uur, "\mathbf{k'}"'] &
\end{tikzcd}$$

Figure 3.10: There is a unique morphism j such that $j \circ h = h'$ and $j \circ k = k'$.

The fact that the colimit of these theories can be taken, means that these theories can be considered as specification modules as they are finitely cocomplete. A category is finitely cocomplete if the category has an initial object and every finite diagram has a pushout (i.e. then every finite diagram has a colimit).

Let us consider an example of a simple system whose behaviour consists of incrementing a counter variable by 1 each time a given variable is set. The system can be decomposed into two components: a cell component that sets its variable to a given value and a counter component that increments its local variable by 1. Each component is described using a temporal theory with first order CTL axioms. The cell component has an attribute v of sort *INT* and an action symbol *set* used in the axioms to describe how v is set to a given value (provided by the parameter). The counter component has an attribute symbol c of sort *INT* and an action symbol *inc* that is used in the axioms for describing how the counter is incremented. Note that *inc* has a parameter

[2]If the diagram commutes, then only one copy of a is obtained in $b \amalg_a c$ as b and c share a. An arbitrary categorical diagram is said to commute [LS97] if for each pair of categories (nodes) x, y in the diagram, all paths in the diagram from the x to y are interpreted as the same map.

that is never used. It is required as we want to synchronise *inc* with *set* when the components are combined, and synchronisation definitions must have the same type in this formalism. The theories for the cell and counter components are as follows.

CELL

Universe signature:
sorts: *INT*
operations: A number of operations are defined for *INT* which include 0, +, - etc.
Attribute symbols:
v : *INT*
Action symbols:
set(x:INT)
Axioms:
BEG $\Rightarrow \ v = 0$
$AG(\forall x \in INT.\ set(x) \ \Rightarrow \ AX\ (v = x))$
Also, some axioms characterising *INT*.

COUNTER

Universe signature:
sorts: *INT*
operations: A number of operations are defined for *INT* which include 0, +, - etc.
Attribute symbols:
c : *INT*
Action symbols:
inc(x:INT)
Axioms:
BEG $\Rightarrow \ c = 0$
$AG(\forall x, y \in INT.inc(x) \ \wedge c = y \Rightarrow \ AX\ (c \ = y + 1))$
Also, some axioms characterising *INT*.

So far, the two signatures and behaviours of the components are totally independent from each other. To define a more complex system with global properties, the independent components must be combined, that is the cell and counter components are combined to define the component *system* that describes the joint behaviour of the counter and the cell. The components are linked by morphisms drawn as edges in a diagram and are combined by taking the colimit of the diagram. Combining components requires them to synchronise via a common sub-component. In the cell and counter system the sub-component that it synchronises via is a with the morphisms: $f : act \mapsto set$ and $g : act \mapsto inc$ where *act* is the action symbol of component a. Figure 3.11 shows with a categorical diagram how component a is shared by the cell and counter components.

Figure 3.12 is the diagram for the cell and counter system where *system* is the colimit of cell and counter components via component a and is represented formally as *cell* $\amalg_a$ *counter.* The two morphisms introduced with *system* are: *h: cell* $\rightarrow$ *system* and *k: counter* $\rightarrow$ *system.*

The system theory, produced by taking the colimit, is as follows.

$$\begin{tikzcd}
\textbf{cell} & \\
\textbf{a} \arrow[u, "\textbf{f}"] \arrow[r, "\textbf{g}"'] & \textbf{counter}
\end{tikzcd}$$

Figure 3.11: The cell and counter components share attribute *act* of component *a*

$$\begin{tikzcd}
 & & \textbf{d} \\
\textbf{cell} \arrow[r, "\textbf{h}"'] \arrow[urr, "\textbf{h'}"] & \textbf{system} \arrow[ur, dashed, "\textbf{j}"] & \\
\textbf{a} \arrow[u, "\textbf{f}"] \arrow[r, "\textbf{g}"'] & \textbf{counter} \arrow[u, "\textbf{k}"] \arrow[uur, "\textbf{k'}"'] &
\end{tikzcd}$$

$\textbf{system} = \textbf{cell} \perp\!\!\!\perp_{\textbf{a}} \textbf{counter}$

Figure 3.12: *system* is the colimit of the cell and counter system diagram.

SYSTEM

Universe signature:
- *sorts:* *INT*
- *operations:* A number of operations are defined for *INT* which include 0, +, - etc.

Attribute symbols:
- $c : INT$
- $v : INT$

Action symbols:
- *act(x:INT)*

Axioms:
- $\textbf{BEG} \Rightarrow\ c = 0\ \wedge\ v = 0$
- $AG(\forall x \in INT.act(x)\ \Rightarrow\ AX\ (v = x))$
- $AG(\forall x, y \in INT.act(x)\ \wedge c = y \Rightarrow\ AX\ (c\ = y + 1))$
- Also, some axioms characterising *INT*.

It consists of two attributes (they are given the same name as in cell and counter, however, different names could be given) and a single action symbol *act* that synchronises *inc* and *set*. Therefore, the axioms in the cell and counter theories that characterise *inc* and *set*, are used in system to characterise *act*.

We use this Object Calculus for defining the two semantic views of SRS statemachines because its modularity can be used to describe elegantly the SRS modules. Also, the notion of locality is similar to that of RSDS: locality arises in SRS statemachines because if no transition of the statemachine occurs, then the state remains unchanged. In the Object Calculus, only actions[3] declared for a theory can change the values of its attributes. Therefore, if no action occurs the values of a theory's attributes remain unchanged.

3.2.2 Semantic views of SRS statemachines

There are two distinct RSDS semantic views for statemachines as illustrated in Figure 3.13:

[3]The occurance of an action in Object Calculus corresponds to a transition in SRS statemachines.

1. **Coarse-grain** semantics describe "computations" of the system as sequences of steps. Each step consists of a single sensor event and the system's reaction to that event as a finite number of actuator events, or it consists of no events. The steps lead from one stable state to the next. This corresponds to the machine descriptions at the specification level of the B method. At this level we can reason about global system behaviour like liveness properties.

2. **Fine-grain** semantics describe computations of a system at a finer level of granularity: the intermediate steps in a reaction cycle between stable states are explicitly represented. Each event that occurs simultaneously within a coarse-grain step, corresponds to a separate step in the fine-grain. Thus, a specific ordering of the actuator commands within the reaction cycle is defined. As in the coarse-grain view, a step can consist of no events. At this level, we can reason about constraints local to the reaction to an event. This level corresponds to the implementation level descriptions in the B method.

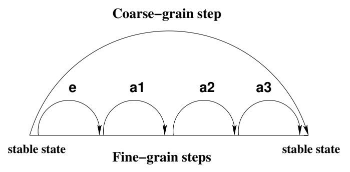


Figure 3.13: In RSDS statemachines, one coarse-grain step corresponds to a finite number of fine-grain steps.

Definition 17 (System states). A system state of an RSDS specification S is a tuple $(\mathbf{s}_1, ..., \mathbf{s}_p)$ of elements $\mathbf{s}_i \in \mathbf{States}_{Sm_i}$ where $\mathbf{Sm}_1, ..., \mathbf{Sm}_p$ are all the sensor, controller and actuator statemachines of the RSDS specification, which satisfies all the invariants of *S*.

System states are the only states in the coarse-grain semantic view and correspond to *stable states* which are the endpoint states in a reaction cycle in the fine-grain semantics.

The semantics for each view of the SRS statemachines is given in terms of temporal Object Calculus theories [FM91] using CTL. This formalism is appropriate for specifying the behaviour of RSDS statemachines because of its modularity where each component is specified by a separate theory which consequently allows for modular verification, that is a property local to a module can be verified independently from the rest of the system. The semantics that we present in this work is a variant of that in [LCA01] as we focus on the meaning of a controller that is derived automatically from its sensor components. The semantics presented in [LCA01] is more general, allowing for the user to define the controller.

3.2.3 Coarse-grain Semantics

Sensors

Each sensor component statemachine Sm in RSDS is considered as an object description defined by a theory Γ_{Sm} using CTL. The object signature consists of:

- an attribute sm ranging over $States_{Sm}$ (set of possible besic states for Sm),
- action symbols for each element of $Events_{Sm}$.
- action symbols for each element of $Trans_{Sm}$.

The object description consists of axioms for:

CG1 The initial state of Sm:

$$\textbf{BEG} \Rightarrow sm = init_{Sm}$$

where $init_{Sm}$ is the initial state of Sm.

CG2 That at most one event of Sm can occur in a step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of distinct events of Sm.

CG3 The state transition behaviour of Sm:

$$sm = s \wedge \alpha \Rightarrow tr$$
$$tr \Rightarrow \alpha$$
$$sm = s \wedge tr \Rightarrow AX(sm = t)$$

for each transition tr of Sm with source s, target t and trigger event α. The first axiom describes the conditions that enable a transition: a transition occurs only if its trigger event occurs and the system is in a particular state (source state). A transition cannot occur if its trigger event does not occur. The final axiom describes the effect of a transition that changes a state from the source to the target in the next step. We use the notation $tr : s \rightarrow_\alpha t$ to describe the state transition behaviour.

CG4 That at most one transition of Sm can occur in a step:

$$\neg(tr \wedge tr')$$

for each pair of distinct transitions of Sm. A step in a coarse-grain view consists of a single sensor event occurring and the system's response as a finite number of actuator commands.

CG5 That a transition can only occur if Sm is in its source state:

$$tr \Rightarrow sm = source_{Sm}(tr)$$

CG6 The locality notion [FM91] requires that there is no visible change to the value of the attribute sm when no transition is taken:

$$\neg tr_1 \wedge ... \wedge \neg tr_n \wedge sm = s \Rightarrow AX(sm = s)$$

for each $s \in States_{Sm}$, where the tr_i are all the transition action symbols of Sm.

THEORY Γ_{Sm}

Attribute symbols:
$sm : States_{Sm}$

Action symbols:
Action symbols for each element of $Events_{Sm}$.
Action symbols for each element of $Trans_{Sm}$.

Axioms:
[CG1]
[CG2]
[CG3]
[CG4]
[CG5]
[CG6]

Contollers

In RSDS, a controller statemachine and algorithm is derived automatically if the developer does not specify one explicitly. The controller is derived from the sensor components by taking the cartesian product of the sensor states from their statemachines. Some states might never be reached and we can use the invariants to identify these and remove them from the statemachine to keep the state space small. The environmental assumptions are particularly good at identifying these. Let us consider an example of a simple system with two sensors: *Sensor1* and *Sensor2*. Figure 3.14 shows how the controller statemachine is derived from the sensor components.

This controller consists of a set of states:

$$States_C = \{S1\_S3, S1\_S4, S1\_S5, S2\_S3, S2\_S4, S2\_S5\}$$

that is derived by taking the cartesian product of the sensor states, and a set of events:

$$Events_C = \{\alpha, \beta, \gamma\}$$

that corresponds to the union of all sensor events and a set of transitions:

$$Trans_C = \{T1\_S3, T1\_S4, T1\_S5, T2\_S1, T2\_S2, T3\_S1, T3\_S2\}$$

where the sensor transitions are combined with the cartesian product of the sensor states that are left unchanged when the transition is taken. Note that because of how the controller is derived, controller transitions do not explicitly express the sensor transition guards as they are implicitly represented by the controller state. For example, the controller state $S1_{S3}$ means that the sensor transition for *Sensor1* would have a guard $in(s3)$.

The meaning of the controller behaviour is obtained when the sensor theories are combined to form a theory that is the colimit of its categorical diagram. This theory is extended with an attribute and additional axioms to form the controller theory. For the example system in Figure 3.14, the categorical diagram in Figure 3.15 illustrates the amalgamation sum of the sensor theories where $\Gamma_{Sensor1}$ and $\Gamma_{Sensor2}$ are interpretations of theory Γ_{Sm} and Γ_{Sens} is the amalgamation

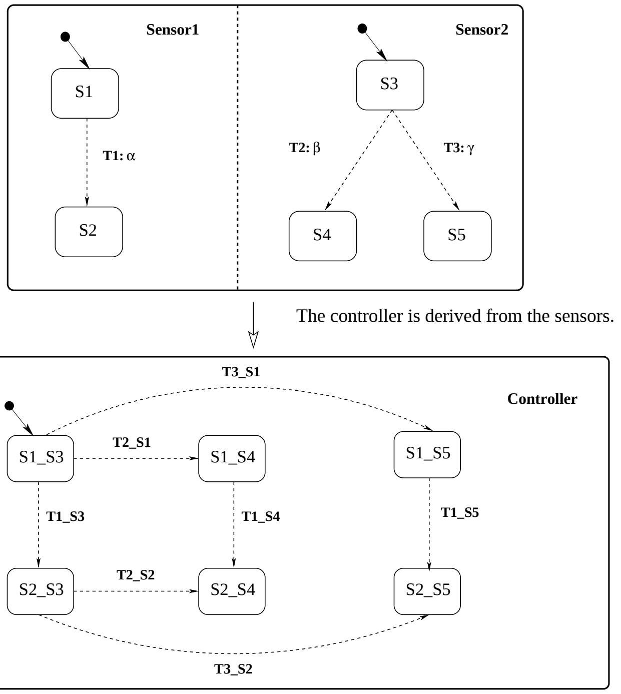


Figure 3.14: Example of how the sensors are flattened to derive a controller statemachine.

$$\begin{tikzcd} & \Gamma_{\text{Cont}} & \\ & \Gamma_{\text{Sens}} \arrow[u] & \\ \Gamma_{\text{Sensor1}} \arrow[ur] \arrow[uur] & & \Gamma_{\text{Sensor2}} \arrow[ul] \arrow[uul] \end{tikzcd}$$

Figure 3.15: The categorical diagram of the example system illustrated in Figure 3.14.

sum of the sensors, while Γ_{Cont} extends Γ_{Sens} with further axioms for formalising the guard of the transitions. Therefore, the controller is modelled in the theory Γ_{Cont}.

In general, a theory Γ_{Sens} is derived from the amalgamation sum of all the sensor theories that is Γ_{Sm_1}, ..., Γ_{Sm_n} where n is the number of sensors. This theory is extended further by Γ_{Cont} that contains the additional axiom:

CG7 We assume that only one sensor event can occur in each step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of events α of Sm and α' of Sm' where Sm and Sm' are distinct sensor components. We assume that there are no common events $Events_{Sm} \cap Events_{Sm'} = \emptyset$ between sensor components (extra-logical constraint). Similarly, the set of transitions of statemachines of different sensor components are disjoint.

THEORY Γ_{Cont}

Attribute symbols:
All attributes (without duplicates) of the sensor theories (interpretations of Γ_{Sm}).

Action symbols:
All action symbols (without duplicates) of the sensor theories.

Axioms:
All axioms from the (interpretations of) sensor theories and:
[CG7]

Actuators

Each actuator component statemachine Asm in RSDS is defined by a theory Γ_{Asm} that is similar to the theory Γ_{Sm} for sensors. The main difference lies with the definition of the state transition behaviour of actuators that considers guards. The following attribute is added to the object signature to represent the guard of a transition:

- a boolean attribute g_{tr} for each transition that will later be identified with the actual condition

With this attribute, the following axioms are defined:

CG8 The state transition behaviour of Asm:

$$sm = s \ \wedge \ \alpha \ \wedge \ g_{tr} \Rightarrow \ tr$$
$$tr \Rightarrow \alpha \ \wedge \ g_{tr}$$
$$sm = s \ \wedge \ tr \ \Rightarrow \ AX(sm = t)$$

for each transition tr of Sm with source s, target t and trigger event α.

THEORY Γ_{Asm}

Attribute symbols:
The same attributes as those of the sensor theory Γ_{Sm}.
g_{tr} : *Boolean*

Action symbols:
The same action symbols as those of the sensor theory Γ_{Sm}.

Axioms:
The same axioms as those of the sensor theory Γ_{Sm}, except that the axiom **[CG3]** is replaced by axiom **[CG8].**

We combine (by taking the colimit of the diagram as visualised in Figure 3.16) all of the actuator theories in a system in order to define an additional attribute and axioms for the guard of transitions. The resulting theory Γ_{Act} is extended to $\Gamma_{Actuators}$ with:

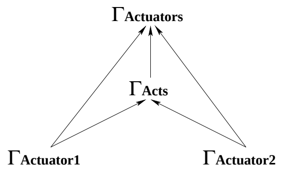


Figure 3.16: The colimit for the categorical diagram of a system with two actuators.

- a boolean attribute G for each transition

Moreover, similar additional axioms are defined:

CG9 The axiom

$$AG(g_{tr} \Leftrightarrow G_{tr})$$

is defined once the symbols in G are available. Since guard conditions of actuator transitions usually refer to the states/attributes of other actuators, and, the actuator theories cannot see the attributes of other actuator theories (e.g. in Figure 3.16, $\Gamma_{Actuator1}$ cannot see the attributes or action symbols of $\Gamma_{Actuator2}$), the guard conditions for the actuator theories must be given in the theory resulting from taking the colimit of the actuator theories (e.g. in Figure 3.16, theory $\Gamma_{Actuators}$). Therefore, the axiom states that if there is a guard for an actuator transition, then there must be a corresponding condition for that transition in the resulting theory Γ_{Act} and vice versa.

We assume that the set of transitions of statemachines of different actuator components are disjoint, that is for any pair of actuator components $Trans_{Sm} \cap Trans_{Sm'} = \emptyset$ (extra-logical constraint).

THEORY $\Gamma_{Actuators}$

Attribute symbols:
All attributes (without duplicates) from the interpretations of Γ_{ASm}.
G : *Boolean* for each transition.

Action symbols:
All action symbols from the interpretations of Γ_{ASm}.

Axioms:
All axioms from the interpretations of Γ_{ASm} and,
the axioms:
[CG9]

Subcontrollers

In RSDS, large systems can be divided into subsystems by applying decomposition approaches to improve its manageability. Each subsystem has at least a single subcontroller, which is ultimately part of the controller. The definition of subcontrollers is just a convenience and does not introduce any new behaviour. Formally, each subcontroller statemachine is an object description defined by a theory Γ_{Sm}. A subsystem is defined by the amalgamation sum of the subcontroller theories and actuator theories and also by copies (identical set of attributes and axioms) of the sensor theories that make up that subsystem. Figure 3.17 illustrates a DCFD of a system with a single controller, three sensors and three actuators. Figure 3.18 shows how the theories for the components in the system are combined in the categorical diagram. Note that guards of transitions of actuator components in a subsystem cannot be composed of states from other actuator components in other subsystems. Thus, the theories of actuators A1 and A2 are combined independently of the theory for actuator A3.

Complete system

The coarse-grain semantics Γ_{Sys} of an RSDS system specification *Sys* is an extension of Γ_{Sys0} (the amalgamation sum of the controller theory with the actuator theories and with the subsystem theories) together with the semantic interpretation $AG(I)$ for each invariant I and the global axioms.

These global axioms are additional axioms that are defined at the system level and consist of:

CG10 A "system locality" principle states that each actuator event occurs as a response to some sensor event:

$$\beta \Rightarrow \alpha_1 \vee ... \vee \alpha_n$$

for each β which is an event of some actuator component, and where the α_i are all events of the set of sensor components.

CG11 The axiom for the guard is defined:

$$tr_C \wedge G_A \Rightarrow tr_A$$

where G_A is the guard for an actuator transition, and, tr_C and tr_A are controller and actuator transitions respectively.

CG12 An axiom that asserts that each controller transition generates events received by subcontroller or actuator components:

$$AG(tr_C \;\Rightarrow\; \rho_1 \wedge \rho_2 \wedge ... \wedge \rho_p)$$

where tr_C is a controller transition and $\{\rho_1, ..., \rho_p\}$ are the subcontroller or actuator events that it generates.

THEORY Γ_{Sys}

Attribute symbols:
All attributes from the interpretations of the theories for the controller, subsystems and actuators.

Action symbols:
All action symbols from the interpretations of the theories for the controller, subsystems and actuators.

Axioms:
All axioms from the interpretations of the theories for the controller, subsystems and actuators, and, the axioms:
[**CG10**]
[**CG11**]
[**CG12**]

Figure 3.17 illustrates the DCFD for an example system specified using RSDS. For each component in the system, a theory is defined and these are added as labels next to the components. This system contains a single subsystem (grouped using a rectangle) which has a corresponding theory i.e. Γ_{SSys1}. The theory corresponding to the entire system is Γ_{Sys}. The categorical diagram for this system is illustrated in Figure 3.18. The sensors are combined to form the controller theory Γ_C. The subsystem theory Γ_{SSys1} is formed as a result of the amalgamation sum of subcontrollers, actuators and sensors (only those that apply to this subsystem). The symbol $\approx$ is introduced to mean "an exact copy" of the theory and is used to make copies of sensor theories to be included as part of various subsystems. At the system level Γ_{Sys}, any duplicates will be removed because of the definition of pushout. There is only one actuator Γ_{A3} which is not part of the subsystem and it is directly combined with the subsystem and controller to form the system theory.

3.2.4 Fine-grain Semantics

For an RSDS system $FSys$ consisting of sensors $S_1...S_p$, controllers $C_1...C_q$ and actuators $A_1...A_r$, the fine-grain semantics is given in terms of theories for each component, as for the coarse-grain. An additional attribute is introduced at the system level:

$$event\_queue : seq(EventToken)$$

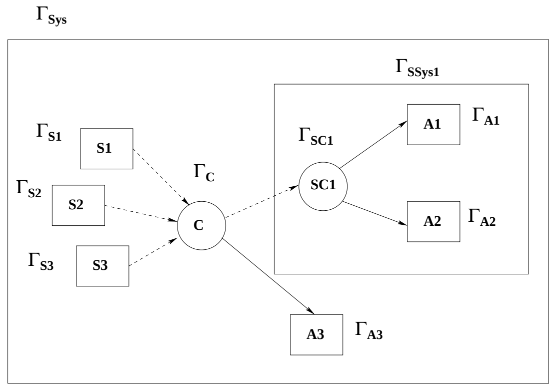


Figure 3.17: The DCFD for a system with theories for the corresponding components.

$$\begin{tikzcd}
 & & & \Gamma_{Sys} & & \\
\Gamma_{S3} \arrow[dr] & & & & & \\
 & \Gamma_{Sens} \arrow[r] & \Gamma_C \arrow[uur] & & \Gamma_{SSys1} \arrow[uul] & \Gamma_{A3} \arrow[uull] \\
\Gamma_{S2} \arrow[ur] & & & & & \\
 & \Gamma_{S1} \arrow[uu] \arrow[r, phantom, "\approx"] & \Gamma_{VS1} \arrow[uurr] & \Gamma_{SC1} \arrow[uur] & \Gamma_{Actuators} \arrow[uu] & \\
 & & & & \Gamma_{Acts} \arrow[u] & \\
 & & & \Gamma_{A1} \arrow[ur] & & \Gamma_{A2} \arrow[ul]
\end{tikzcd}$$

Figure 3.18: The categorical diagram for entire system illustrated in Figure 3.17.

that holds a sequence of tokens representing pending events. $EventToken$ is some set that is bijective to the set $Events_{FSys}$ of all events of the system:

$$Events_{FSys} = \bigcup_{i:1..p} Events_{S_i} \cup \bigcup_{i:1..q} Events_{C_i} \cup \bigcup_{i:1..r} Events_{A_i}$$

Let $name : Events_{FSys} \rightarrow EventToken$ be this semantic bijection. The *external events* of the $FSys$ are the sensor events received from the environment:

$$Ext_{FSys} = \bigcup_{i:1..p} Events_{S_i}$$

All other events of FSys are *internal*: $Int_{FSys} = Events_{FSys} - Ext_{FSys}$

The axioms of the components are the same as those for the coarse-grain, as are the system axioms, except that for the "system locality" principle, for the coverage of actuator events, is dropped. (Thus, these axioms are CG1-CG10, CG12 and CG13.) Instead, the following axioms on *event_queue* are defined.

FG1 Initially the event queue is empty:

$$\textbf{BEG} \Rightarrow event\_queue = []$$

FG2 An external event $\alpha \in Ext_{FSys}$ that triggers a transition t may only be responded to if the queue is empty:

$$t \Rightarrow event\_queue = []$$

FG3 When an external event α is received by the system and it triggers a sensor or controller transition t, its generations become the new event queue:

$$t \Rightarrow AX(event\_queue = generations(t))$$

The generated events must all be internal. If several transitions in different components are triggered by α then some interleaving of their generations becomes the new event queue. Normally, this case would arise when a transition is triggered in some S_i (and has no generations) as well as a transition in the outermost controller, so no non-determinism actually features. We assume for the purpose of consistency that each component has at most one transition for an event from each component state.

FG4 An internal event found in the queue can only be processed if it is at the head of the queue:

$$\beta \Rightarrow event\_queue \neq [] \wedge name(\beta) = head(event\_queue)$$

where β is the event at the head of the queue and *name* is defined above.

FG5 The head of the queue is removed when the corresponding event is processed and replaced with some interleaving ρ of the generations (set of internal events) of all the transitions triggered by the occurrences of β, the event at the head of the queue:

$$\beta \ \wedge \ q = event\_queue \Rightarrow AX(event\_queue = \rho \frown tail(q))$$

If there are no generations ($\rho \ = \ []$), then the event β at the head of the queue is simply removed. This models the "run to completion" semantics of statecharts in UML [UMLa].

THEORY Γ_{FSys}

Attribute symbols:
All attributes from the theory Γ_{Sys}.
$event\_queue : seq(EventToken)$

Action symbols:
All action symbols from the theory Γ_{Sys}.
However, events are divided into external and internal events.

Axioms:
All axioms from theory Γ_{Sys}, except for **[CG11]**,
and the axioms:
[FG1]
[FG2]
[FG3]
[FG4]
[FG5]

There are two properties that should be proved for a system, instead of being assumed as axioms:

FG6 If the event queue is not empty, the event at the head of the queue must eventually be processed:

$$event\_queue \neq [] \ \wedge \ name(e) = head(event\_queue) \Rightarrow AF(e)$$

for each event e of $FSys$.

FG7 Eventually all events in the event queue will be processed:

$$event\_queue \neq [] \Rightarrow AF(event\_queue = [])$$

A stable state is a configuration of $FSys$ where $event\_queue = []$. Therefore, a specification formula ϕ that is valid in the coarse-grain semantics of $FSys$ will be valid in the fine-grain semantics in the form:

$$event\_queue = [] \Rightarrow \phi^*$$

where ϕ^* is a relativisation of ϕ to the stable states of the fine-grain model. Relativisation means building a model of one theory inside another theory, such as taking $0, 1$ as a model of *Bool* inside the theory Nat with $1 - x$ as logical negation. Not all elements in the big theory appear in the small theory, i.e. for RSDS specifications, only stable states are used as the interpretation of the course-grain semantics inside the fine grain theory.

3.3 Design

DCFDs are used in the design phase for describing the system architecture. They are not part of the formal model of an RSDS specification. They are simple diagrammatical representations of the system structure. In reactive systems they depict the communication flows from the sensor components in the form of sensor device signals (sensor events) to the controller which then reacts to the sensor signals by issuing commands to the actuators. Square nodes are used for representing sensor and actuator components, and oval nodes for representing the controllers. The edges represent the flow of events between these elements. There are two types of communication flows: input event flows that are illustrated by *dashed* arrowed lines from sensor nodes to controller nodes; output command flows that are illustrated by *solid* arrowed lines from controller or subcontroller nodes to subcontroller or actuator nodes. The notation for the edges is adopted from the convention used for finite statemachines [San96]. The *uncontrollable* transitions in a finite statemachine correspond to the input event flows in the DCFDs, while the *controllable* transitions correspond to the output command flows. The simplest form of a DCFD for reactive systems consist of all the sensors and actuators and their connections to a single controller as in Figure 3.19.

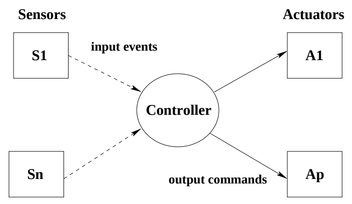


Figure 3.19: Basic DCFD for representing reactive system with RSDS

For most systems, it is necessary to decompose the controller further in order to reduce the complexity of the control algorithm. This results in a modularised specification of the control algorithm with manageable and analysable descriptions. There are a number of controller decomposition approaches that have been identified and invariants are used, for example as in [LAC00], to decide which approach to apply. The invariants are initially attached to the controller and when decomposed they are separated and those that apply to the subsystems are attached to the respective subcontrollers. Different decomposition approaches can be applied together to define complex systems. These are:

- **Hierarchical (vertical) decomposition of controllers:** input events e sent by the sensors $S1..Sn$ are handled first by an overseer controller C which manages certain interactions between components as illustrated in Figure 3.20. Derived events de are then forwarded from the overseer controller to subordinate controllers (subcontrollers) $C1..Cp$ responsible for handling the individual behaviour of subcomponents $A1..Am$.

  For a disjoint group $A_1..A_n$ of actuators, that is actuators that are not shared by controllers, if there are few invariants in the requirements relating the states of actuators A_i to those in A_j when $i \neq j$, then the hierarchical approach is applied. The coordinator (overseer) controller invokes the controllers for the actuators A_i and A_j in such a way that the invariants that link them are maintained.

  New invariants are generated that must be satisfied by the subcontrollers. These are obtained by selecting the invariants that refer only to the actuators that the subcontroller monitors. The invariants that refer to two or more actuators controlled by different subcontrollers remain as obligations on the overseer controller.

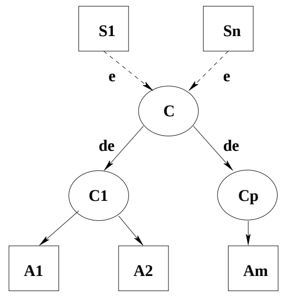


Figure 3.20: Hierarchical decomposition of controllers.

This design approach is suited to systems where some control aspects are managed at an aggregate level, separately from the control aspects that can be dealt with at an individual component level. For example, in [LAC00] a train control system is decomposed hierarchically with two subcontrollers, namely the Motor/Brake controller and the Door controller, that are responsible for issuing commands to their respective group of actuators, that is motor and brake, and door. Interaction between these subcontrollers is required as there is one safety invariant and one operational invariant that link the states of these groups. The coordinator controller is responsible for coordinating the interaction between the subcontrollers. Figure 3.21 gives the DCFD for the train control system.

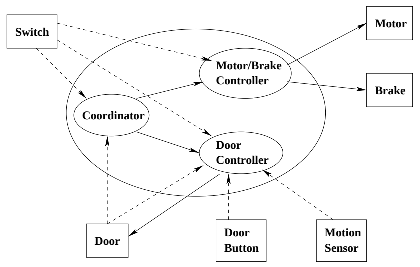


Figure 3.21: Hierarchical decomposition applied to the train control system.

The hierarchical decomposition is of particular interest for systems that include fault detection mechanisms [LCAK00], where separate controllers are used for detecting and responding to inputs that indicate failure of components. This approach is based on the physical decomposition of the actual system.

- **Horizontal decomposition:** input events e are copied to separate control algorithms (at least two) $C1..Cp$, which compute their reaction independently of each other as illustrated in Figure 3.22. The controllers compute their reactions in any order to ensure a coherent control algorithm and therefore the responses cannot be time-critical in relation to each other. This design approach is also based on the physical decomposition of a system.

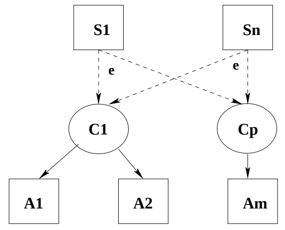


Figure 3.22: Horizontal decomposition of controllers.

Unlike in the hierarchical decomposition, for disjoint groups $A_1..A_n$ of actuators, if there are no invariants in the requirements relating the states of actuators in A_i to those in A_j when $i \neq j$, then the horizontal approach is applied. A subcontroller includes as obligations only the invariants that refer to the actuators in their subgroup.

- **Phase decomposition (decomposition by control mode):** this structuring is based on a conceptual division of the system. The DCFD for a system decomposed by control mode is similar to that of a system decomposed hierarchically as illustrated in Figure 3.20. A separate controller is specified to compute reactions for each mode or phase of the system. To determine whether this decomposition is appropriate: identify an invariant of the form $P_1 \vee ... \vee P_n$ of the system, where each P_i involves at least one sensor variable, the P_i are logically disjoint $P_i \Rightarrow \neg P_j$ for $i \neq j$, and the phases of the system are taken as those sets of states corresponding to the truth of each particular P_i.

  As in the hierarchical decomposition approach, new invariants are generated which must be satisfied by the subcontrollers. Contrarily, these refer to actuator states in that system phase only. It is possible for each phase to refer to the states of all actuators if they are involved in a specific system phase. The invariants that refer to the states of actuators involved in different system phases become obligations in the overseer controller. An example of such invariants is one that describes the order in which the phases occur.

- **Annealing:** repeated groups of actuator commands issued by the controller, are identified and packaged into a single module, so that the controller only needs to issue single commands to this module instead of sets of specific commands to particular actuators. This design approach aims to make the controller more manageable, so that if the precise set of actuators are changed, only these are changed and there is no need to traverse the specification to find which controllers issue these commands. Invariants are removed from the main controller and attached to this controller in order to improve verifiability in B.

- **Recognition of standard controllers:** In some systems, simple control mechanisms occur frequently and a list of ready-built controllers have been identified in [LAC00] to be used and adapted to particular sensors and actuators by renaming or chaining them together to

define more elaborate functions. To decide which standard controller is appropriate for the control problem, the invariants are examined for patterns that are similar to those defined in the standard controllers. Two such standard controllers have been identified:

1. The *AND controller* that takes inputs from two switches (sensors) and if both have been set to *on*, the actuator is set to *on*, otherwise the actuator is set to *off*.

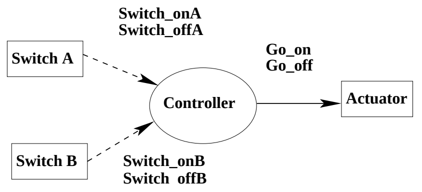


Figure 3.23: The DCFD for the AND controller system.

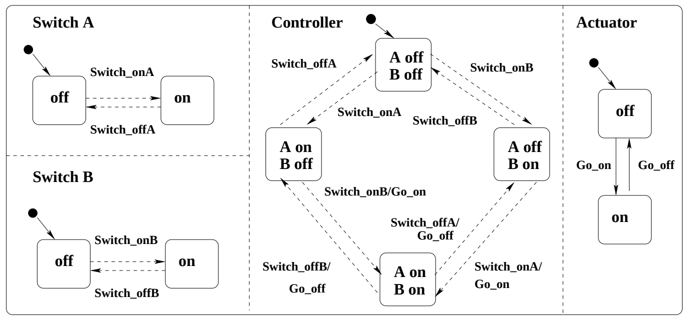


Figure 3.24: The SRS statemachine for the AND controller system.

2. The *priority controller* that sets actuator *A* to *on* when switch *A* is pressed and likewise actuator *B* is set to *on* when switch *B* is pressed. However, actuator *A* has priority over actuator *B* so that actuator *B* cannot be set to *on* until switch *A* is *on*.

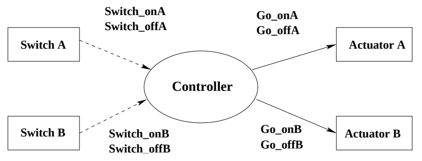


Figure 3.25: The DCFD for the priority controller system.

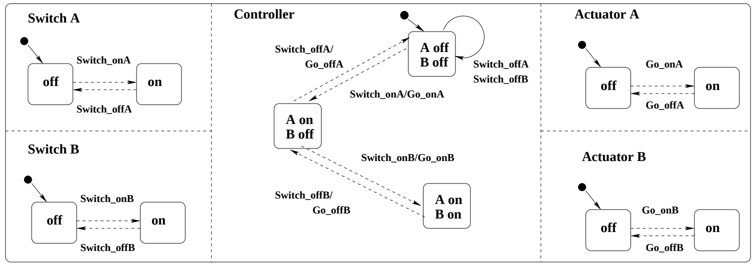


Figure 3.26: The SRS statemachine for the priority controller system.

3.4 Analysis and Verification

The analysis and verification phase consists of automatically synthesising a control algorithm from the invariants of an RSDS specification and translating it into B machines for verification. The RSDS tool also automatically generates Java code from the control algorithm. However, the Java code generated cannot be guaranteed as being correct (i.e. the desired properties hold) unless: the RSDS specification has been verified in B, and any corrections required are traced back and made to the RSDS specification before translating, and there is a proof of correctness of the translation to Java. At the moment, there is no proof of correctness of the translation to Java, therefore we cannot guarantee that the Java code generated accurately represents the RSDS specification even if the specification has been proved correct in B.

A control algorithm for a discrete system describes the system's reaction given in response to each sensor event received from the sensors. The controller computes the system's reaction which is then given in terms of commands to the actuators. Invariants, in particular operational invariants, are used to derive the control algorithm. However, the system behaviour is usually defined by static invariants. Therefore, the RSDS tool must be able to automatically convert the static invariants into operational invariants, which will then be used to derive the control algorithm. Safety invariants are used to (automatically) determine the exact order in which the actuator commands are issued, which is very important when specifying the fine-grain view of a system.

The following static invariant that is used to describe the behaviour of a reactive system has the typical form:

$$sstate = s1 \ \wedge \ G \Rightarrow astate = on$$

where *sstate* is some sensor state, *G* is a guard involving other sensor or actuator states and *astate* is some actuator state. In particular, this represents the obligation that the actuator state is set to *on* if the triggering sensor state is *s1* and the guard is true. This form of invariant can be automatically converted to an operational invariant of the form:

$$sstate \neq s1 \ \wedge \ set\_s1 \ \wedge \ G \Rightarrow AX(astate = on \ \wedge sstate{=}s1)$$

where *set_s1* is the sensor event that when triggered sets the sensor state to *s1* and if G is true, the actuator state is set to *on* in the next step. Also, any event that when triggered results in making *G* true while *sstate=s1*, sets the actuator state to *on.*

The control algorithm can be used to:

- **Synthesise abstract B specifications:** The B method is the formal approach used by RSDS to verify static invariants. In RSDS, the B specification is generated automatically from the control algorithm defined by operational invariants. A B machine is created for each component in the system and the DCFD defines the structuring of the machines where the arrowed lines represent the INCLUDES construct. In B, no cycles are allowed or Directed Acyclic Graph (DAG) with the INCLUDES structuring of machines as two machines are not allowed to change the state of a shared machine at the same time (single writer, multiple readers only). This restriction arises to enforce non-interference with compositionality and to preserve independent refinement of machines. In [BPR96, BB99, DBMM00, Lec02, Rod] ways of overcoming these limitations are discussed but unfortunately have not yet been implemented in the available B tools. When specifying reactive systems with RSDS, a DAG structure arises in the following cases:

  1. *Controllers sharing actuators:* In Figure 3.27, the LHS illustrates a common DCFD structure of a reactive system that when translated to B using the INCLUDES construct for defining the structure of the machines, a DAG[4] is produced because of the sharing of actuator *A2* by the two subcontrollers. Since this DAG structure occurs frequently in reactive systems, the authors in [LK98] describe a method for "tricking" the B Toolkit into accepting it. Dummy specifications are defined that do not INCLUDE any actuators and whose operations are defined as *skip.* The controller machine will INCLUDE these dummy specifications instead of the subcontroller machines. The original subcontroller machines remain and are refined. Once the implementations have been defined and code has been generated, the code generated for the dummy specifications is replaced by that generated for the subcontrollers. However, with this trick we would need to manually prove any invariants in the controller that refer to the states of both subcontrollers, thus limiting the benefits of using a theorem prover for development.

  2. *Controllers sharing sensors:* This problem arises with the sharing of sensors by separate controllers as defined by the horizontal decomposition approach. Ordinarily controllers are described in B as machines that include the B machine for the sensor component for updating its state, as shown on the LHS of Figure 3.28. To overcome this problem, a copy of the shared sensor machine is made and each controller has access to its own copy of the sensor, as shown on the RHS of Figure 3.28. The duplicated sensors must have identical state at all times.

  Generally, because of these sharing violations in B, the decomposition approaches are restricted accordingly to allowing the controllers to manage separate sets of actuators and only

[4]Also known as the shared violation problem caused by the inclusion of an actuator shared by two or more controllers.

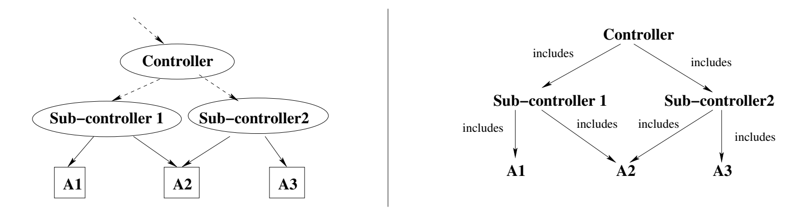


Figure 3.27: A DCFD structure where controllers share an actuator that produces a DAG structure in B.

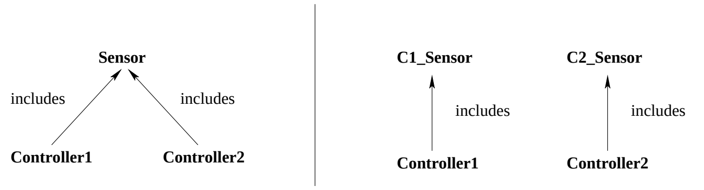


Figure 3.28: A DAG structure in B produced by controllers sharing a sensor and how to reconfigure it.

sharing few sensors.

Each sensor event corresponds to an operation of the controller in B. The body of the operation is determined from the operational invariants, for example, the invariant: $sstate \neq s1 \wedge set\_s1 \wedge G \Rightarrow AX\ astate = on$ is specified in B as follows:

```
set_s1 =
PRE sstate /= s1
  THEN
     sstate := s1 ||
     IF G
     THEN
        astate := on
     END
END
```

Details of the translation of RSDS to B are defined formally in [LBA99, LAC00].

- **Generate Java code:** A direct translation, without prior verification, exists from RSDS specifications to Java. Since B has limiting structuring constructs and Java allows for true object-oriented representation, other systems besides critical reactive systems can be developed in RSDS and have Java code generated.

For applying RSDS to process control systems, static invariants can be used to generate ladder logic code. Ladder logic is one of four standard languages for specifying the behaviour of PLC defined in [Com93]. Currently the RSDS tool does not support an automatic translation even though it has been theoretically defined in [LCA$^+$02e].

3.5 The Gas Burner System

The gas burner system [LFA02] is a simple reactive system used to demonstrate how an RSDS specification is defined in practice. Its components are shown in Figure 3.29. The behaviour of the gas burner is described as follows: when the switch is pressed, the controller should aim to move the system into a state where the air valve and the gas valve are open, the flame is ignited and the igniter is off. The system is shut down when the switch is turned off. While the switch is off, the air valve should be opened if the flame appears. All the sensors and actuators have binary states: for the sensors $sw : \{on, off\}$ represents the states of the switch and $fd : \{present, absent\}$ the states of the flame detector; for the actuators $av : \{open, closed\}$ represents the states of the air valve, $gv : \{open, closed\}$ are the states of the gas valve and $ig : \{on, off\}$ are the states of the igniter.

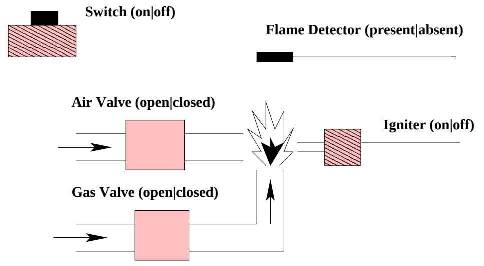


Figure 3.29: The gas burner elements

The DCFD diagram in Figure 3.30 is simple enough for this system and no further decomposition of the controller is required.

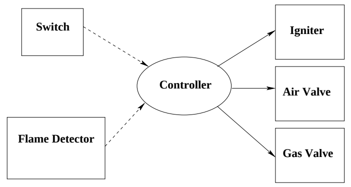


Figure 3.30: The DCFD for the gas burner

The control invariants that describe the operational behaviour are:

$$swstate = on \;\Rightarrow\; gvstate = open \wedge avstate = open \tag{3.1}$$

$$swstate = on \wedge fdstate = absent \;\Rightarrow\; istate = on \tag{3.2}$$

$$fdstate = present \;\Rightarrow\; avstate = open \wedge istate = off \tag{3.3}$$

$$swstate = off \;\Rightarrow\; gvstate = closed \wedge istate = off \tag{3.4}$$

$$swstate = off \wedge fdstate = absent \;\Rightarrow\; avstate = closed \tag{3.5}$$

Also, an operation constraint is that the igniter cannot be on unless the gas valve is open:

$$istate = on \Rightarrow gvstate = open \tag{3.6}$$

A required safety invariant is that the gas valve cannot be open unless the air valve is:

$$gvstate = open \Rightarrow avstate = open \tag{3.7}$$

These two properties need to be verified to ensure that the system is operating correctly and safely. Also, they describe the strict order in which the reactions must occur when both actuators are set. That is, the air valve must be switched on first, then followed by the gas valve and finally by the igniter. Also, the igniter is the first to be switched off and then the gas valve and finally the air valve.

The control invariants are converted into action invariants in order to synthesise the controller statemachine. The form of the action invariants is $P \Rightarrow AX(Q)$ where P and Q are state constraints on the sensors and actuators and AX is a CTL operator that means “in the next state”.

$$swon \wedge swstate = off \Rightarrow AX(gvstate = open \wedge avstate = open) \quad \text{from 3.1} \tag{3.8}$$

$$swon \wedge swstate = off \wedge fdstate = absent \Rightarrow AX(istate = on) \quad \text{from 3.2} \tag{3.9}$$

$$fdoff \wedge swstate = on \wedge fdstate = present \Rightarrow AX(istate = on) \quad \text{from 3.2} \tag{3.10}$$

$$fdon \wedge fdstate = absent \Rightarrow AX(avstate = open \wedge istate = off) \quad \text{from 3.3} \tag{3.11}$$

$$swoff \wedge swstate = on \Rightarrow AX(gvstate = closed \wedge istate = off) \quad \text{from 3.4} \tag{3.12}$$

$$swoff \wedge swstate = on \wedge fdstate = absent \Rightarrow AX(avstate = closed) \quad \text{from 3.5} \tag{3.13}$$

$$fdoff \wedge swstate = off \wedge fdstate = present \Rightarrow AX(avstate = closed) \quad \text{from 3.5} \tag{3.14}$$

Figure 3.31 illustrates the SRS statemachine *Sys* for the gas burner. The statemachine for the controller is derived automatically from the operational invariants and its state is the tuple

$$state\text{: } \{off\_absent,\ off\_present,\ on\_absent,\ on\_present\}$$

which is an amalgamation of the sensor states switch and flame detector.

In the fine-grain semantic view of the gas burner, the order of the events generated is very important as it ensures the safe operation of the system actuators. For example, if the gas valve is opened before the air valve and the igniter, then this can cause a hazard: the gas build up could lead to a large and potentially dangerous initial flame; or if the air valve and igniter fail to open, the leakage of gas produced is dangerous for humans to breathe.

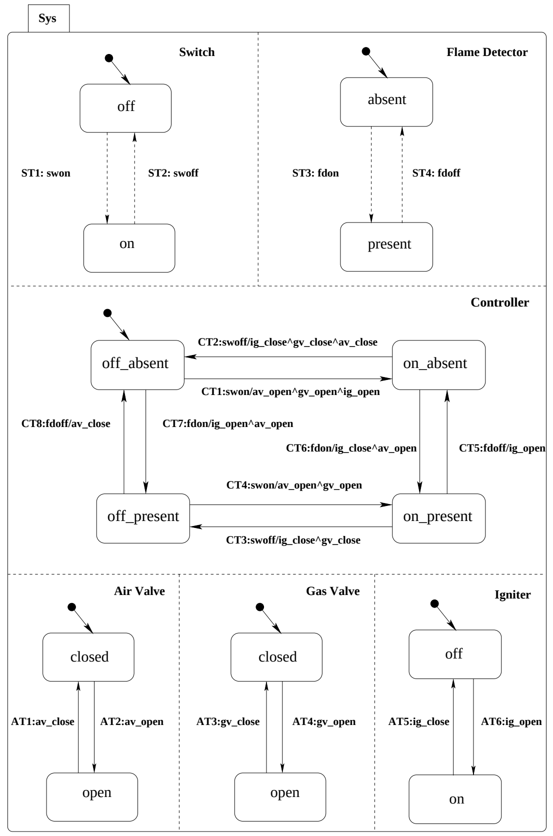


Figure 3.31: The SRS statemachine *Sys* for the gas burner example

Therefore, as the safety invariants impose, the air valve is opened first, then followed by the gas valve and the igniter is switched on. In [LBA99] rewriting techniques for invariants are proposed in order to obtain a number of ordering possibilities of actuator commands. For example, a safety invariant of the form $astate = x \Rightarrow sstate = s$ can be rewritten as $sstate \neq s \Rightarrow astate \neq x$. These rewriting techniques have been used to produce the ordering of generated events for the fine-grain representation of the gas burner.

Temporal invariants cannot be verified using B. For the gas burner, an example of a temporal invariant (liveness) that cannot be verified is:

$$swon \Rightarrow EF\ (swstate{=}\ on\ \wedge\ fdstate = present) \tag{3.15}$$

which states that if the switch is on, then eventually there is a path where a flame will be detected. We verify the weaker form of this liveness invariant (i.e. EF instead of AF) because the environment is not controlled and events occur nondeterministically. Besides the temporal operators that cannot be expressed in B, it is also not possible to express properties with events in the B language as events are represented as operations in B machines and these cannot appear in the description of an invariant.

3.6 Summary

The RSDS method aims to facilitate the modeling process of reactive systems and to improve communication between control engineers, who are interested in developing control systems and know the details of how they work, and software engineers that actually develop the software for these systems. These aims are achieved by the familiar graphical user interface (statemachines) provided by RSDS for defining the dynamic behaviour of the system as well as by the high level of automation provided by RSDS for generating as much of the source code as possible.

This chapter has presented the development phases of the RSDS method for the development of reactive systems. We briefly summarise the steps involved from a user's perspective and indicate which are automated and which require user interaction.

1. **Requirements analysis and specification development:**
   (a) The user formalises: (i) the safety, operation and liveness invariants that are associated with the main controller, (ii) the required reactions to events in the SRS modules. The RSDS tool automatically checks the completeness and consistency of the invariants.
   (b) The user produces a statemachine module for the sensor and actuator components. The invariants can be used by the RSDS tool to generate the statemachine for the controller. Moreover, templates of statemachines that are commonly used for control systems are provided by RSDS thus reducing the interaction required with the user. Any non-standard statemachine for a component must be provided by the user.
   (c) The RSDS tool produces automatically the DCFD for the entire control system.
2. **Design:**

   (a) A number of decomposition approaches are provided by the tool that can be automatically applied to the main controller to decompose it into subsystems. If a different decomposition approach is required, it must be defined by the user.
3. **Analysis and verification:**
   (a) A control algorithm is automatically produced by the RSDS tool from the invariants. The control algorithm is then used by the RSDS tool to automatically generate a B specification, ladder logic or Java code.
   (b) The control system is verified with the B method, but this usually relies on user interaction.

In addition, we defined formally the meaning of an RSDS specification and demonstrated how RSDS is used to develop reactive systems with the gas burner system.

CHAPTER 4

Applying Model Checking to the Coarse-grain View of RSDS

When developing reactive systems with RSDS, it is essential to capture errors as early as possible in the development process. Some syntax and consistency errors are captured by the RSDS tool when formulating the specification. The remaining errors are identified by the B method through verification of the system properties for both semantic views of RSDS specifications. However, temporal properties cannot be proven easily in B.

To overcome this problem, we have extended the RSDS method to verify temporal properties by using model checking. Model checking is suitable for incorporation into the collection of techniques in RSDS as it verifies automatically the temporal properties without the assistance of the user. Thus, its behaviour is appropriate with respect to the objectives of RSDS. The only requirement is that a system model is produced in the language of the model checker together with the properties to be verified expressed in the temporal logic used by the model checker. We have chosen to use SMV which has its own tool support. In order to reduce the user interaction with the model checker, we define a translation, that can be automated, from the RSDS specification into the input language of SMV. The translation that is defined must preserve the semantics of RSDS specifications.

RSDS statemachines have two semantic views: the coarse-grain and the fine-grain view. In this chapter we are concerned with applying model checking to the coarse-grain view. The coarse-grain semantic view describes the system computations as a sequence of steps. Each step consists of a transition triggered by a sensor (external) event and a finite sequence of transitions triggered by internal events.

In this chapter, we define the translation from the coarse-grain view of a RSDS specification to SMV by describing a number of translation rules and SMV schemas. Before we present the translation, we introduce the SMV model checker and give an account of the SMV language. Only a subset of the SMV language constructs are used in the translation, and we give an axiomatic semantics for this subset. This is required for the proof of correctness of the translation that certifies that the translation preserves the semantics of RSDS specifications. Also, we show how

to apply the decomposition approaches of RSDS to the SMV models in a natural way that helps to reduce the state space. We end this chapter with a comparative discussion of related translations.

4.1 The SMV Model Checker

SMV is a BDD-based model checker introduced by McMillan in [McM92b, McM93] that is used for automatically checking the validity of CTL formulas. The original SMV tool [McM92a] is maintained by Carnegie Mellon University (CMU). Another SMV tool is developed by Cadence Berkeley Laboratories [McM98a] that is also authored by McMillan. It provides some new features such as modular verification and compositional verification that are not backward compatible with the original version. Furthermore, a third tool exists called NuSMV [NuS, CCGR00] which is a re-implementation of the original SMV tool but with added features. It was developed as a joint project between ITC- IRST (Centre for scientific and technological research), CMU, the University of Genova and the University of Trento. A useful feature of NuSMV is that it can model check LTL properties as well as CTL. In this work, we mainly use NuSMV but we have also checked our models with the Cadence SMV model checker.

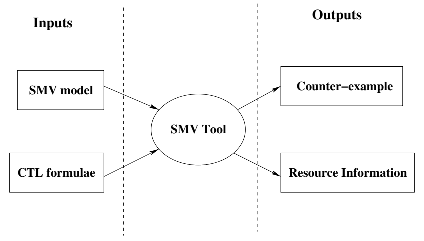


Figure 4.1: The SMV model checker.

The main constituents of the SMV tool are illustrated in Figure 4.1. It takes as input a system model described in the SMV language and CTL formulas to be verified. The SMV tool automatically checks the model against the CTL formulas and produces a counter-example in the form of a trace if the truth-value of the formula is false. The trace details the sequence of steps that lead to making the formula false. Each step in the trace only illustrates the values of the states that have changed. The counter-example is used for debugging a specification. A recent trend involves using the counter-examples produced by the SMV model checker for automatic test generation [HLSC01, GH99], thus emphasising the importance of counter-examples for debugging purposes. A further output of the SMV tool that can be requested is the resource information that provides the performance statistics of the model checker. This is useful for comparing the efficiency of different modelling approaches as well as for evaluating the efficiency of various techniques addressing the state space explosion problem.

4.1.1 The SMV language

The SMV specification language is used for describing the system model and the CTL formulas to be verified. It provides for modular system descriptions and for the definition of reusable components. The systems that it models can be deterministic or nondeterministic, synchronous or asynchronous. Since SMV was originally aimed at modelling finite transition systems, the language is limited to only boolean, scalar or fixed array data types. We only make use of a subset of the SMV language for defining RSDS specifications. This subset is explained as follows:

1. An SMV program is composed of *modules* with a mandatory module called *main*. Each module can declare variables and assign values to them. They can have parameters of any data type but also of module type. A module can be passed as a parameter to another module in SMV, in order for all of its variables and definitions to be visible (read-only access). The variables contained in the parameter modules can be used in the definition of a CTL formula under the SPEC clause, on the LHS of an assignment and on the RHS of a DEFINE clause.

2. A step in SMV corresponds to a a single tick of the global clock. However, for modules, the meaning of a step depends on how the modules are composed. SMV modules can be composed in two ways: *synchronously* or *asynchronously*. For modules that are composed synchronously, a step corresponds to a single step in each of the modules i.e. all assignments from all of the modules are executed in parallel. For asynchronous composition of modules, a step corresponds to a step by exactly one module i.e. the execution of modules is interleaved. We only use synchronous composition of modules for defining RSDS specifications.

3. Variable declarations in modules are defined under the VAR clause and can be of type module, boolean, scalar or fixed array or two-dimensional (2D) array. The variables that we define frequently are those of enumerated type, module instances and arrays.

4. DEFINE clauses characterise boolean variables without increasing the state space, whose values are given at all time points by the evaluation of the RHS of the clause. These are used for readability and re-usability purposes.

5. There are two types of assignments for a variable *v*: *init(v) := e* that sets its initial value to the evaluation of the expression *e* and *next(v) := e* sets the next value of a variable in terms of an expression *e* involving the current values of variables. The expressions *e* on the RHS include VAR and DEFINE variables of the module as well as those respective variables of parameter modules and no *next* expressions or other temporal operators, and at most one use of a *case* statement, at the outer level in *e*. Each assignment in each module is simultaneously executed at each time point for synchronously composed modules.

6. SPEC clauses introduce the definition of CTL formulas. In SMV, CTL formulas are expressed using a combination of the temporal operators (X, F, G, U) and path quantifiers (A, E), as well as $\&, |, ->, <->$ and ! for the respective propositional operators $\wedge, \vee, \rightarrow, \leftrightarrow$ and $\neg$. The propositional atoms in these formulas are simple boolean expressions composed of variables (local and those from other modules that are visible), the boolean constants true or false, symbolic and numerical constants, and comparative operators.

Other SMV constructs exist for describing asynchronous interleaving of modules, such as the keyword *process* that is used to declare a module and allows SMV at each step to nondeterministically decide whether to select it for execution. Also, a fairness constraint, defined with the *FAIRNESS* clause, restricts the search tree to execution paths along which an arbitrary CTL formula ϕ is true infinitely often. However, we do not use these when we model RSDS specifications in SMV.

4.1.2 Axiomatic semantics for a subset of SMV

In order to prove the correctness of our translation from an RSDS specification to an SMV model, we need to show that the meaning of the interpretation of an RSDS specification is preserved by the interpretation of the SMV model of the RSDS specification. Therefore, we require the formal definition of the meaning of the interpretation of the SMV model. McMillan in [McM92b] assigns a denotational semantics to the SMV specification language. In fact, the semantics is given in two parts: first for a subset of the language without the *process* keyword (similar to the subset we are interested in) and secondly for the language that includes the *process* keyword. However, since the semantics of RSDS is given using object calculus (see section 3.2), we need an axiomatic semantics for the subset of SMV. Such a semantics is given in [CH00]. We adapt this semantics and use the same formalism as that used for the RSDS semantics.

An SMV module M based on the form described in section 4.1.1 is given an axiomatic semantics in an object calculus theory Λ_M defined as follows:

1. Variable declarations can be of the following different types: either a boolean, an enumerated set, an array or 2D array where each element is a boolean, or a module instance. We consider each separately.

   (a) Each VAR declaration *var: Type;* in M, where *Type* is either a boolean or an enumerated set, gives an attribute *var* of sort *Type* in Λ_M.

   (b) Each VAR declaration *q: array x..y of Type;* in M, where *Type* is an enumerated set, gives an attribute *q* of $INDEX \rightarrow ITEM$, where the sort $INDEX = x..y$ being scalar and finite and the sort $ITEM$ is the representation of *Type* in Λ_M.

   (c) Each VAR declaration *qq: array x..y of array m..n of Type;* (2D array) in M, where *Type* is an enumerated set, gives an attribute *qq* of $INDEX \rightarrow INDEX \rightarrow ITEM$, where the sort $INDEX = x..y$ and $INDEX = m..n$ being scalar and finite and the sort $ITEM$ is the representation of *Type* in Λ_M.

   (d) Each module is defined in an object calculus theory. Therefore, if *C : CM*(M_1,...,M_n)*;* is a module variable declaration in M, then each axiom φ of Λ_{CM} is included in Λ_M, in the renamed form defined by replacing each local variable v or d of *CM* in φ by *C.v* and *C.d*, and by replacing each parameter variable $P_i.x$ of *CM* used in φ by $M_i.x$.

2. Each DEFINE clause *d := e;* in *M* gives a boolean attribute *d : {0,1}* in Λ_M with the axiom *AG((d=1)* $\equiv$ *e')* where *e'* is the translation of expression *e* into the language of Λ_M (see later).
3. (a) If *v* is a variable, each *init(v) := e;* clause is expressed by the axiom

$$BEG \;\Rightarrow\; v = e'$$

if *e* is not a *case* expression. As for DEFINE, *e'* is the translation of *e* into the language of Λ_M (see later). If *e* is a case expression,

$$\begin{array}{l} case \\ \qquad p_1 : e_1; \\ \qquad \vdots \\ \qquad p_n : e_n; \\ \qquad 1 : e_{n+1}; \\ esac; \end{array}$$

where 1 stands for *true*, then the initial state is defined by the axioms

$$\begin{array}{l} BEG \;\wedge\; p_1' \;\Rightarrow\; v = e_1' \\ BEG \;\wedge\; \neg\, p_1' \;\wedge\; p_2' \;\Rightarrow\; v = e_2' \\ \vdots \\ BEG \;\wedge\; \neg\, p_1' \;\wedge\; \ldots \;\wedge \neg\, p_n' \;\Rightarrow\; v = e_{n+1}' \end{array}$$

(b) If *v* is an array then *init(v[i]) := e;* is expressed by the axiom *BEG* ⇒ *v(i) = e'*.

(c) If *v* is a 2D array then *init(v[i][j]) := e;* is expressed by the axiom *BEG* ⇒ *v(i)(j) = e'*.
4. (a) Each *next(v) := e;* clause is expressed by the axiom

$$AG(v \;=\; x \;\Rightarrow\; AX(v \;=\; e'[x/v]))$$

if *e* is not a case expression. Similar axioms are defined for each value of *v*. If *e* is a case statement:

$$\begin{array}{l} case \\ \qquad p_1 : e_1; \\ \qquad \vdots \\ \qquad p_n : e_n; \\ \qquad 1 : e_{n+1}; \\ esac; \end{array}$$

then by the axioms

$$\begin{array}{ll} AG(v = x \;\Rightarrow & \\ & (p_1' \;\Rightarrow\; AX(v = e_1'\;[x/v])) \;\wedge \\ & (\neg\, p_1' \;\wedge\; p_2' \;\Rightarrow\; AX(v = e_2'\;[x/v])) \;\wedge \\ & \vdots \\ & (\neg\, p_1' \;\wedge\; \ldots \;\wedge \neg p_n' \;\Rightarrow\; AX(v = e_{n+1}'\;[x/v]))) \end{array}$$

Similar axioms are defined for each possible value of *v*.

(b) Similarly for each *next(v[i]) := e;* clause where *v* is an array and for each *next(v[i][j]) := e;* clause where *v* is a 2D array.

5. Each SPEC φ; clause is expressed by the axiom φ' where φ' is the translation of φ into the language of Λ_M (see later).

The translation of expressions and predicates of *M* into Λ_M is defined by:

1. The translation *d'* for a local DEFINE variable *d* is *d=1.*

   Similarly for *N.d* if $N : MOD(Q_1, ..., Q_k)$ is a module variable of *M* with *d* a DEFINE variable of *MOD.* If *P.x* is a parameter variable, it is treated as a DEFINE variable if

   - it does not appear on the RHS of a case clause, and
   - it occurs directly on its own as the LHS of a case clause or as a sub-formula within a logical expression $e\&f$, $e|f$, $!e$, $e \rightarrow f$ without being an argument of an equality expression $P.x$ = *val* or *val=P.x.*
2. The translation *v'* of local VAR variable *v*:

   - $(e)'$ is e
   - $(0)'$ is 0
   - $(1)'$ is 1

   Even though in SMV you can define variables of type BOOL, we choose to express booleans in terms of enumerated sets[1]. The reason for this is that we want to ensure that only one of the boolean values can be true at a time. The variable types declared in the coarse-grain translation are: enumerated type, boolean, module instance.
3. $(e \;=\; f)'$ is $e' \;=\; f'$
4. $(e \;>\; f)'$ is $e' \;>\; f'$
5. $(e \;<\; f)'$ is $e' \;<\; f'$
6. $(e \;>=\; f)'$ is $e' \;>=\; f'$
7. $(e \;<=\; f)'$ is $e' \;<=\; f'$
8. $(e \;\&\; f)'$ is $e' \;\wedge\; f'$
9. $(e \mid f)'$ is $e' \;\vee\; f'$
10. $(e \;->\; f)'$ is $e' \;\Rightarrow\; f'$
11. $(e \;<->\; f)'$ is $e' \;\Leftrightarrow\; f'$

[1]In section 4.3 that describes the translation from RSDS to SMV, all of the variables are defined in terms of enumerated sets i.e. variables representing current state (Rule 4) and current sensor event (Rule 7). This is also true for the fine-grain translation defined in Chapter 5.

12. $(!e)'$ is $\neg e'$

13. $(AGe)'$ is AGe' and similarly for all the other CTL temporal operators.

Λ_M has a single action symbol $step_M$ which may modify any of the variables local to M. There is the axiom

$$AG(step_M)$$

4.2 Translation Issues

A translation between notations involves finding a suitable model for the source notation in the target language that preserves the meaning of the source. It must guarantee a close relationship between the properties true in the source with those in the target. It is often the case that some modelling aspects in one notation are not available in another and a translation aims to find an acceptable solution [KG02]. We consider, in this section, the key issues that either make it difficult or easy to translate. These issues are identified at a low-level by comparing the languages associated with the translation and at a high-level by discussing the general modelling problems associated with the notations.

4.2.1 RSDS vs SMV

SMV was chosen for model checking the properties of RSDS specifications because of similarities between the RSDS and SMV notations that simplify the translation. Their underlying semantics are based on transition systems. We discuss the similarities and differences of these specification languages.

- Both SMV and RSDS describe the system in terms of modules. A module in RSDS is conceptually similar to a module in SMV as they both describe all the possible changes of a given state or variable. Moreover, the way that modules are composed in SMV and in SRS statemachines is similar, i.e. synchronous composition and AND composition respectively. They both execute assignments (or fire transitions) in parallel.

- The structuring mechanism in SMV promotes readability by breaking up the system into modules. These modules are instantiated by declaring a variable of type module. The modules can be flattened into a single module. RSDS provides several structuring techniques (decomposition techniques) that are based on conceptual or physical properties of the system. They were developed to reduce the complexity of the control algorithm and simplify proof in B. Since SMV code is generated automatically by the RSDS tool and the user does not need to know any SMV, the benefit of translating these techniques for improving readability is not required. However, it would be greatly beneficial if these techniques could be used to exploit the modularity of systems in order to reduce the state space by verifying properties locally, for example. In section 4.5, we present a natural way of applying these decomposition techniques to SMV models to obtain the benefits discussed.

- The locality principle is important in RSDS and should be respected in the SMV model generated. For each module, the locality principle states that if no transition occurs, then the states should remain the same. In SMV, the locality principle can be easily modelled with a *case* statement, where each transition is given as a possible case, and otherwise" nothing happens.

- RSDS provides a particular model template for implementing reactive systems while SMV does not. SMV was originally devised for hardware verification and therefore its language is very low level.

- The granularity of "next" in the coarse-grain view of RSDS is different to that in the fine-grain. The SMV model must implement correctly the granularity for each case.

- The behaviour of RSDS systems is deterministic as SRS statemachines do not allow any nondeterministic transitions. However, the environment generates sensor events nondeterministically i.e. any sensor event can occur and an RSDS specification should be able to respond accordingly. SMV can model both deterministic and nondeterministic behaviour. For example, the deterministic transitions of SRS statemachines can be modelled exactly in SMV by defining which transition should be taken at each step (there is no choice). Nondeterminism is modelled in SMV by "under-specification", i.e. by not specifying a particular behaviour, it considers all possible cases. For example, sensor events in RSDS specifications can be defined as variables of an enumerated set. By not defining in SMV which sensor event is true in which step, the SMV model considers all the possible sensor events in each step.

- There is no straightforward way of representing environmental assumptions in SMV. This is because environmental assumptions are not part of the system specification. We have to somehow restrict the environmental behaviour that the system deals with and in SMV this consists of restricting some of the branches of the system model produced.

- The state space of an SMV model is fixed. A state space is fixed if the universe does not change during a run. Likewise, RSDS specifications describe systems whose state space does not change during a run.

4.2.2 The state space explosion problem

Owing to the state space explosion problem, the state space of the SMV model produced for an RSDS specification must be small and finite. If a SMV model for a large reactive system has a large state space, the model checking tool will spend hours or days verifying it. The slow performance of the model checker makes verification impossible.

4.3 Coarse-grain Translation from RSDS to SMV

A single step in the coarse-grain grain semantics must be modelled in a single SMV step because we want to check that the system properties hold at each step. Therefore, if a sensor event occurs, the system's reaction to that event must be carried out in a single SMV step. Besides ensuring that the translation is semantics-preserving, our aim is to keep the state space of the SMV program produced as small as possible, by avoiding the unnecessary introduction of variables.

In this section, we describe the rules for translating from the coarse-grain semantic view of RSDS to SMV. These translation rules are used to derive the algorithm for the automatic generation of SMV modules. SMV schemas are used to describe the generalised form of the SMV program produced from the translation that can be substituted for particular system specifications. The translation schemas for the coarse-grain have been published in [AL01].

4.3.1 Translation of SRS statemachines

Translations are usually described as direct mappings between elements in the source to elements in the target language. However, in our translation we consider all the information provided to us by the statemachines and provide an optimised translation into SMV. This is because we know how a controller is synthesised, and the standard forms of sensors and actuators and we can use this prior knowledge of the system to make decisions concerning the translation. For example, for a reactive system whose controller is generated automatically, we know that the controller state and transitions are an AND-composition of the sensor states and transitions. Thus, the sensor state can be determined from the controller state and does not need to be explicitly modelled in SMV which consequently reduces the state space. Also, we decide not to model explicitly the generation of events because it is not possible to do so in a single SMV step. Instead, our approach is to synchronise in SMV the transitions and events within the modules, thus avoiding the introduction of new variables for each event generated that would increase the state space immensely.

First, we show the translation rules that are applied to all SRS modules to obtain the respective SMV definitions. Then, we show the details of how specific features of controller, sensor and actuator modules are translated into SMV modules.

In order to preserve the modular structure of SRS statemachines, each module for each component is translated into an SMV module. An SMV module is defined by declaring an SMV variable of type module in the main module. Since the controller module is composed from sensor modules (see semantics in section 3.2.3), the sensor modules do not need to be translated.The *main* module in SMV is used to compose the modules synchronously and to define with parameters which modules have read-only access to the states of other modules. The parameters are modules and are determined from the structure of the DCFD. Usually, the actuator modules require read-only access to the controller modules responsible for managing them in the structuring hierarchy. The translation rules below describe how the component modules are defined in the main module of SMV and what modules' variables they can access.

Rule 1: *For each controller component C in Sys*
```
VAR
 C : Controller;
```

Rule 2: *For each actuator component A in Sys*
```
VAR
 A : Actuator(C, OA);
```
where C is the controller responsible for managing actuator A and OA are any other actuator components that it will need to refer to (see translation rule 9).

The translation schema in Figure 4.2 can be expanded for any number of components in a reactive system. Actuator components have read-only access to the controller module because of the module parameter.

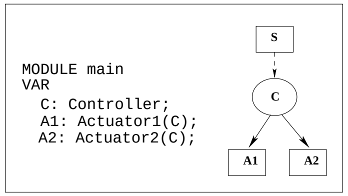

```
MODULE main
VAR
  C: Controller;
  A1: Actuator1(C);
  A2: Actuator2(C);
```

Figure 4.2: The translation schema for the main module and DCFD of a simple reactive system.

In Figure 4.3, the SMV code is given for the gas burner system illustrating the system components and how they are linked together. The actuator modules have read-only access to the controller module.

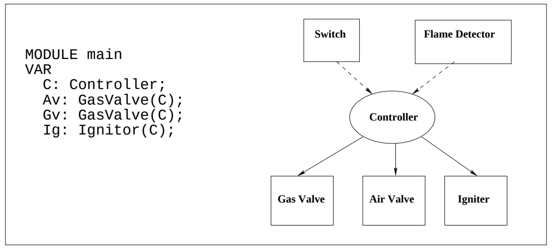

```
MODULE main
VAR
  C: Controller;
  Av: GasValve(C);
  Gv: GasValve(C);
  Ig: Ignitor(C);
```

Figure 4.3: The main module for the gas burner system.

All SRS statemachine modules share common elements: they all consist of states with transitions for moving between the states. The translation rules below describe how these common elements of statemachine modules are defined in SMV modules. These include: a declaration of a variable for representing the current state and assignments for defining its initial value and the values of states that are true in the next SMV step based on a transition occurring.

Rule 3: *For a state* **st** *in a module that ranges over the set of possible states* $(s_1, s_2, ..., s_m)$:
```
VAR
 st : {s1, s2, ...,sm};
```

Rule 4: *The initial state of st:*
```
ASSIGN
 init(st) := init_s;
```

Rule 5: *For each state st:*
```
ASSIGN
 next(st) :=
   case
    –For each transition tr with target state s2
    tr : s2;
    –For the default case
    1 : st;
   esac;
```
where tr is a controller or actuator transition

The definitions of transitions differ for various component types. The controller transition is defined in terms of its state and sensor event, while the transitions for the actuators are defined in terms of controller transitions. We consider each component type in turn.

Controller

For an automatically synthesised controller we can assume that the controller states and transitions are an amalgamation of states and transition of all sensor modules. Therefore, for a system with a set of sensors $S_1, \ldots, S_i$ and actuators $A_1, \ldots, A_j$, the controller statemachine C contains states $(x_1, \ldots, x_i)$ that are tuples of sensor states $x_1 \in States_{S1}, \ldots, x_i \in States_{S_i}$. The variable *stable_state* is defined (using translation Rule 3) in the SMV module for the controller state and ranges over a set of these possible system states that are described using the following notation: $x_1\_x_2\_...\_x_i$. The external event that occurs nondeterministically from the environment is modelled by the declaration of the variable *sensor_event* of enumerated type ranging over the set of all possible external events.

The transitions of C are the transitions of the AND composition of the sensors $S_1, \ldots, S_i$ with a set of generated events. Guards are not explicitly expressed in the SMV definition of the controller transitions because the guard is a condition on the sensor states and the controller state provides information on all of the sensor states. Therefore, to express a guard in the controller transition the appropriate controller state must be chosen.

Controller transitions $CT_1 \ldots CT_m$ are expressed using the DEFINE clause in SMV as a combination of sensor events, and the current stable state. The following translation rules show how events are defined in the controller module and the transitions in terms of these.

Rule 6: *For a sensor (external) event sensor_event that occurs:*
```
VAR
 sensor_event : {e1, e2, ...,en};
```
where sensor_event $\in$ *ExtEvents and e1, e2, ...,en are all possible events.*

Rule 7: *For each controller transition CT1,...,CTk :*
```
DEFINE
 CT1 := ex = e1 & stable_state = x1;
```
Similarly for all controller transitions

The translation schema for the controller is as follows.

```
MODULE Controller
VAR
 stable_state : {x1,...,xn};
 sensor_event : {e1,...,ek};

DEFINE
 CT1 : sensor_event = e1 & stable_state = x1;
 ...

ASSIGN
 init(stable_state) := c_init;
 next(stable_state) :=
  case
   CT1 : x2;
   ...
   1: stable_state;
  esac;
```

The controller transition CT_1 defined in the translation schema consists of $e_1 : x_1 \rightarrow x_2$. In SMV, the execution of this transition is expressed with a case statement in the *next* clause. Only one sensor state can change in each step. The generated events are not explicitly expressed in the controller module, instead the controller transition is synchronised (because of the synchronous composition of modules) with the sensor and actuator transition to ensure that the sensor and actuator states are updated in the same step.

Figure 4.4 illustrates the statemachine for the controller of the gas burner from which the following SMV code is generated.

```
MODULE Controller
VAR
 stable_state : {Off_Absent, On_Absent, Off_Present, On_Present };
 sensor_event : {swon, swoff, fdon, fdoff};

DEFINE
 CT1 : sensor_event = swon & stable_state = Off_Absent;
 CT2 : sensor_event = swoff & stable_state = On_Absent;
 CT3 : sensor_event = swoff & stable_state = On_Present;
 CT4 : sensor_event = swon & stable_state = Off_Present;
 CT5 : sensor_event = fdoff & stable_state = On_Present;
 CT6 : sensor_event = fdon & stable_state = On_Absent;
 CT7 : sensor_event = fdon & stable_state = Off_Absent;
```

```
  CT8 :  sensor_event = fdoff & stable_state = Off_Present;

ASSIGN
  init(stable_state) := Off_Absent;
  next(stable_state) :=
    case
      CT1 | CT5 :  On_Absent;
      CT2 | CT8 :  Off_Absent;
      CT3 | CT7 :  Off_Present;
      CT4 | CT6 :  On_Present;
      1:  stable_state;
    esac;
```

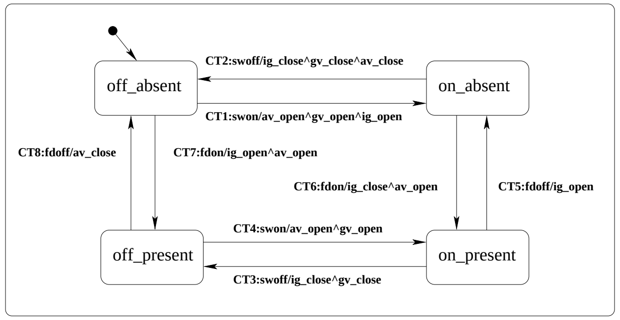


Figure 4.4: The SRS statemachine for the controller of the gas burner.

The controller state *stable_state* is a composition of the sensor states for the switch and flame detector components. The controller transitions are described in terms of sensor events and source states of the controller. By using the DEFINE clause to declare the transitions with boolean variables, the transitions can be accessed (read-only) by actuator modules that are parameterised with the controller module. This allows for the actuator modules to define their state change as a reaction to a controller transition.

Actuators

Actuators are subordinate to controllers and receive commands from them. Their transitions are synchronised with the controller transitions to ensure that actuator changes occur in a single SMV step. This synchronisation is expressed with the definition of actuator transitions in terms of controller transitions. The following translation rule shows how actuator transitions are translated.

Rule 8: *For each actuator transition AT1,...,ATy:*
`DEFINE`
`  AT1 := C.CT1 & G & a_state = a1;`
where C.CT1 refers to controller transition and G is the guard and a_state is the actuator state.

Moreover, actuator transitions can contain guard conditions that are boolean terms composed of other actuator states that can be evaluated. These are defined explicitly in SMV as a collection

of actuator states from any actuator in the system. For example, if G in translation rule 10 refers to the states of actuator *A1*, then it would have the form of *A1.a_state = s1* where s1 is some value of the state of actuator *A1*. Since the guard is given as actuator states, the actuator modules that house those states must be passed as a parameter in order for them to be referred to. This is illustrated in the translation schema for the controller where *A1* is an actuator whose state is referred to in the actuator transition *AT1* of *Actuator* module. If no guard is given, then it is assumed to be true.

The translation schema for an actuator is as follows.

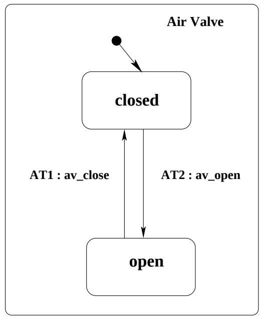
```
MODULE Actuator(C, A1)
VAR
  a_state :  {a1,...,ar};

DEFINE
  AT1 :  C.CT1 & A1.a_state = aa1 & a_state = a1;
  ...

ASSIGN
  init(a_state) := a_init;
  next(a_state) :=
    case
      AT1 :  a2;
      ...
      1:  a_state;
    esac;
```

The gas burner system consists of three actuators: an air valve, a gas valve and an igniter. Figure 4.5 illustrates the statemachine for the air valve and the SMV module generated by applying the translation rules. The current state for air valve is represented by the SMV variable *av* where its type corresponds to all the possible states of air valve (Rule 4). The variables under the *DEFINE* clause represent the actuator transitions (Rule 10). There are more definitions of transitions in the SMV code than in the statemachine because the transitions in the SMV code are given in terms of controller transitions in order to represent controller transitions generating actuator events (see next paragraph for a detailed example). The initial state of air valve is *closed*, which is defined in SMV with the *init* clause (Rule 5), and the possible state changes that occur as a result of a transition being taken, are defined using a case statement within the *next* clause (Rule 6). The locality of the module is maintained because of *1:av;* which means that if no other transition is true (i.e. *AT1..AT12*) then the current state remains the same. The SMV modules for the other actuators in the system are generated similarly.

There is a discrepancy between the transitions defined in the statemachines and those in the SMV code. This is because in the statemachine, the controller transitions generate the events that trigger transitions in the actuator statemachine, and in SMV there is no way of implementing this in a single step. Instead, the actuator state changes are defined in terms of the controller transitions and this synchronisation ensures that the reaction to a sensor event occurs in a single SMV step. For example, in the gas burner system the controller transitions *CT2* and *CT8* are the only transitions that generate the action *av_close*. In Figure 4.5, the actuator transition *AT1* in the statemachine is triggered when *av_close* is true (the event has been generated by a controller



```
MODULE AirValve(C)
VAR
 av :  {closed, open};

DEFINE
 AT1 :  C.CT1 & av = closed;
 AT2 :  C.CT1 & av = open;
 AT3 :  C.CT2 & av = closed;
 AT4 :  C.CT2 & av = open;
 AT5 :  C.CT4 & av = closed;
 AT6 :  C.CT4 & av = open;
 AT7 :  C.CT7 & av = closed;
 AT8 :  C.CT7 & av = open;
 AT9 :  C.CT6 & av = closed;
 AT10 :  C.CT6 & av = open;
 AT11 :  C.CT8 & av = closed;
 AT12 :  C.CT8 & av = open;

ASSIGN
 init(av) := closed;
 next(av) :=
  case
   AT1 | AT2:  open;
   AT3 | AT4:  closed;
   AT5 | AT6:  open;
   AT7 | AT8:  open;
   AT9 | AT10:  open;
   AT11 | AT12:  closed;
   1:  av;
  esac;
```

Figure 4.5: The statemachine and SMV code for the air valve component.

transition) and the state of the air valve actuator becomes *closed*. The actuator transition *AT1* in the statemachine corresponds to *AT4* and *AT12* in the SMV model. Additionally, *AT3* and *AT11* are the implicit self-transitions on the *closed* state that are explicitly defined in the SMV model. The self-transitions are only expressed explicitly for completeness as the model represents them with *1 : a_state* that means that "otherwise" (part of the case statement) the state stays the same.

4.3.2 Translation of the invariants

The system properties to be verified are described as CTL formulas under the SPEC clause in the main SMV module and are expressed in terms of the variables of the modules defined. The general form of the properties is: $AG(AG(Env) \rightarrow T)$ where Env is an environmental assumption that is used to impose a filter on the system to be model checked, and T is any invariant that requires verification.

Rule 9: *For each temporal invariant T:*
```
SPEC
 AG(AG(Env) -> T)
```
where Env is the environmental assumptions.

SMV verifies the following properties for the gas burner system that have no environmental assumptions:

```
SPEC
 AG(Gv.gv = open -> Av.av = open)

SPEC
 AG(Ig.ig = on -> Gv.gv = open)

SPEC
 AG(C.event = swon -> EF(C.state = On_Present))
```

There is a direct mapping between the action invariants and the SMV code generated. For example, the action invariant 3.1 for the gas burner system:

$$swon \wedge swstate = off \wedge fdstate = absent \Rightarrow AX(istate = on)$$

is expressed in the SMV code by the controller transitions CT1.

4.3.3 Interpreting the results of SMV

For each property described in an SMV model, the SMV model checker outputs: true if the property holds in the model; or false and a counter-example if the property does not hold in the model. It is clear from the construction of the SMV from RSDS, that any counter-example trace in RSDS can be translated into a counter-example in SMV. This means that if property Q holds in the SMV model, it also holds in the RSDS model.

The counter-example is a sequence of SMV steps of a path that leads to a violation of a property. All the values of variables and define variables (under the DEFINE clause) are given at each step. Therefore, the SMV steps are mapped to history states in SRS statemachines, where the *sensor_event* is mapped to the sensor event in the statemachines, the variables representing component states map to states in the statemachine modules, and the defined variables map to transitions in the corresponding statemachines. This mapping is very easy to automate as there is a direct correspondance between RSDS and SMV counter-examples (see Figure 4.6).

Lets consider the counter-example produced when checking the false property

$$AG(C.event = swon \rightarrow AF\ C.state = On\_Present)$$

of the gas burner system. It consists of three steps which are stuck in a loop. Figure 4.6 shows how the counter-example is mapped to elements in the SRS statemachines. The variables for each step in the counter-example (on the LHS of figure) correspond to the highlighted states and transitions in the snapshot of the statemachines (on the RHS of the figure). The property must be corrected to be

$$AG(C.event = swon \rightarrow EF\ C.state = On\_Present)$$

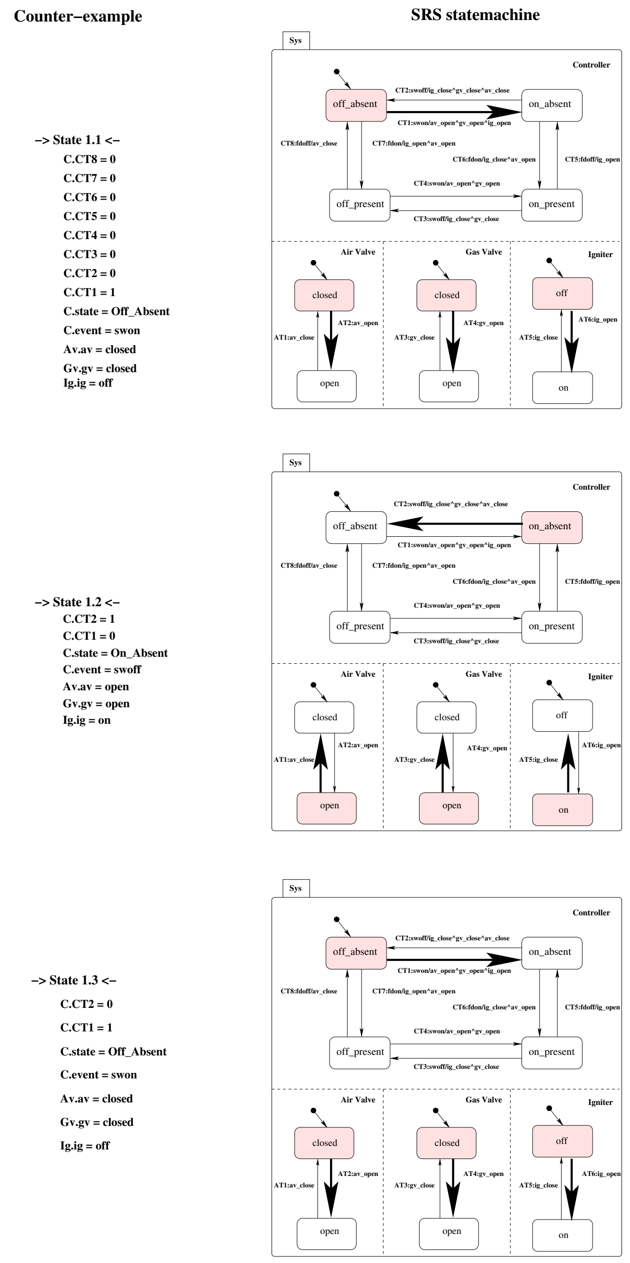


Figure 4.6: An example of how a counter-example maps to SRS statemachines.

4.4 Proof of Correctness for the Coarse-grain Translation

We need to show that the translation defined preserves the coarse-grain semantics of RSDS specifications. The semantics of SMV and RSDS are given in the same semantic domain (i.e. object calculus theories), so we can relate these semantics directly. We need to prove that everything that is true in the RSDS system theory Γ_{Sys}, is also true in the SMV interpretation of that theory $\Lambda_{M(Sys)}$ (soundness), and vice versa (completeness). In order to prove this, we must first prove the following lemma.

Lemma 1. The axioms of Γ_{Sys} hold in $\Lambda_{M(Sys)}$ under the interpretation ξ of an RSDS system theory Γ_{Sys} as a SMV interpretation of that theory in $\Lambda_{M(Sys)}$.

$$\begin{tikzcd}
\text{Sys} \arrow[r, "M"] \arrow[d, "\Gamma"'] & \text{M(Sys)} \arrow[d, "\Lambda"] \\
\Gamma_{\text{Sys}} \arrow[r, "\xi"'] & \Lambda_{\text{M(Sys)}}
\end{tikzcd}$$

$\xrightarrow{\Gamma}$	Derive the meaning of RSDS specification	Sys	RSDS specification of a system
$\xrightarrow{M}$	Apply SMV translation	Γ_{Sys}	Interpretation of the theory of RSDS specification
$\xrightarrow{\Lambda}$	Derive the meaning of SMV specification	M(Sys)	SMV specification of a RSDS specification
$\xrightarrow{\xi}$	Preserve the meaning	$\Lambda_{\text{M(Sys)}}$	Interpretation of the theory of the SMV specification

Figure 4.7: A sketch of the correctness proof for coarse-grain RSDS specifications.

Proof. We first define the interpretation of an RSDS system theory Γ_{Sys} into the theory $\Lambda_{M(Sys)}$, and then show that the axioms hold under this interpretation. The theory $\Lambda_{M(sys)}$ of the interpretation of an RSDS specification **Sys** is defined to be the theory Λ_{main} of the main module of **M(Sys)**. Figure 4.7 illustrates the relationship between these theories.

The interpretation of an RSDS system theory Γ_{Sys} into the theory $\Lambda_{M(Sys)}$ is defined by:

1. $\alpha : Events_s$ for a sensor S is interpreted by *C.sensor_event* $= \alpha_{tok}$ where α_{tok} is the name for α listed in the enumerated type of *sensor_event*.
2. $tr : Trans_s$ for a sensor S is interpreted as the disjunction $C.CT1 = 1 \vee \ldots \vee C.CTn = 1$ where the $CT1, \ldots, CTn$ represent all the controller transitions which depend on tr (in the RSDS model). We will use the shorthand given in SMV for $C.CT1$ to mean $C.CT1 = 1$, similarly for all boolean or DEFINE variables found on the RHS of assignments.
3. s, the sensor state variable of S, is the i-th projection $proj_i$(*C.stable_state*) of the controller state, if S is S_i.
4. A transition tr of the controller is interpreted by the predicate $C.CT_j = 1$ where CT_j is the corresponding DEFINE variable of the controller module.
5. The state attribute st of the controller is interpreted by *C.stable_state*.

6. A transition *tr* of an actuator is interpreted by the predicate $A.ATi = 1$ where ATi is the corresponding DEFINE variable of the actuator module.
7. Actuator events $\rho_1, ..., \rho_j$ are not explicitly represented in SMV. Each event ρ is interpreted by the disjunction of controller transitions $C.CT1 = \mathit{1} \vee ... \vee C.CTj = \mathit{1}$, where $CT1, ..., CTj$ represent all the controller transitions which generate ρ.
8. The state attribute *state* of an actuator is interpreted by $A.state$.
9. The guard g of an actuator transition is interpreted by the predicate $G = 1$, which is described in the definitions of actuator transitions $A.ATi$. It is composed of a conjunction of references to other actuator states, for example : $A2.state = g1$ & $A3.state = g2$, which refers to the state of actuator $A2$ and $A3$.

The axioms of Γ_{Sys} hold in $\Lambda_{M(Sys)}$ under this interpretation:

Controller

CG1 The axiom defining the initial state of Sm:

$$BEG \Rightarrow sm = init_{sm}$$

where Sm is a controller statemachine and sm is its current state, is interpreted in $\Lambda_{M(Sys)}$ as

$$BEG \Rightarrow C.stable\_state = init_{stable\_state}$$

which is immediately true. *C.stable_state* is a tuple of sensor states, which means that its initial state corresponds to the initial values of all the sensor modules $init_{S1}\_...\_init_{Sn}$, where $S1, .., Sn$ are the sensor modules of a system.

CG2 That at most one event of Sm can occur in a step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of distinct events of Sm. This axiom is interpreted as

$$\neg(C.sensor\_event = \alpha \ \wedge \ C.sensor\_event = \alpha')$$

which holds because of the definition of sensor event: *C.sensor_event :Type*, where Type is an enumerated set, and *C.sensor_event* can have only one value at a time.

CG3 The state transition behaviour of Sm:

$$sm = s \ \wedge \ \alpha \ \Rightarrow \ tr \tag{4.1}$$

$$tr \Rightarrow \alpha \tag{4.2}$$

$$sm = s \ \wedge \ tr \ \Rightarrow \ AX(sm = t) \tag{4.3}$$

for each sensor transition tr of Sm with source s, target t and trigger event α. Sensor transitions are interpreted as the disjunction of controller transitions. Therefore, axiom 4.1 is interpreted as

$$(C.stable\_state = s1 \vee ... \vee C.stable\_state = sn) \wedge C.sensor\_event = \alpha \ \Rightarrow \ C.CT1 \vee ... \vee C.CTn$$

were $s1, ..., sn$ are all the controller states where one of the elements in the tuple is the same i.e. corresponds to the sensor source of the sensor transition, and $C.CT1, ..., C.CTn$ refers to the controller transitions that are depended on tr. This axiom holds because of the definition of controller transitions:

$$C.CTj := stable\_state = si \ \& \ C.sensor\_event = \alpha$$

which are in terms of *C.stable_state* values and sensor events.

The axiom 4.2 is interpreted as

$$C.CT1 \vee ... \vee C.CTn \ \Rightarrow \ C.sensor\_event = \alpha$$

which holds, because of the definitions of the controller transitions.

The axiom 4.3 is interpreted as:

$$(C.stable\_state = s1 \vee ... \vee C.stable\_state = sn) \wedge (C.CT1 \vee .... \vee C.CTn) \Rightarrow$$
$$AX(C.stable\_state = t1 \vee ... \vee C.stable\_state = tn)$$

whereby $s1, ..., sn$ are the controller state values that contain the sensor state s in their tuple, and $t1, ..., tn$ are the controller state values that contain t in their tuple. This axiom follows from the definition of *next* for *C.stable_state*:

```
ASSIGN
  next(stable_state):=
  case
  CT1 :  t1;
   ...
  CTk :  tn;
  1:  stable_state;
  esac
```

which is interpreted as:

$$AG(C.stable\_state = s1 \ \Rightarrow \ (\neg C.CT1 \wedge ... \wedge \neg C.CTk \ \Rightarrow \ AX(C.stable\_state = t1)))$$

and similarly for all values of *C.stable_state*.

CG4 That at most one sensor transition of Sm can occur in the step:

$$\neg(tr \wedge tr')$$

for each pair of distinct transitions of Sm. This axiom is interpreted as:

$$\neg((C.CT1 \vee ... \vee C.CTn) \wedge (C.CTj \vee ... \vee C.CTk))$$

where $(C.CT1 \vee ... \vee C.CTn)$ are the controller transitions invoked by tr, and $(C.CTj \vee ... \vee C.CTk)$ are the controller transitions invoked by tr'. An assumption of RSDS specifications is that the events are unique, and each sensor transition is defined in terms of events. Therefore, this axiom is true because each sensor transition will invoke a different set of controller transitions depending on the sensor event.

CG5 That a sensor transition can only occur if Sm is in its source state:

$$tr \Rightarrow sm = source_{Sm}(tr)$$

This axiom is interpreted as:

$$C.CT1 \vee ... \vee C.CTn \Rightarrow C.stable\_state = s1 \vee ... \vee C.stable\_state = sn$$

where the source state is an element in the tuple of controller states $s1, ..., sn$. Since the controller state is an amalgamation of sensor states, a sensor transition can correspond to a number of controller transitions (maximum number = number of sensor modules) because although the event and the source state are the same, the states of the other sensors can be different. This axiom is true because of the definition of a controller transition:

$$C.CTi := C.stable\_state = s \ \& \ C.sensor\_event = \alpha;$$

which is in terms of the source state for *C.stable_stable*.

CG6 The locality notion [FM91] requires that there is no visible change to the value of the attribute sm when no transition is taken:

$$\neg tr_1 \ \wedge ... \wedge \ \neg tr_n \wedge sm = s \Rightarrow AX(sm = s)$$

for each $s \in States_{Sm}$, where the tr_i are all the transition action symbols of Sm.

This axiom is interpreted as:

$$\neg C.CT1 \wedge \neg C.CTk \wedge C.stable\_state = s \Rightarrow AX(C.stable\_state = s)$$

This is true because of the default case in the case statement within the *next* clause that defined the state changes to *C.stable_state*:

```
ASSIGN
  next(stable_state):=
  case
  CT1 :  s2;
   ...
  CTk :  sk;
  1:  stable_state;
  esac
```

which is interpreted as:

$$AG(C.stable\_state = s_0 \ \Rightarrow \ (\neg C.CT1 \wedge ... \wedge \neg C.CTk \ \Rightarrow \ AX(C.stable\_state = s_0)))$$

and similarly for all values of *C.stable_state*.

CG7 We assume that only one sensor event can occur in each step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of events α of Sm and α' of Sm' where Sm and Sm' are distinct sensor components. This axiom is interpreted as:

$$\neg(C.sensor\_event = \alpha \wedge C.sensor\_event = \alpha')$$

which holds because the type of *C.sensor_event* is an enumerated set, and by definition only one value of this set can be true in each SMV step (which corresponds to a coarse-grain step).

Actuators

CG1 The initial state of the actuator statemachine Sm:

$$\textbf{BEG} \ \Rightarrow \ sm = init_{Sm}$$

where $init_{Sm}$ is the initial state of Sm. This axiom is interpreted as

$$\textbf{BEG} \ \Rightarrow \ A.state = init_{Sm}$$

so is immediately true.

CG2 That at most one actuator event of Sm can occur in a step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of distinct events of Sm. Actuator events are not explicitly represented in SMV. We know from CG12 that actuator events are generated from controller transitions and are interpreted as a disjunction of controller transitions. Therefore, this axiom is interpreted as:

$$\neg((C.CT1 \vee ... \vee C.CTx) \wedge (C.CTy \vee ... \vee C.CTz))$$

where α is generated by $C.CT1 \vee ... \vee C.CTx$ and α' is generated by $C.CTy \vee ... \vee C.CTz$. This axiom is true because of CG4 of the controller theory.

CG4 That at most one actuator transition of Sm can occur in a step:

$$\neg(tr \wedge tr')$$

for each pair of distinct transitions of Sm. This axiom is interpreted as:

$$\neg(A.AT1 \wedge A.AT2)$$

for each pair of distinct transitions of actuator Sm. This axiom holds because actuator transitions are defined in terms of controller transitions, and we know that controller transitions are unique (i.e. only one of them occurs during a step) because of CG4 of the controller theory.

CG5 That a transition can only occur if Sm is in its source state:

$$tr \Rightarrow sm = source_{Sm}(tr)$$

This axiom is interpreted as:

$$A.ATi \Rightarrow A.state = a1$$

where $a1$ is the source state of transition $A.ATi$ and is immediately true because of the definition of an actuator transition:

$$A.ATi := A.state = a1 \ \&\ C.CTi \ \&\ G;$$

where G is the guarding condition whose default value is true.

CG6 The locality notion [FM91] requires that there is no visible change to the value of the attribute sm when no transition is taken:

$$\neg tr_1 \ \wedge ... \wedge \ \neg tr_n \wedge sm = s \Rightarrow AX(sm = s)$$

for each $s \in States_{Sm}$, where the tr_i are all the transition action symbols of Sm. This axiom is interpreted as:

$$\neg A.AT1 \wedge \neg A.AT2 \wedge ... \wedge \neg A.ATn \wedge A.state = s \Rightarrow AX(A.state = s)$$

This is true because of the default case in the case statement within the *next* clause that defined the state changes to $A.state$, the actuator state:

```
ASSIGN
  next(state):=
  case
  AT1 :  a2;
   ...
  ATn :  ak;
  1:  state;
  esac
```

which is interpreted as:

$$AG(A.state = a1 \ \Rightarrow \ (\neg A.AT1 \wedge ... \wedge \neg A.ATn \ \Rightarrow \ AX(A.state = a1)))$$

and similarly for all values of $A.state$.

CG8 The state transition behaviour of Asm:

$$sm = s \ \wedge \ \alpha \ \wedge \ g_{tr} \Rightarrow \ tr \tag{4.4}$$

$$tr \Rightarrow \alpha \ \wedge \ g_{tr} \tag{4.5}$$

$$sm = s \ \wedge \ tr \ \Rightarrow \ AX(sm = t) \tag{4.6}$$

for each transition tr of Sm with source s, target t and trigger event α. The first axiom is interpreted as:

$$A.state \ = s \wedge C.CTi \wedge G \ \Rightarrow \ A.ATi$$

where G is the guard. This axiom is immediately true because of the definition of the actuator transition which is given in terms of the controller transition, actuator state and guard.

Axiom 4.5 is interpreted as:

$$A.ATi \Rightarrow C.CTi \wedge G$$

which is immediately true because of the definition of the actuator transition.

The final axiom 4.6 is true because of the *next* assignment for the actuator state:

```
ASSIGN
  next(state):=
  case
  AT1 :  a2;
   ...
  ATn :  an;
  1:  state;
  esac
```

which is interpreted as:

$$\begin{aligned} &AG(A.state = a1 \ \Rightarrow \\ &\quad (A.AT1 \ \Rightarrow \ AX(A.state = a2)) \ \wedge \\ &\quad (\neg \ A.AT1 \ \wedge \ A.AT2 \ \Rightarrow \ AX(A.state = a3)) \ \wedge \\ &\quad \vdots \\ &\quad (\neg \ A.AT1 \ \wedge \ \ldots \ \wedge \neg A.ATn \ \Rightarrow \ AX(A.state = a1)))) \end{aligned}$$

showing that the cases when an actuator transition occurs brings about a change to the actuator state.

CG9 The axiom

$$AG(g_{tr} \Leftrightarrow G_{tr})$$

is defined once the symbols in G are available. This axiom is required in order to obtain the exact value of the guard (references to the state of other actuator modules) which are not visible from the actuator theory. Therefore, the interpretation of this axiom is:

$$AG(g_{tr} \Leftrightarrow A2.state = g)$$

if, for example, the guard is $A2.state = g$. Similarly for other guarding conditions.

Complete System

CG10 A "system locality" principle states that each actuator event occurs as a response to some sensor event:

$$\beta \;\Rightarrow\; \alpha_1 \vee ... \vee \alpha_n$$

for each β which is an event of some actuator component, and where the α_i are all events of the set of sensor components. In the SMV code, actuator events are represented by controller transitions. Therefore, the interpretation of this axiom is:

$$C.CT1 \wedge ... \wedge C.CTn \Rightarrow C.sensor\_event = \alpha_1 \vee ... \vee C.sensor\_event = \alpha_n$$

where $C.CT1 \wedge ... \wedge C.CTn$ are the controller transitions that generate β. This axiom is immediately true because of the definitions of the controller transitions (CG3 of the controller theory).

CG11 An axiom for the guard is defined:

$$tr_C \;\wedge\; G_A \Rightarrow\; tr_A$$

where G_A is the guard for an actuator transition, and tr_C and tr_A are controller and actuator transitions respectively. This axiom is interpreted as:

$$C.CTi \wedge G \Rightarrow A.ATi$$

which is immediately true because of the definition of $A.ATi$ that contains both $C.CTi$ and G.

CG12 The axiom that asserts that each controller transition generates events received by sub-controller or actuator components:

$$AG(tr_C \;\Rightarrow\; \rho_1 \wedge \rho_2 \wedge ... \wedge \rho_p)$$

where tr_C is a controller transition and $\{\rho_1, ..., \rho_p\}$ are the subcontroller or actuator events that it generates. This axiom is interpreted as:

$$AG(C.CTi \Rightarrow ((C.CT1 \vee ... \vee C.CTi \vee ... \vee C.CTx) \wedge ... \wedge (C.CTy \vee ... \vee C.CTi \vee ... \vee C.CTz)))$$

where each disjunction representing ρ_i (on the RHS of the implication) includes $C.CTi$, and therefore the axiom is is immediately true.

- All system invariants are valid in the RSDS controller module, so remain true in the SMV translation.

□

Theorem 1. *The translation is sound if we can show that: if* $\Gamma_{Sys} \vdash \varphi$ *then* $\Lambda_{M(Sys)} \vdash \xi(\varphi)$

Proof. The theorem follows because the rules of deduction are the same in the theories (quantifier free CTL in object calculus) and deduction steps [Eme90] in Γ_{Sys} are preserved by ξ. For example, for Modus Ponens:

If

$\Gamma_{Sys} \vdash \varphi$

and

$\Gamma_{Sys} \vdash (\varphi \implies \psi)$

then by induction on the length of the proof

$\Lambda_{M(Sys)} \vdash \xi(\varphi)$

and

$\Lambda_{M(Sys)} \vdash \xi(\varphi \implies \psi)$

which means

$\Lambda_{M(Sys)} \vdash \xi(\varphi) \implies \xi(\psi)$

so

$\Lambda_{M(Sys)} \vdash \xi(\psi)$

□

Therefore, with this theorem we proved the soundness of the translation. In order to show the completeness of the translation, we use the following lemma.

Lemma 2. For each trace h of the SMV model $M(Sys)$ there is a corresponding trace $r(h)$ in the RSDS model Sys which satisfies the same formulae with respect to the translation ξ.

Proof. If h consists of a sequence $h_1, ..., h_n$ of n Kripke nodes (where RSDS states and events are represented by variables), then $r(h)$ is defined as a sequence of n states and events as follows.

The settings of the sensors at state i of $r(h)$ are given by the value of $C.stable\_state$ in h_i. The settings of other components are given by their values in h_i. The sensor event which occurs to move from node i to node $i+1$ is given by $C.sensor\_event$ in h_i. The response actuator events and transitions are given by those disjunctions of conditions used in $M(Sys)$ to define their occurrence, which are true (i.e. their value is 1) in h_i.

These two traces have the property that:

$h \vdash_{\Lambda_{M(Sys)}} \xi(\varphi) \;\Leftrightarrow\; r(h) \vdash_{\Gamma_{Sys}} \varphi$ □

Hence, if h is an SMV counter-example to an RSDS property ψ, i.e.

$h \vdash_{\Lambda_{M(Sys)}} \neg\xi(\psi)$

the $r(h)$ is an RSDS counter-example to ψ:

$r(h) \vdash_{\Gamma_{Sys}} \neg(\psi)$

Let us consider a simple RSDS system with two sensors: $S1$ with states $\{s11, s12\}$, and $S2$ with states $\{s21, s22, s23\}$; a controller C and an actuator $A1$ with states $\{a1, a2, a3\}$. If

$$\varphi = C.stable\_state = s11_s21 \; \& \; C.sensor\_event = \alpha \;\Rightarrow\; AX(C.stable\_state = s11\_s22)$$

then a possible counter-example h produced for this system could be:

h_1 $C.stable\_state = s11\_s21$
$C.sensor\_event = \alpha$
$A.state = a1$
$C.CT1 = 1$ (where $CT1$ is defined in terms of this particular stable state and event)
$A.AT1 = 1$ (where $AT1$ is defined in terms of $CT1$)

h_2 $C.stable\_state = s12\_s21$
$C.sensor\_event = \beta$
$A.state = a2$
$C.CT2 = 1$ (where $CT1$ is defined in terms of this particular stable state and event)
$A.AT2 = 1$ (where $AT1$ is defined in terms of $CT1$)

where in h_2 $C.stable\_state = s12\_s21$ instead of $C.stable\_state = s11\_s22$. The corresponding counter-example $r(h)$ in the RSDS model Sys is:

$r(h_1)$ $S1 = s11$
$S2 = s21$
$stable\_state = s11\_s21$
$sensor\_event = \alpha$
$A1 = a1$
$tr_C = \alpha/\rho_1$ (where ρ_1 is the generated event of tr_C)
$tr_A = \rho_1$

$r(h_2)$ $S1 = s12$
$S2 = s21$
$stable\_state = s12\_s21$
$sensor\_event = \beta$
$A1 = a2$
$tr_C = \beta/\rho_2$ (where ρ_1 is the generated event of tr_C)
$tr_A = \rho_2$

This example illustrates the close correspondence between h and $r(h)$.

The same reasoning as in 6 also shows the completeness of the translation.

Theorem 2. *The translation is complete if we can show that: if $\Lambda_{M(Sys)} \vdash \xi(\varphi)$ then $\Gamma_{Sys} \vdash \varphi$.*

Proof. We prove the contra-positive: if $not(\Gamma_{Sys} \vdash \varphi)$ then $not(\Lambda_{M(Sys)} \vdash \xi\varphi)$ For every trace s of an RSDS system it is simple to construct a trace h of the corresponding SMV system for which $s = r(h)$. If

$$not(\Gamma_{Sys} \vdash \varphi)$$

there is a counter-example trace

$$s \vdash_{\Gamma_{Sys}} \neg\varphi$$

But then

$$h \vdash_{\Lambda_{M(Sys)}} \neg(\xi(\varphi))$$

and

$$not(\Lambda_{M(Sys)} \vdash \xi\varphi) \quad \text{which proves completeness.}$$ □

4.5 Applying Decomposition Techniques to SMV Models

The RSDS method uses a number of decomposition techniques for dividing large systems into manageable subsystems and simultaneously structuring them. Each subsystem consists of a subcontroller and a set of actuators. In order to apply the decomposition approaches to the SMV model generated, the subcontroller components must be translated into SMV as well.

Translating to SMV more than one level of controllers in a system at the coarse-grain level has no benefits, neither for improving readability or maintainability. This is because of the required synchronisation of the controller and actuator transitions for ensuring the coarse-grain step corresponds to a single SMV step. If an RSDS specification at the coarse-grain level, that has been decomposed and contains a second layer of controllers (known as subcontrollers), were to be translated into SMV, then the subcontroller transitions must synchronise with the controller transitions. The actuator transitions cannot synchronise with their subcontroller's transitions as this would require two SMV steps and that would not correspond to the coarse-grain semantics. Therefore, they would still have to synchronise with the controller transitions, making the subcontroller module redundant. Therefore, we translate into SMV a system with only one level of controllers.

In this section we propose some natural ways in which decomposition approaches can be applied to the SMV generated in order to reduce its state space. Where possible, subsystems are represented in separate SMV programs and model checked independently.

4.5.1 Hierarchical composition of controllers

The hierarchical decomposition approach divides the system into subsystems, as illustrated in Figure 4.8. Each subsystem consists of a subcontroller and a set of actuators, with an overseer controller known as the supervisor who is responsible for providing state information between its subcontrollers.

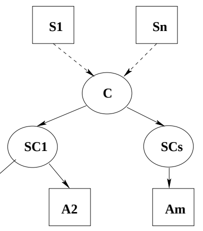


Figure 4.8: The DCFD for a hierarchical system.

For some large hierarchical systems (have more than 10^{20} states), model checking is infeasible. To make model checking feasible, we want to exploit the modular structure of RSDS specifications by verifying the subsystems independently, as separate SMV programs. However, there is usually a strong dependency of information between the subsystems of a hierarchical system. A supervisory controller is required to inform the subcontroller of one subsystem about the state of another. Moreover, global properties to be verified often refer to states of distinct subsystems. It is impossible to verify these global properties automatically if the hierarchical system is defined in separate SMV programs. Therefore, verifying subsystems independently is not straightforward.

To overcome this problem, we introduce a *virtual sensor* for each subsystem that depends on information from another. The virtual sensor acts as an interface between one subsystem and another and is treated as a sensor in one subsystem and as an actuator in the other. Its state is a boolean whose value is the truth or falsity of a condition composed of states of the subsystem that it depends on. The condition is determined by looking at the LHS of the invariants of the local subsystem, for sensor states of other subsystems that influence the change of a local actuator. The RSDS tool could determine this condition automatically.

Model checking this system consists of defining a separate SMV program for each subsystem that may contain virtual sensors. The global properties are split into local properties expressed in terms of the virtual sensor state and states local to the subsystem. These local properties are proven automatically using SMV. Some manual work is required for splitting the global property into local properties that are proven automatically, and also for combining them to prove the global property.

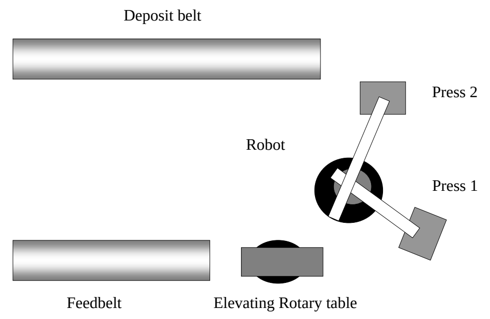


Figure 4.9: The main components of the fault-tolerant production cell.

As an example, let us consider the fault-tolerant production cell [Lot96] that is illustrated in Figure 4.9. This is a manufacturing system that processes metal pieces (blanks) that enter the system via a feedbelt. An elevating rotary table, found at the end of the feedbelt, receives a single blank at a time. There are two robot arms for moving a blank from the table (when the table is in the right position) to one of the presses, and for moving a blank after it is pressed from the press to the deposit belt. The system is fault-tolerant as it has two presses. If one of the presses fails, the system continues to operate. The RSDS specification for this system has been developed in [LCAK00, LCA02b]. Two main decomposition approaches have been applied: the horizontal and the hierarchical. Figure 4.10 highlights these decomposition approaches in the DCFD for the complete system. We will consider only part of this system, namely the "Introducing blank"

subsystem that manages the control of the feedbelt and table.

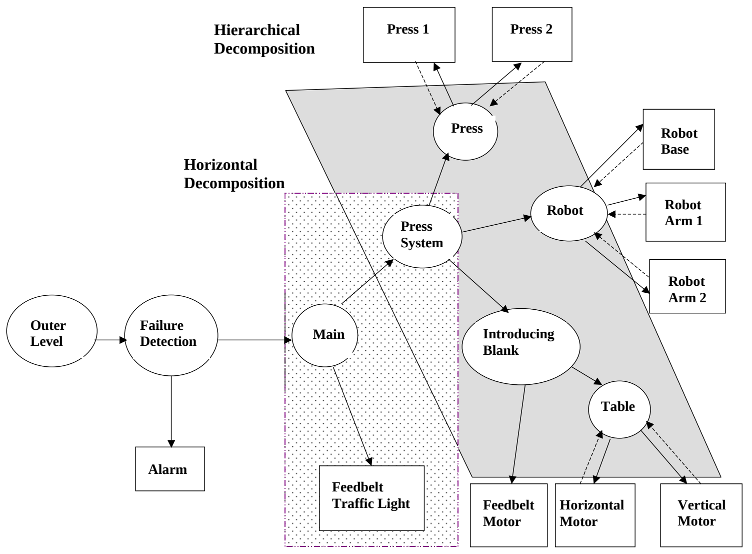


Figure 4.10: The DCFD for the fault-tolerant production cell.

Figure 4.11 visualises the table and feedbelt components with the sensors, $\{isOnTable, atTop, atBottom\}$ and $\{sw, atStart, atEnd\}$ respectively. All of these devices have two possible states: true and false; except for sw which has the states: on and off. We only consider the vertical motor $\{vertMotor\}$ as the actuator of the table that has three states (up, down, off) and the beltmotor $\{beltMotor\}$ as the actuator of the feedbelt that has two states (on, off). The control invariants for the table are:

$$isOnTable = true \wedge \quad atBottom = false \quad \Rightarrow \quad vertMotor = down \tag{4.7}$$

$$isOnTable = true \wedge \quad atBottom = true \quad \Rightarrow \quad vertMotor = off \tag{4.8}$$

$$isOnTable = false \wedge \quad atTop = false \quad \Rightarrow \quad vertMotor = up \tag{4.9}$$

$$isOnTable = false \wedge \quad atTop = true \quad \Rightarrow \quad vertMotor = off \tag{4.10}$$

The control invariants for the feedbelt are:

$$sw = off \quad \Rightarrow \quad atBottom = false \tag{4.11}$$

$$atEnd = false \wedge \quad sw = on \quad \Rightarrow \quad atBottom = true \tag{4.12}$$

$$atEnd = true \wedge \quad (atTop = false \vee \quad isOnTable = true) \quad \Rightarrow \quad atBottom = false \tag{4.13}$$

$$sw = on \wedge \quad atEnd = true \wedge \quad atTop = true \wedge \quad isOnTable = false \quad \Rightarrow \quad beltMotor = on \tag{4.14}$$

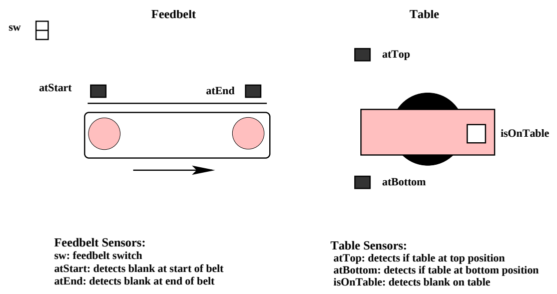

Feedbelt
Table
sw
atTop
atStart
atEnd
isOnTable
atBottom
Feedbelt Sensors:
sw: feedbelt switch
atStart: detects blank at start of belt
atEnd: detects blank at end of belt
Table Sensors:
atTop: detects if table at top position
atBottom: detects if table at bottom position
isOnTable: detects blank on table

Figure 4.11: The sensors for the feedbelt and table components.

Blanks are transferred from the feedbelt to the table only when the table is in the top position and clear of a blank. Therefore, the behaviour of the feedbelt subsystem cannot be described without considering the state of the table sensors. The feedbelt needs to know whether it is safe for it to move and to deposit a blank on the table:

$$atTop = true \ \& \ isOnTable = false \tag{4.15}$$

that is, the table is in the top position and empty of blanks (table is ready to receive a blank). We call this condition *stm* ("safe for feedbelt to move"). A virtual sensor *Stm* is thus introduced into the feedbelt subsystem as illustrated by the DCFD in Figure 4.12. The value of the state *stm* of *Stm* is the evaluation of the condition in 4.15. The invariants 4.13 and 4.14 are rewritten using *stm* as follows:

$$atEnd = true \ \& \ \ stm = false \Rightarrow atBottom = false$$
$$sw = on \ \& \ atEnd = true \ \& \ stm = true \ \Rightarrow \ atBottom = true$$

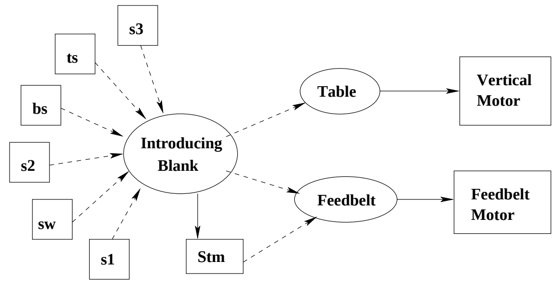


Figure 4.12: The DCFD of the feedbelt with the virtual sensor STM

We show how to model check the following global liveness property that involves both the feedbelt and the table sensors:

$$AG(AF(isOnTable = false)) \ \Rightarrow \ AG(AF(atEnd = false)) \tag{4.16}$$

which means that if blanks are infinitely often removed from the table then, they are infinitely often removed from the feedbelt. We use the following environmental assumptions, that are assumed to be true in SMV for the respective subsystems, to help prove the global property 4.16.

$$AG(vertMotor = up \ \Rightarrow \ AF(atTop = true)) \tag{4.17}$$
$$AG(onTable \Rightarrow \ atTop = true) \tag{4.18}$$
$$AG(atEnd = true \ \& \ stm = true \ \Rightarrow \ AF(atEnd = false)) \tag{4.19}$$

These environmental assumptions describe the basic physical properties for the behaviour of the table and feedbelt. Property 4.17 means that if the table motor is set to *up*, the table will eventually reach the top position. Property 4.18 means that blanks are only added when the table is in the top position i.e. if the event *onTable* occurs then the table is in the top position. Finally, property 4.19 means that if it is safe for the feedbelt to move (table is ready to receive a blank) and it is currently moving, then eventually it will stop.

From assumptions 4.17 and 4.18, we can prove, using SMV, the following local property in the table subsystem.

$$AG(isOnTable = false \ \Rightarrow \ AF(isOnTable = false \ \& \ atTop = true)) \tag{4.20}$$

which states that if there is no blank on the table, then eventually the table will be in the top position, i.e. it is ready to receive a blank.

Therefore, the complete property to be verified, defined under the *SPEC* clause in SMV, is:

```
SPEC
 AG(AG(C.event = onTable -> Mts.atTop = true) &
  AG(Mvm.vertMotor = up -> AF(Mts.atTop = true)) ->
  (Ms3.isOnTable = false -> AF(Ms3.isOnTable = false & Mts.atTop = true)) )
```

The environmental assumption 4.19 states that if a blank is detected at the end of the feedbelt and the table is in the position ready to receive a blank, then eventually there will be no blank detected at the end of the feedbelt.

We show mechanically that the global property 4.16 for this system is true by using environmental assumptions and the local property discussed. If infinitely often *isOnTable = false* is true (we assume LHS of implication of 4.16), then according to 4.20 sometime in the future

$$isOnTable = false \& atTop = true$$

If $atEnd = false$ then local property 4.19 is true. Otherwise, if $atEnd = true$, then because of 4.19, sometime in the future $atEnd = false$. Since $AG(AF(atEnd = false))$ is stronger than $AG(atEnd = true \ \& \ stm = true \ \Rightarrow \ AF(atEnd = false))$, we have shown that the global property 4.16 is true. Figure 4.13 illustrates a model that satisfies the global property. The SMV code generated for the table and feedbelt can be found in the appendix.

4.5.2 Horizontal composition of controllers

The horizontal decomposition approach divides the system into subcontrollers that process the sensor events independently from each other as illustrated in Figure 4.14. Each controller is

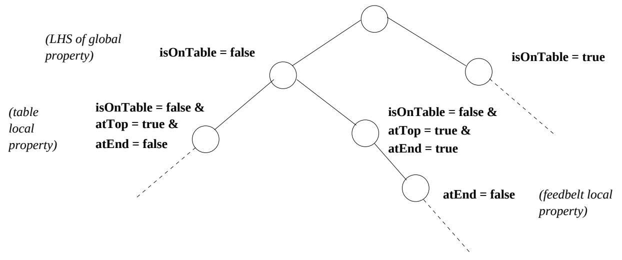

(LHS of global property)
isOnTable = false
isOnTable = true
(table local property)
isOnTable = false & atTop = true & atEnd = false
isOnTable = false & atTop = true & atEnd = true
atEnd = false
(feedbelt local property)

Figure 4.13: A Kripke model showing the intuition for proving global property.

considered as a separate subsystem that can be decomposed further if required. The controllers of this decomposed system can share sensor components but not actuator components and this is because it models real devices that are physically decomposed in this way. There are no invariants relating the state of one subsystem to the state of another.

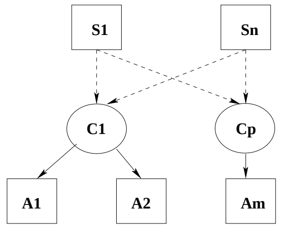


Figure 4.14: The DCFD for a horizontal system.

When model checking large systems that are decomposed horizontally, each subsystem can be represented as a separate SMV program and all verification is done locally. There are no properties that refer to states in both subsystems. The controller and actuator modules have the same schema as those presented. If the controllers share sensors, the sensor state will overlap in the definition of the controller state. For small systems, all of the subsystems are represented in the same SMV program as follows.

```
MODULE main
VAR
 C1 :  Controller1;
 ...
 Cn :  Controllern;

 A1 :  Actuator1;
 ...
 Ak :  Actuatorm;
```

4.5.3 Phase composition of controllers

For the phase decomposition (or decomposition by control mode), a separate controller is specified for the control reactions to be carried out in each mode or phase of the system. The structure of the system components is like the hierarchical one. The difference is that only one subcontroller will be active at a time, so the system could be divided into separate SMV programs for each active controller and verify properties of active controllers. There are therefore restraints on the overseeing controller to send events at each step to only one subcontroller (assumption). However, for liveness properties and if the controllers depend on each other in any way, they need to be either implemented in the same SMV program or make use of virtual sensors as mentioned in section 4.5.1. Furthermore, only one level of controllers can be modelled for coarse-grain specifications that are to be model checked.

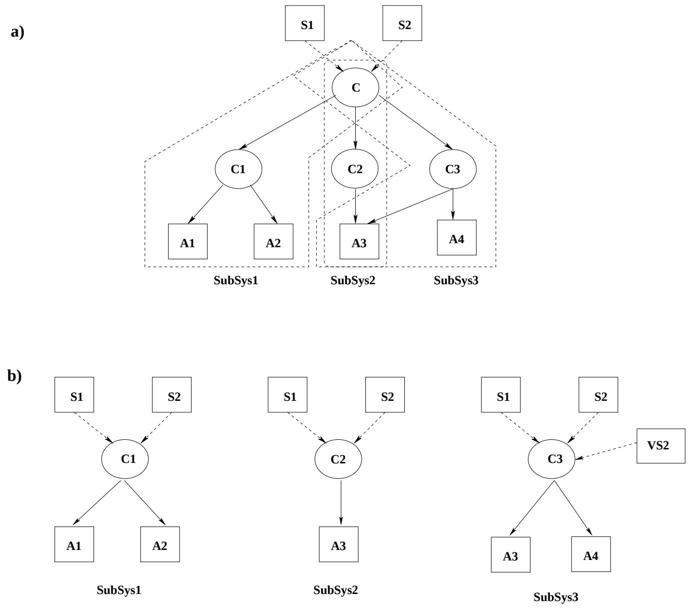


Figure 4.15: The DCFD for a system with a phase composition of controllers.

Lets consider the example system visualised in a) of Figure 4.15. It consists of three distinct modes where controller *C2* and *C3* share actuator *A3*. This system can be represented as three separate SMV programs, each with a DCFD as illustrated in b) of Figure 4.15. This is a valid representation as only one controller is ever active at a time. A virtual sensor *VS2* is introduced in subsystem *SubSys3* as it requires information from subsystem *SubSys2* to make a decision about controlling the shared actuator *A3*. It is sufficient for each subsystem to implement only one controller (the second is redundant) which makes it possible to model check the RSDS coarse-grain specification of this system.

4.5.4 Annealing

Annealing is when a separate subcontroller is created to encapsulate repeated control sequences as a single operation. Its aim is to encourage maintainability, that is, it allows updates to the current system specification to be made easily. For example, it would be used for specifying a chemical plant system where a single operation is defined for opening a number of valves and pumps to open one flow between vessels.

This decomposition approach is only applicable to B specifications as they are described in terms of operations. In SMV the use of annealing is counterintuitive. When translating to SMV, the modules make use of the synchronisation capabilities of SMV and don't need to explicitly call operations as in B. SMV is a low level language and cannot implement the high level concepts in B. If this approach was to be implemented in SMV, then it would result in complicating the SMV modules and increasing the state space unnecessarily with the introduction of a separate module, with more variables.

4.5.5 Standard controllers

This decomposition approach consists of recognising common control patterns in a system and chaining together suitable versions of these standard controllers to achieve a more complex control function. There are two standard controllers: the AND controller and the priority controller. We present the translation of the controller and actuator modules for systems described in Figures 4.16 and 4.17 that can be generated by applying the translation rules.

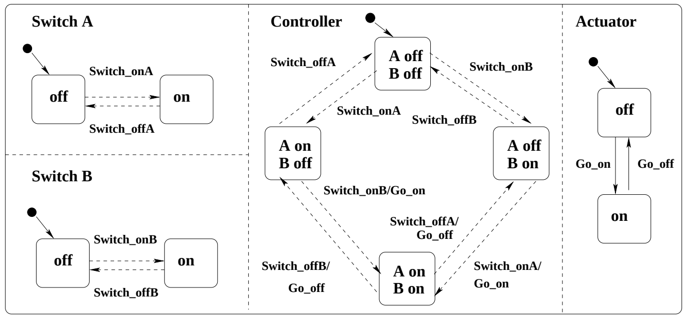


Figure 4.16: The SRS statemachine for the AND controller system.

The AND controller sets the actuator state to *on* only if both the sensor states are set to *on*, that is, it works like the logical "and" connective. The SMV modules for the controller and actuator generated are displayed below, where only controller transitions CT1 and CT5 can change the actuator state to *on*.

```
MODULE Controller
VAR
 stable_state :  {Aon_Boff, Aon_Bon, Aoff_Boff, Aoff_Bon};
 sensor_event :  {swonA, swonB, swoffA, swoffB };

DEFINE
 CT1 := sensor_event = swonA & stable_state = Aoff_Bon;
 CT2 := sensor_event = swonA & stable_state = Aoff_Boff;
 CT3 := sensor_event = swoffB & stable_state = Aoff_Bon;
 CT4 := sensor_event = swoffB & stable_state = Aon_Bon;
 CT5 := sensor_event = swonB & stable_state = Aon_Boff;
 CT6 := sensor_event = swonB & stable_state = Aoff_Boff;
 CT7 := sensor_event = swoffA & stable_state = Aon_Bon;
 CT8 := sensor_event = swoffA & stable_state = Aon_Boff;

ASSIGN
 init(stable_state) := Aoff_Boff;
 next(stable_state):=
   case
    CT1 | CT5 :  Aon_Bon;
    CT2 | CT4 :  Aon_Boff;
    CT3 :  Aoff_Boff;
    CT6 | CT7 :  Aoff_Bon;
    CT8 :  Aon_Boff;
    1:stable_state;
   esac;
```

```
MODULE Actuator(C)
VAR
 act :  {on, off};

ASSIGN
 init(act) := off;
 next(act):=
   case
    CT1 | CT5 :  on;
    CT2 | CT3 | CT4 | CT6 | CT7 | CT8 :off;
    1 :  act;
   esac;
```

The system with the priority controller has the same SMV controller module generated as that of the system with the AND controller. The priority controller manages two actuators and ensures that actuator *A* has priority over actuator *B*, that is, actuator *B* cannot be set to *on* if switch *A* is *on*. This is enforced by ensuring that the actuator states change according to the appropriate controller transition.

```
MODULE Controller
VAR
 stable_state :  {Aon_Boff, Aon_Bon, Aoff_Boff, Aoff_Bon};
 sensor_event :  {swonA, swonB, swoffA, swoffB };

DEFINE
 CT1 := sensor_event = swonA & stable_state = Aoff_Bon;
 CT2 := sensor_event = swonA & stable_state = Aoff_Boff;
```

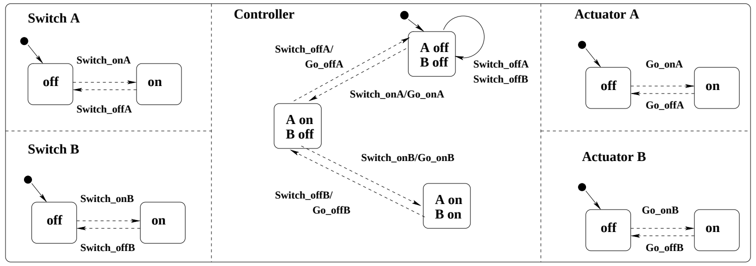


Figure 4.17: The SRS statemachine for the priority controller system.

```
  CT3 := sensor_event = swoffB & stable_state = Aoff_Bon;
  CT4 := sensor_event = swoffB & stable_state = Aon_Bon;
  CT5 := sensor_event = swonB & stable_state = Aon_Boff;
  CT6 := sensor_event = swonB & stable_state = Aoff_Boff;
  CT7 := sensor_event = swoffA & stable_state = Aon_Bon;
  CT8 := sensor_event = swoffA & stable_state = Aon_Boff;

ASSIGN
  init(stable_state) := Aoff_Boff;
  next(stable_state):=
    case
      CT1 | CT5 :  Aon_Bon;
      CT2 | CT4 :  Aon_Boff;
      CT3 :  Aoff_Boff;
      CT6 | CT7 :  Aoff_Bon;
      CT8 :  Aon_Boff;
      1:stable_state;
    esac;
```

```
MODULE ActuatorA(C)
VAR
 act :  {on, off};

ASSIGN
 init(act) := off;
 next(act):=
   case
     CT1 | CT2 :  on;
     CT7 | CT8 :  off;
     1:act;
   esac;
```

```
MODULE ActuatorB(C)
VAR
 act :  {on, off};

ASSIGN
 init(act) := off;
 next(act):=
   case
     CT6 :  on;
     1:act;
   esac;
```

4.6 Related Work

Model checking has been applied to various variants of statemachine and statechart notations that are used to describe the behaviour of either software or hardware. In this section, we discuss approaches that have defined translations from such notations (excluding UML statecharts as

this is presented in section 5.5) into SMV in order to apply model checking as they are directly comparable to our work. Moreover, we only consider translations that are applied to statechart variants that specify software. There have been two such significant translations: a translation [CAB$^+$98] from RSML [LHHR94] to SMV and a translation [CH00] from STATEMATE [HN96] to SMV.

The translation from RSML to SMV was defined to investigate the scalability of model checking large RSML specifications. RSML is a variant of Harel statecharts [HN96], that adopts the notions of OR states, AND states and the broadcast mechanism used for communicating from statecharts, and has some additional features such as directed communication between statecharts. Compared to RSDS statemachines, the RSML notation is more expressive, for example, it allows multiple transitions to occur from a state (i.e. non-deterministic transitions), and it allows for OR states to include AND states. A step in RSML is defined by the system's reaction to external events as a succession of microsteps from the point when they occur until the system is stable. A microstep consists of the state change for each interim. The system is stable at the point where no transitions are generated, after a cascading of events initiated by the occurrence of sensor events. The synchrony hypothesis assumes that once a step is initiated, no sensor events can arrive until the system is stable. This hypothesis is similar to that of RSDS, although RSDS receives and processes only one sensor event at a time, while RSML can receive and process many.

Despite the differences between the semantics of RSML statecharts and SRS statemachines, we decided to implement the gas burner system using the translation algorithm in [CAB$^+$98] in order to compare its performance with that of RSDS. The SMV code (manually generated) can be found in Appendix B.1. We describe the general rules for translating from RSML to SMV and we also describe which were used for translating the gas burner system:

1. The state hierarchy, inputs and events are translated into SMV variables. OR states are translated into variables of enumerated type which ranges over the substates of the OR-states. Events are translated into boolean variables, meaning that more than one event can be true during an SMV step. Since RSDS specifications only receive one sensor event at a time, we model this assumption in SMV by translating the sensor event into a variable of enumerated type, while the rest of the events are translated into boolean variables. Input variables are translated into variables with an enumerated or integer subrange type. The gas burner system does not have any input variables.

2. Variables under the DEFINE clause (that do not increase the state space) are defined to indicate the conditions under which a particular state is in. For example, for the gas burner system, the conditions under which SwOn is true are:

```
DEFINE
  in-Sys := 1;
  in-Switch := in-Sys;
  in-SwOn := in-Switch & Switch = On;
```

3. The transitions are translated into variables under the DEFINE clause. They are composed of: the source state, the trigger event and the guarding condition. All these must be true in order for a transition to occur.

4. The state changes are defined using the *next* clause of SMV with an inner *case* statement that ranges over all the transitions in defined. Also, the *next* clause is used to define the events that are generated from transitions. For example, the event av_open in the gas burner system is generated by transitions $CT1$, $CT4$, $CT6$, $CT7$ as is defined as follows:

```
ASSIGN
  next(av_open):= CT1|CT4|CT6|CT7;
```

The states and events are initialised using the *init* in SMV.

5. Sensor events are modelled in SMV non-deterministically, i.e. any event can happen. However, in order to satisfy the synchrony hypothesis, the sensors only occur when the system is stable and do not change while the system is not stable. Therefore, a variable, called stable, is defined under the DEFINE clause which is the conjunction of the negation of all events. For the gas burner system, since we have defined the sensor event as an enumerated type, we add the additional value of *none*. The stable state can be defined as follows:

```
DEFINE
  stable := event=none & !av_close & !av_open & !gv_close &
    !gv_close & !ig_close & !ig_open;
```

The key differences concerning the syntax and semantics of the translation with respect to our translation are:

1. The events are translated differently in the RSDS translation. The sensor events are translated into a variable of enumerated type. This is to ensure that only one event occurs at a time. In RSML many events can occur at a time. The generated events are not explicitly represented in the SMV model for RSDS specifications, while they are in the SMV model for RSML specifications. This is because we want a coarse-grain step to correspond to a single SMV step. In the RSML translation, each microstep corresponds to an SMV step, and also, the system is stable for one SMV step.

   States are similarly translated.

2. RSDS does not explicitly represent the state hierarchy with variables (and conditions) defined under the DEFINE clause.

3. Transitions are translated in the same way for both RSDS and RSML.

4. The state changes are modelled in the same way for both RSDS and RSML, as well as the initial state values.

5. Both RSDS and RSML define the sensor events non-deterministically, i.e. any event can happen. However, the RSML model allows for many events to happen at a time, while the RSDS translation allows only one sensor event to occur. The synchrony hypothesis of RSML states that the sensor events cannot change during a step, and can only occur when the system is stable. Since an RSML step corresponds to a number of SMV step, additional

SMV code is added to ensure that the environment only changes when the system is stable. Therefore, a variable is added under the DEFINE clause that does not increase the state space, and it is used as a condition when defining the change of an event. For the coarse-grain translation of RSDS, a step corresponds a single SMV step and sensor events can occur at every step.

6. The translation [CAB+98] does not generate a modular SMV model. In general, this is not a limitation because the SMV structure is flat, that is, all the modules are eventually flattened. Modularity in SMV only enhances readability and maintainability of models.

7. A proof of correctness of the RSML translation has not been given.

The following table presents the resources used for executing both SMV models of the gas burner system generated by the translations:

nuSMV resources used	RSDS translation	[CAB+98] translation
User time:	0.06 seconds	0.31 seconds
System time:	0.04 seconds	0.06 seconds
Virtual data size (bytes allocated):	6357K	6901K
BDD nodes allocated:	733	27669
BDD cluster size:	59	18459

The performance of model checking the SMV programs generated by both translations is equally good. Since the values for the performance time are very small and thus not accurate enough, we cannot assert that the results of model checking the one SMV model is indeed better than the other. What is interesting however, is that the number of BDD nodes allocated for the SMV model generated by the RSML translation is much higher than that of the SMV model generated by the RSDS translation. Therefore, we believe that potentially large systems whose SMV model is generated by the RSML translation that cannot be model checked due to the state space explosion problem, might still be model checked if translated using the RSDS translation.

The second translation [CH00] adopts a modular approach for translating a fragment of the STATEMATE language into SMV that has been automated. The approach is modular because the hierarchical structure, AND-charts and OR-charts, of statecharts is preserved in SMV where possible. The top level statechart is represented directly in the main module, which also contains declarations for all global variables for events and conditions. Subcharts are translated into separate SMV modules known as chart modules. Also, separate modules are defined for handling global events and condition variables, that are collectively called monitor-modules. All elements of these statecharts are represented explicitly, for example, each global event is represented as a boolean variable and these are declared in a separate SMV module. Inter-level transitions cannot be translated, and nor can the STATEMATE priority scheme for conflicting transitions. The translation is described by first giving the formal definition of an element in the statecharts and then showing how it is defined in SMV. In addition, the SMV language is given a formal temporal logic definition.

If the STATEMATE translation were to be used to translate a reactive system specification, we believe that the SMV code generated would be very complicated as a large number of events are usually being broadcasted between the components. This would lead to an SMV model with a large state space that could drastically affect the performance of the model checker. Nevertheless, the translation [CH00] can be applied to statecharts defining the behaviour of a wider range of applications, modelling them exactly.

4.7 Summary

In this chapter, we presented a way of model checking the coarse-grain view of RSDS specifications by translating into the input language of the SMV model checker. We gave a brief overview of the available tools that implement the SMV algorithm and described what their main constituents are. Only a subset of the SMV language is used for the translation.

The translation is given using a number of translation rules and translation schemas that can be expanded for individual systems. A coarse-grain step corresponds to an SMV step. Each statemachine module of RSDS specifications is translated into an SMV module. The sensor, controller and actuator transitions are synchronised to ensure that the reaction to a sensor event happens in a single SMV step. These rules are summarised in Table 4.1. The gas burner system was used to illustrate how to generate SMV code using the translation rules.

We gave a proof of correctness to show that the translation preserves the meaning of the RSDS specification. In order to prove the correctness of the translation, an axiomatic semantics is described for the SMV language.

In addition, design decomposition techniques have been used to manage the state space explosion problem of model checking. The decomposition approaches divide a system into more manageable subsystems and we propose defining a separate SMV program for each subsystem. Therefore, verification is done locally where possible. Table 4.2 summarises how (if at all) the decomposition approaches are applied to the SMV model.

We end this chapter with a discussion of related work, that is, other definitions of translations into SMV from modelling notations whose semantics are based on statecharts or statemachines. In the next chapter, we present the translation from the fine-grain view of the RSDS specification to SMV.

Table 4.1: Summary of translation rules for the coarse-grain

	SRS Statemachine	**SMV**

Rule 1:	Each controller module C in an RSDS system *Sys*	`VAR`
		`C : Controller;`

Rule 2:	Each actuator module A in *Sys*	`VAR`
		`A : Actuator(C, OA);`
		where C is the controller responsible for managing actuator A and OA are any other actuator modules that it will need to refer to.

Rule 3:	For a state *st* in a module that ranges over the set of possible states $(s_1, s_2, ..., s_m)$:	`VAR`
		`st :  {s1, s2, ...,sm};`

Rule 4:	The initial state of a module	`ASSIGN`
		`init(st) := init_s;`
		where *st* represents the current state.

Rule 5:	Each controller transition tr
	Similarly for actuator transitions
		`ASSIGN`
		`next(st) :=`
		`case`
		–For each transition tr with target state s2
		`tr :  s2;`
		–For the default case
		`1 :  st;`
		`esac;`

Rule 6:	The sensor event that currently occurs	`VAR`
		`sensor_event :  {e1, e2, ...,en};`
		where $sensor\_event \in ExtEvents$ and $e1, e2, ..., en$ are all possible events.

Rule 7:	Each controller transition *CT1,...,CTk*	`DEFINE`
		`CT1 := ex = e1 & st = x1;`
		Similarly for all controller transitions

Rule 8:	Each actuator transition *AT1,...,ATy:*	`DEFINE`
		`AT1 := C.CT1 & G & a_state = a1;`
		where *C.CT1* refers to controller transition and G is the guard and *a_state* is the actuator state.

Rule 9:	Each temporal invariant T	`SPEC`
		`AG(AG(Env) -> T)`
		where *Env* is the environmental assumptions.

Table 4.2: Summary of translation of decomposition approaches

Decomposition Approach	**SMV**	**Reduce state space explosion problem**
Hierarchical	Only one level of controllers is translated using the translation rules in Table 4.1.	Using virtual sensors for verifying global properties of system whose subsystems are defined as separate SMV programs.
Horizontal	Each subsystem is translatedinto a separate SMV program. Only one level of controllers is translated using the translation rules in Table 4.1.	Each subsystem verifies its properties independently.
Phase	Each phase can be translated into a separate SMV program. Only one level of controllers translated using the translation rules in Table 4.1.	For global properties, virtual sensors are used. Properties for each active controller are verified independently.
Annealing	There are no benefits to translate this approach	N/A
Standard controllers: priority and AND controller	Translated using the translation rules in Table 4.1.	N/A

CHAPTER 5

Applying Model Checking to the Fine-grain View of RSDS

The fine-grain semantic view of RSDS statemachines provides a much finer level of granularity of computation than that of the coarse-grain view. A single step in the coarse-grain, which occurs between two stable states, corresponds to a reaction cycle in the fine-grain. A reaction cycle is the system's response to an external event given as a finite number of steps where the endpoints consist of stable states and the intermediate states are known as unstable states. In figure 5.1 a coarse-grain step between *stable_state1* and *stable_state2* corresponds to a reaction cycle with fine-grain steps for the external event and for an ordered sequence of actuator events. Consequently, the fine-grain models the exact order of actuator events that cannot be modelled in the coarse-grain view. We have already described how to model check the coarse-grain view of RSDS specifications where the properties are shown to hold in the stable states. However, this is not enough for the fine-grain view where the properties must hold in the unstable states too. For example, in the coarse-grain representation of the gas burner system we have shown that the invariant $gv = open \Rightarrow av = open$ is true in the stable states, but in the fine-grain representation this invariant must hold in all states. Violations of properties by unstable states may lead to the development of unsafe systems. Therefore, we want to verify the fine-grain view of RSDS as well.

In this chapter, we complement the work of the previous chapter with a translation from the fine-grain RSDS statemachines to SMV. We focus on keeping the size of the SMV source as small as possible for improving the efficiency of model checking. We guarantee the quality of

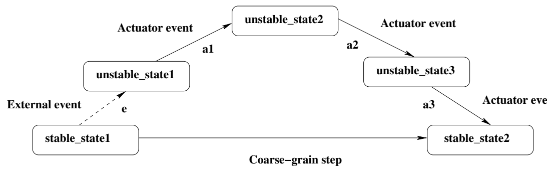


Figure 5.1: A reaction cycle for event **e** in the fine-grain semantic view of flattened SRS statemachines that corresponds to a single coarse-grain step.

the translation by formally proving its correctness. Furthermore, we apply the decomposition approaches of RSDS to the fine-grain SMV models generated, describing new translation rules where necessary. We exploit the structure of the decomposed systems to define, where possible, separate SMV programs for subsystems that can be model checked independently, thus defining a natural way to reduce the state space of the models.

5.1 Fine-grain Translation from RSDS to SMV

In the fine-grain, the distinction between external (sensor) events and internal (generated) events is important since external events indicate the start of a reaction cycle while internal events trigger transitions that are part of the reaction. In Figure 5.1, the external event is e and the internal events are $\{a1, a2, a3\}$. Each fine-grain step is triggered by a single event, either external or internal.

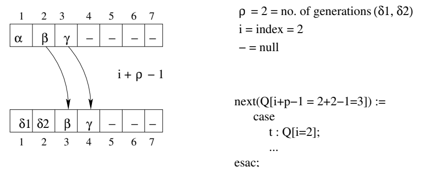
In this section, a translation from the fine-grain view of RSDS to SMV is presented. As with the coarse-grain translation, it is described using a set of translation rules and SMV schemas. A fine-grain step is modelled as a single SMV step. The order in which internal events are processed is significant. To guarantee that this ordering is modelled correctly in SMV, the concept of an event queue is adopted from the definition of the fine-grain semantics in section 3.2.4 and modelled in SMV. Since the SMV model mimics the behaviour of the fine-grain semantic view of RSDS, it is easier to prove that the translation preserves the semantics.

5.1.1 Pre-processing and translation of the event queue

For the fine-grain translation, the array construct in SMV is used for representing the queue of events and since it is static, the maximum length of the array must be calculated before generating the translation. Therefore, in order to prepare for an automatic translation, a pre-processing phase is required that consists of:

- Calculating the maximum size of an array by iterating through the number of levels of components (depending on the decomposition approach chosen), and returning the largest number of internal events generated by a transition. For most simple reactive systems specified in RSDS without subcontrollers, the maximum number of event generations correspond to the total number of actuators in the system as the controller usually issues at most one command for each actuator. For example, if there are three actuators as in the gas burner system, then the controller transitions generate a maximum of three events which become the maximum array size of the queue.

- Automatically determining the position in the array (index) where an element in the tail of the queue will be moved to after a transition has occurred. This is calculated by: $i + \rho - 1$ where i is the index of any element in the tail of the queue, and ρ is the number of generations for a transition (including actuator transitions) and 1 represents the element that is removed from the head of the queue. If the elements in the array are less than the maximum size of the array, then the rest of the elements are equal to **null**. Figure 5.2 illustrates how the index is calculated for an array of events, where $\rho = 2$, $i = 2$ and the symbol $-$ represents **null**.



```
next(Q[i+p−1 = 2+2−1=3]) :=
   case
      t : Q[i=2];
      ...
esac;
```

Figure 5.2: Example of how the positions in the next snapshot of the queue are calculated.

- Determining the form of the array after an actuator transition occurs: the head of the queue is removed and the elements in the array are shifted to the left, making the last element equal to null.

With this information, the queue can be represented in SMV as an array of length *maximum length*. It is initially empty and is built by using the next construct for each element in the array. Each controller transition is triggered by a sensor event and its generations are added to the queue. If a system is further decomposed and contains subcontrollers, then the internal event found at the head of the queue that triggers the subcontroller transition, is replaced with the transition's generated events. If the transition has no generations, then the event at the head of the queue that triggered the transition is removed and the elements in the array are shifted to the left.

A module, separate from the modules representing system components, is created for defining the event queue. Its parameters are all of the other modules collectively representing the system components because it requires read-access to their transitions. The following translation rule shows how this module is defined in the main module of SMV.

Rule 1: *An SMV module* **Queue** *for defining the event queue*

```
MODULE main
  VAR
    Event_queue :  Queue(Cont, Act1,...,Actp);
```

where Cont is the controller module and Act1...Actp are the p actuators of a system.

As we have already mentioned, the event queue is declared using the array construct in SMV. The boolean variable *is_empty* is introduced to record whether the queue is empty after a transition occurs at each step. Actuator and subcontroller transitions with no event generations change the value of *is_empty* to 1, if the internal event that triggered the transition is the only element left in the queue. Usually this requires a check of the second element in the array to ensure that equals null. The translation rules for declaring and initialising the event queue are defined as follows. The *init* and *next* clauses for an array define the initial and next states for each element in the array, i.e. they describe the state changes for each position in the array.

```
Rule 2:   For each event queue Q with a set of
          possible generated events (ge_1, ge_2, ..., ge_m) :
          VAR
           Q : array 1..max of {ge1, ge2, ...,gem, null};
          where max is the maximum length of the array.

Rule 3:   To determine whether the queue is empty:
          VAR
           is_empty :  boolean;

Rule 4:   Initialising the queue:
          ASSIGN
           init(is_empty) := 1;
           init(Q[1]) := null;
           ...
           init(Q[max]) := null;
          Each element is set to null and is_empty to true.
```

The event queue is updated in one of two ways:

1. *Add events to the queue:* A controller or subcontroller transition is triggered by an internal event at the head of the queue. If the transition generates any events, then they are added to the front of the queue while simultaneously replacing the internal event that triggered the transition found at head of the queue and pushing the tail of the queue further to the right. The following translation rule explains how this is modelled in SMV.

```
Rule 5:   For indexes i:1..p of a sequence of generated events ge produced,
          the transition tr that are added to the head of the queue:
          ASSIGN
           next(Q[i]) :=
             case
               –For each transition tr that generates events
               tr :  ge(i);
               ...
               –Default case, value in the array stays the same.
               1:Q[i]
             esac;

           The elements at the tail of the queue are moved to their new array
           positions j where j= i+p-1 where:
           i is the index of the event in the tail of the queue and
           p is the number of generated events for a transition.
           next(Q[j]) :=
             case
               –For each transition tr that generates events
               tr :  Q[i];
               ...
               –Default case, value in the array stays the same.
               1:Q[j]
             esac;

           next(is_empty) :=
             case
```

```
               –If a transition that generates events
               –occurs then the queue is not empty
               tr :  0;
               ...
               –Default case, value in the array stays the same.
               1:is_empty
             esac;
```

2. *Remove the head of the queue:* An event is removed from the head of the queue when that event triggers an actuator or subcontroller transition. If the transition generates internal events these are added to the head of the queue replacing the triggered event (see translation rule 5). If no events are generated, then the triggered event is simply removed. Only one event is removed at each step. Translation rule 6 shows how this is modelled in SMV.

Rule 6: *For each event that is processed by an actuator transition* **ATr**

```
ASSIGN
 –For each element in the queue where x is the array index:
 next(Q[x]) :=
   case
     –The queue is shifted to the left as the event
     – at the head is removed.
     ATr :  Q[x+1];
     ...
     –Default case, value in the array stays the same.
     1:Q[x]
   esac;

j= i+p-1 where:
i is the index of the event in the tail of the queue and
p is the number of generated events for a transition.
 next(Q[j]) :=
   case
     –For each transition tr that generates events
     tr :  Q[i];
     ...
     –Default case, value in the array stays the same.
     1:Q[j]
   esac;

 next(Q[max]) :=
   case
     –the last element in the queue is set to null
     ATr :  null;
     ...
     –Default case, value in the array stays the same.
     1:Q[max]
   esac;

 next(is_empty) :=
   case
     –If a transition occurs and the second element
     –in the queue is null, then the queue is empty.
     ATr & Q[2]=null :  1;
     ...
     –Default case, value in the array stays the same.
     1:is_empty
   esac;
```

The following translation schema for the event queue module shows how to declare, initialise and modify the array representing the event queue and its status variable *is_empty*.

```
MODULE Queue(C, A)
VAR
 Q: array 1..max of {ge1, ge2, ..., geq};
 is_empty :  boolean;

ASSIGN
 init(is_empty) := 1;
 next(is_empty):=
```

```
  case
   C.CT1 :  0;
   SC.SCT1 :  0;
   ...
   A.AT1 :  0;
   A.AT2 & Q[2]=null :  1;
   ...
   1:  is_empty;
  esac;

 init(Q[1]) := null;
 next(Q[1]):=
  case
   C.CT1 :  ge1;
   SC.SCT1 :  ge3;
   ...
   A.AT1 :  Q[2];
   ...
   1:  Q[1];
  esac;
 ...

 init(Q[max]) := null;
 next(Q[max]):=
  case
   C.CT1 :  null;
   SC.SCT2 :  ge2;
   ...
   A.AT1 :  null;
   ...
   1:  Q[max];
  esac;
```

For the gas burner system, once the ordering of the generated events has been determined from the invariants, the SMV module for the queue is generated based on the snapshots of the queue in figure 5.3. The snapshots of the queue illustrate how the queue is updated with each fine-grain step, i.e. with each transition that occurs, and is generated automatically by the RSDS tool from the RSDS specification. For example, in Figure 5.3, the controller transition CT1 occurs when the event queue is empty and generates three events, av_open, gv_open and ig_open. These are then responded to one by one by an actuator and hence are removed from the queue (from the head). Once the queue is empty, another controller transition can be enable and so on.

As for all reactive systems, controller transitions $\{CT1,...,CT8\}$ of the gas burner are triggered by external events i.e. when there are no internal events in the queue waiting to be processed. These are thus responsible for initiating each reaction cycle. In figure 5.3 the reaction cycle initiated by each controller transition is illustrated. The actuator transitions within the reaction cycle are not explicitly labelled in the figure but their effect on the event queue for each fine-grain step is illustrated.

The maximum size of the event queue at any time is three and therefore this is the size of the array in SMV. The queue module has read-access to all other system modules i.e. it has parameters for the controller and the three actuators. Since an array in SMV is modified one position at a time, the effect of a transition is perceived as the collection of changes brought about

CT1	[]
	[av_open, gv_open, ig_open]
	[gv_open, ig_open]
	[ig_open]
CT2	[]
	[ig_close, gv_close, av_close]
	[gv_close, av_close]
	[av_close]
CT3	[]
	[ig_close, gv_close]
	[gv_close]
CT4	[]
	[av_open, gv_open]
	[gv_open]
CT5	[]
	[ig_open]
CT6	[]
	[ig_close, av_open]
	[av_open]
CT7	[]
	[ig_closed, av_open]
	[av_open]
CT8	[]
	[av_close]
	[]

Figure 5.3: A snapshot of the event queue for the gas burner system.

by that transition for each array position. For example, the controller transition CT1 for the gas burner adds the events {*av_open, gv_open, ig_open*} to the queue, i.e. Q[1] = *av_open*, Q[2] = *gv_open* and Q[3] = *ig_open*. Moreover, by looking at the *next* changes for the third element in the array, we deduce that transitions CT1 and CT2 are the only transitions that generate three internal events. This fact is confirmed by figure 5.3.

```
MODULE Queue(C, AV, GV, IG)
VAR
 Q: array 1..3 of {av_close, av_open, gv_close, gv_open, ig_close,ig_open, null};
 is_empty :  boolean;

ASSIGN
 init(is_empty) := 1;
 next(is_empty):=
  case
   C.CT1 :  0;
   C.CT2 :  0;
   C.CT3 :  0;
   C.CT4 :  0;
   C.CT5 :  0;
   C.CT6 :  0;
   C.CT7 :  0;
   C.CT8 :  0;
   AV.AVT0 | AV.AVT1 & Q[2]=null :  1;
   AV.AVT2 | AV.AVT3 & Q[2]=null :  1;
   GV.GVT0 | GV.GVT1 & Q[2]=null :  1;
   GV.GVT2 | GV.GVT3 & Q[2]=null :  1;
   IG.IGT0 | IG.IGT1 & Q[2]=null :  1;
   IG.IGT2 | IG.IGT3 & Q[2]=null :  1;
   1:  is_empty;
  esac;

 init(Q[1]) := null;
 next(Q[1]):=
  case
   C.CT1 :  av_open;
```

```
   C.CT2 :  ig_close;
   C.CT3 :  ig_close;
   C.CT4 :  av_open;
   C.CT5 :  ig_close;
   C.CT6 :  ig_close;
   C.CT7 :  ig_open;
   C.CT8 :  av_close;
   AV.AVT0 | AV.AVT1 :  Q[2];
   AV.AVT2 | AV.AVT3 :  Q[2];
   GV.GVT0 | GV.GVT1 :  Q[2];
   GV.GVT2 | GV.GVT3 :  Q[2];
   IG.IGT0 | IG.IGT1 :  Q[2];
   IG.IGT2 | IG.IGT3 :  Q[2];
   1:  Q[1];
  esac;

 init(Q[2]) := null;
 next(Q[2]):=
  case
   C.CT1 :  gv_open;
   C.CT2 :  gv_close;
   C.CT3 :  gv_close;
   C.CT4 :  gv_open;
   C.CT5 :  av_open;
   C.CT6 :  av_open;
   C.CT7 :  null;
   C.CT8 :  null;
   AV.AVT0 | AV.AVT1 :  Q[3];
   AV.AVT2 | AV.AVT3 :  Q[3];
   GV.GVT0 | GV.GVT1 :  Q[3];
   GV.GVT2 | GV.GVT3 :  Q[3];
   IG.IGT0 | IG.IGT1 :  Q[3];
   IG.IGT2 | IG.IGT3 :  Q[3];
   1:  Q[2];
  esac;

 init(Q[3]) := null;
 next(Q[3]):=
  case
   C.CT1 :  ig_open;
   C.CT2 :  av_close;
   C.CT3 :  null;
   C.CT4 :  null;
   C.CT5 :  null;
   C.CT6 :  null;
   C.CT7 :  null;
   C.CT8 :  null;
   AV.AVT0 | AV.AVT1 :  null;
   AV.AVT2 | AV.AVT3 :  null;
   GV.GVT0 | GV.GVT1 :  null;
   GV.GVT2 | GV.GVT3 :  null;
   IG.IGT0 | IG.IGT1 :  null;
```

```
    IG.IGT2 | IG.IGT3 :  null;
    1:  Q[3];
   esac;
```

5.1.2 Translation of the system components

The modular structure of the SMV generated for the fine-grain is similar to that of the coarse-grain, i.e. an SMV module is defined for each RSDS module. Also, the sensor modules are not explicitly translated into SMV for the same reason as that given in the coarse-grain. However, component transitions are defined differently as in the fine-grain they have to consider the event queue.

Large reactive systems have controllers that can be decomposed into several subcontrollers by applying the RSDS decomposition approaches. In the coarse-grain, subcontrollers were not translated as they would only result in redundant modules in the SMV model and no other benefits were realised. However, in the fine-grain, subcontrollers can be translated and they do help in improving the readability of the SMV models generated. We describe the subcontrollers as part of the standard components because their schema is similar to that of the actuator component.

Controller

In the controller module **C**, two variables are declared. The **stable_state** variable that represents the current state of the controller and the **sensor_event** variable represents the sensor event that is currently received by the current stable state. Stable states and sensor events correspond to those in the coarse-grain implementation of the controller. Internal events are generated by the controller or subcontroller and are received by either subcontrollers or actuators. These are collected in the event queue. The controller module contains as a parameter the event queue module in order to have read-access to the status variable of the event queue (*is_ empty*). The status variable of the queue is used as part of the condition for taking that transition as a controller transition can only occur if the event queue is empty. As for the coarse-grain, the guards of controller transitions are not explicitly represented in SMV because controller states contain all the information of all the sensor states in the system. Thus, it is sufficient to use the appropriate controller state to define the transition. If no guard is given, then it is assumed to be true. The translation rules below show how to map the RSDS statemachine elements of the controller module to SMV as described.

Rule 7: *For each state defined in the controller:*
```
VAR
 stable_state :  {stb1, stb2, ...,stbn};
```
where $stb_1, stb_2, ..., stb_n$ is the set of possible stable states.

Rule 8: *Initialising the controller state:*
```
ASSIGN
 init(stable_state) := init_stb;
```
where init_stb is the initial stable state.

Rule 9: *For sensor events in the controller:*
```
VAR
 sensor_event :  {ex1,ex2,...,exp};
```

Rule 10: *For each controller transition:*
```
DEFINE
 CT0 := stable_state = stb1 & sensor_event = ex1 & q.is_empty = 1;
```
where q refers to the module with the event queue.

Rule 11: *For each controller state:*
```
ASSIGN
 next(stable_state) :=
   case
     –For each controller transition CTi
     CTi :  stb2;
     ...
     –For the default case
     1 :  stable_state;
   esac;
```

The translation schema for the controller module is defined as follows, where the transition **CT0** ($e1 : c1 \rightarrow c2$) is triggered only if there are no internal events in the queue.

```
MODULE Controller(q)
VAR
  stable_state :  {c1, ..., cm};
  sensor_event :  {e1, ..., er};

DEFINE
 CT0 := stable_state = c1 & sensor_event = e1 & q.is_empty = 1;
 ...

ASSIGN
 init(stable_state) := c_init;
 next(stable_state):=
  case
   CT0 :  c2;
   ...
   1:  stable_state;
  esac;
```

The fine-grain controller module for the gas burner is similar to the coarse-grain controller module, except for the definition of the transitions CT_i where they can only occur if the event queue is empty.

```
MODULE Controller(q)
VAR
  stable_state :  {Off_Absent, On_Absent, Off_Present, On_Present};
  sensor_event :  {swon, swoff, fdon, fdoff};

DEFINE
 CT1 := stable_state = Off_Absent & sensor_event = swon & q.is_empty = 1;
 CT2 := stable_state = On_Absent & sensor_event = swoff & q.is_empty = 1;
 CT3 := stable_state = On_Present & sensor_event = swoff & q.is_empty = 1;
 CT4 := stable_state = Off_Present & sensor_event = swon & q.is_empty = 1;
 CT5 := stable_state = On_Absent & sensor_event = fdoff & q.is_empty = 1;
 CT6 := stable_state = On_Absent & sensor_event = fdon & q.is_empty = 1;
 CT7 := stable_state = Off_Absent & sensor_event = fdon & q.is_empty = 1;
 CT8 := stable_state = Off_Absent & sensor_event = fdoff & q.is_empty = 1;

ASSIGN
 init(stable_state) := Off_Absent;
 next(stable_state):=
  case
   CT1 :  On_Absent;
   CT2 :  Off_Absent;
   CT3 :  Off_Absent;
   CT4 :  On_Present;
   CT5 :  On_Absent;
   CT6 :  On_Present;
   CT7 :  Off_Present;
   CT8 :  Off_Absent;
   1:  stable_state;
  esac;
```

Subcontrollers and actuators

The SMV modules for subcontrollers and actuators are derived similarly. Both have transitions that are triggered by an internal event at the head of queue, i.e. one of the conditions defined in the transition is that the event queue is not empty, and the guard is true. Moreover, in SMV the guard of the actuator transitions must be defined explicitly, otherwise it is assumed to be true. The following translation rules describe how these components are defined as SMV modules.

Rule 12: *For each actuator or subcontroller state:*
```
VAR
 st :  {st1, st2, ...,stq };
```
where $st_1, st_2, ..., st_q$ is the set of possible actuator or subcontroller states.

Rule 13: *For each actuator or subcontroller transition:*
```
DEFINE
 AT0 := st = st1 & Ma.act & q.is_empty = 0 & q.Q[1] = e2;
```
where q refers to the event queue module and Ma.act is the guard.
The guard is a condition that refers to other actuator states.

Rule 14: *For each actuator or subcontroller state:*
```
ASSIGN
 next(st) :=
   case
    –For each actuator or subcontroller transition ATi
    ATi :  st2;
    ...
    –For the default case
    1 :  st;
   esac;
```

Subcontrollers and actuators have the same schema in SMV. The schema below can be used for subcontrollers by replacing the labels with those appropriate for the subcontroller. They both have read-only access to the event queue, hence the module parameter, because the actuator or subcontroller transition is only triggered by the event at the head of the queue. Furthermore, they have read-only access to any actuator that is used to compose the guard. The translation of the guard differs from that discussed in the coarse-grain as in the fine-grain the subcontroller and

actuator transitions are not synchronised but are triggered by generated events and in subsequent steps. Therefore, there are two cases that are considered:

1. The case where the actuators refer to the states of actuators within their subsystem, then each actuator module will have read-only access to those actuator modules whose state they refer to in the guard. If the subcontroller has a guard that refers to any actuator state, then the subcontroller is parameterised with the respective actuator modules. This duplication of the visibility of modules is not necessary in the coarse-grain because the transitions are synchronised. In the fine-grain we need to ensure that the guard is still true in the next step.
2. The second case considers actuators that refer to the states of actuators from other subsystems. This is rare as one of the main objective of the decomposition approaches is to localise behaviour within the subsystem. Anyhow, if this is the case, then the actuators must be parameterised with actuator modules from other subsystems whose state they refer to in the guard.

```
MODULE Actuator(q, A1)
VAR
 act :  {a1, ..., ah};

DEFINE
  AT0 := q.is_empty = 0 & A1.act = aa1 & q.Q[1] = ge3 & act = a1;
 ...

ASSIGN
 init(act) := act_init;
 next(act):=
  case
    AT0:  a2;
   ...
   1:  act;
  esac;
```

The gas burner system consists of three actuators: the air valve, the gas valve and the ignitor. They all have transitions that are triggered by events at the head of the event queue. Self-transitions must be explicitly defined in order to ensure that the head of the queue is removed even though the state is not changed. The actuator modules for the gas burner are described in SMV as follows.

```
MODULE AirValve(q)
VAR
 av :  {closed, open};

DEFINE
 AVT0 := q.is_empty = 0 & q.Q[1] = av_close & av = closed;
 AVT1 := q.is_empty = 0 & q.Q[1] = av_close & av = open;
 AVT2 := q.is_empty = 0 & q.Q[1] = av_open & av = closed;
 AVT3 := q.is_empty = 0 & q.Q[1] = av_open & av = open;
```

```
ASSIGN
 init(av) := closed;
 next(av):=
  case
    AVT0 | AVT1 :  closed;
    AVT2 | AVT3 :  open;
   1:  av;
  esac;
```

```
MODULE GasValve(q)
VAR
 gv :  {closed, open};

DEFINE
 GVT0 := q.is_empty = 0 & q.Q[1] = gv_close & gv = closed;
 GVT1 := q.is_empty = 0 & q.Q[1] = gv_close & gv = open;
 GVT2 := q.is_empty = 0 & q.Q[1] = gv_open & gv = closed;
 GVT3 := q.is_empty = 0 & q.Q[1] = gv_open & gv = open;

ASSIGN
 init(gv) := closed;
 next(gv):=
  case
    GVT0 | GVT1 :  closed;
    GVT2 | GVT3 :  open;
   1:  gv;
  esac;
```

```
MODULE Ignitor(q)
VAR
 ig :  {off, on};

DEFINE
 IGT0 := q.is_empty = 0 & q.Q[1] = ig_close & ig = off;
 IGT1 := q.is_empty = 0 & q.Q[1] = ig_close & av = on;
 IGT2 := q.is_empty = 0 & q.Q[1] = ig_open & ig = off;
 IGT3 := q.is_empty = 0 & q.Q[1] = ig_open & ig = on;

ASSIGN
 init(ig) := off;
 next(ig):=
  case
    IGT0 | IGT1 :  off;
    IGT2 | IGT3 :  on;
   1:  ig;
  esac;
```

Linking the modules

All of the component modules are linked in the main module as in the coarse-grain and parameterised with the event queue module. If the system consists of more than one actuator and they contain transitions with guards referring to the states of other actuator modules, then the actuator module is parameterised with the respective actuator modules that it refers to. Similarly for subcontrollers.

```
MODULE main
VAR
 Cont :  Controller(Event_queue);
 SubCont :  SubController(Event_queue);
 Act :  Actuator(Event_queue);
 Event_queue :  Queue(Cont, Act);
```

The gas burner modules are defined in the main module and all of the components have read-only access to the event queue module. The queue module has read-only access to all of the system components. In SMV, parameters increase the state space but the clarity induced by modularisation overrides this disadvantage. For very large systems, the modularity can be collapsed into a single module.

```
MODULE main
VAR
   C : Controller(Qu);
   Av :  AirValve(Qu);
   Gv :  GasValve(Qu);
   Ig :  Ignitor(Qu);
   Qu :  Queue(C, Av, Gv, Ig);
```

5.1.3 Translation of the invariants

All of the invariants that were verified for the coarse-grain view of the system, must also be verified for the fine-grain view, that is they must hold for all fine-grain states, not only for stable states. Their form is identical to that of the invariants for the coarse-grain view and are translated as follows:

Rule 15: *For each temporal invariant T:*
```
SPEC
 AG(AG(Env) -> T)
```
where Env is the environmental assumptions.

Stable states are identified in the fine-grain as those states where *Event_queue.is_empty = 1* is true, and thus we can use this status variable to define the properties that hold in the coarse-grain, in the fine-grain model.

The invariants for the fine-grain view of the gas burner system consist of the same set of invariants as for the coarse-grain view.

```
MODULE main
 ...
SPEC
 AG(Gv.gv = open -> Av.av = open)
SPEC
 AG(Ig.ig = on -> Gv.gv = open)
```

Two additional properties that should be proven for a fine-grain system are:

1. That the system responds to every waiting event (if the event queue is non-empty, then the event at its head must eventually be processed.) This is expressed for every possible event that occurs at the head of the queue as:

$$AG(\text{Event\_queue.is\_empty} \ = \ 0 \ \& \ \text{Event\_queue.Q[1]} \ = \ e \ \rightarrow \ AF(\text{Tr1 / Tr2 ... / Trn}))$$

   where $Tr1, Tr2, ..., Trn$ are all the possible transitions for **e** that can be taken in a system.

2. Eventually all events in the event queue will be processed: this is a stronger property and is expressed in SMV as:

$$AG(\text{Event\_queue.is\_empty} \ = \ 0 \ \rightarrow \ AF \ \text{Event\_queue.is\_empty} \ = \ 1)$$

   and is verified for each specification.

5.1.4 Interpreting the results of SMV

The results produced when model checking the SMV model generated for the fine-grain view of an RSDS specification are similar to those produced for the SMV model generated for the coarse-grain view of an RSDS specification, i.e. true if a property holds in the model and, false and a counter-example otherwise. Once again, it is clear from the construction of the SMV from RSDS, that any counter-example trace in RSDS ($\neg P$) can be translated into a counter-example in SMV ($\xi(\neg P)$ where ξ represents the translation). This means that P holds in the SMV model, it also holds in the RSDS model (contra-positive).

The counter-example for the SMV model of a fine-grain view of an RSDS specification consists of a sequence of steps that corresponds to a sequence of fine-grain steps. For each step, a list of all variables and array values that have changed are given. The variables can be directly mapped to elements in SRS statemachines representing component states, sensor events and the occurrence of transitions. The array corresponds to the event queue defined in the fine-grain semantics and the array values can be mapped to generated events.

For the gas burner system, a counter-example is produced for the false property

$$AG((Qu.Q[1] = gv\_open \ \& \ Qu.is\_empty = 0) \rightarrow Gv.gv = open)$$

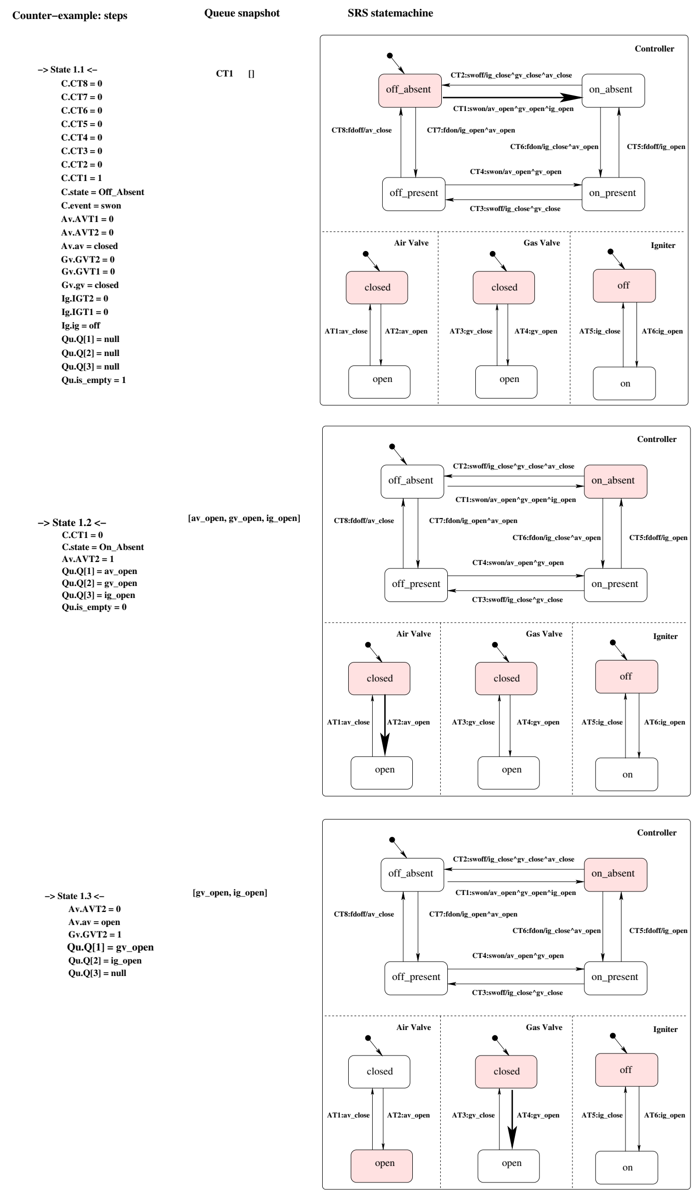
stating that it is always the case that if the event *gv_open* is in the front of the queue then the gas valve is open. This property is only true if the gas valve is open in the next step, i.e.

$$AG((Qu.Q[1] = gv\_open \ \& \ Qu.is\_empty = 0) \rightarrow AX(Gv.gv = open))$$

Figure 5.4 depicts the relationship between the elements in the counter-example with those in the SRS statemachines and event queue.

5.2 Proof of Correctness for the Fine-grain Translation

For the proof of correctness of the fine-grain translation we need to show that everything that is true in the fine-grain RSDS system theory Γ_{FSys}, is also true in the SMV interpretation of that theory $\Lambda_{M(FSys)}$ (soundness). In order to show this, we need to first prove the following lemma.


```
Counter–example: steps          Queue snapshot

–> State 1.1 <–                 CT1    []
     C.CT8 = 0
     C.CT7 = 0
     C.CT6 = 0
     C.CT5 = 0
     C.CT4 = 0
     C.CT3 = 0
     C.CT2 = 0
     C.CT1 = 1
     C.state = Off_Absent
     C.event = swon
     Av.AVT1 = 0
     Av.AVT2 = 0
     Av.av = closed
     Gv.GVT2 = 0
     Gv.GVT1 = 0
     Gv.gv = closed
     Ig.IGT2 = 0
     Ig.IGT1 = 0
     Ig.ig = off
     Qu.Q[1] = null
     Qu.Q[2] = null
     Qu.Q[3] = null
     Qu.is_empty = 1

–> State 1.2 <–                 [av_open, gv_open, ig_open]
     C.CT1 = 0
     C.state = On_Absent
     Av.AVT2 = 1
     Qu.Q[1] = av_open
     Qu.Q[2] = gv_open
     Qu.Q[3] = ig_open
     Qu.is_empty = 0

–> State 1.3 <–                 [gv_open, ig_open]
     Av.AVT2 = 0
     Av.av = open
     Gv.GVT2 = 1
     Qu.Q[1] = gv_open
     Qu.Q[2] = ig_open
     Qu.Q[3] = null
```

Figure 5.4: The counter-example produced when model checking fine-grain RSDS specification of the gas burner system.

Lemma 3. The axioms of Γ_{FSys} hold in $\Lambda_{M(FSys)}$ under the interpretation ξ of an RSDS system theory Γ_{FSys} as an SMV interpretation of that theory in $\Lambda_{M(FSys)}$.

$$\begin{tikzcd}
\text{FSys} \arrow[r, "M"] \arrow[d, "\Gamma"'] & \text{M(FSys)} \arrow[d, "\Lambda"] \\
\Gamma_{\text{FSys}} \arrow[r, "\xi"'] & \Lambda_{\text{M(FSys)}}
\end{tikzcd}$$

$\xrightarrow{\Gamma}$	Derive the meaning of RSDS specification	FSys	The fine−grain RSDS specification of a system
$\xrightarrow{M}$	Apply SMV translation	Γ_{FSys}	Interpretation of the theory of RSDS specification
$\xrightarrow{\Lambda}$	Derive the meaning of SMV specification	M(Sys)	SMV specification of a RSDS specification
$\xrightarrow{\xi}$	Preserve the meaning	$\Lambda_{\text{M(FSys)}}$	Interpretation of the theory of the SMV specification

Figure 5.5: A sketch of the proof of correctness.

Proof. The proof of the lemma is organised as follows:

- We first define the interpretation of a fine-grain RSDS system theory Γ_{FSys} into the theory $\Lambda_{M(FSys)}$.
- Then, we show that the component and system axioms hold from Γ_{FSys} in $\Lambda_{M(FSys)}$ under the interpretation as illustrated in Figure 5.5. These axioms are the same as in the coarse-grain semantics, i.e. CG1 to CG12, except for CG9 and CG10 that are dropped.
- Finally, we show that the axioms on *event_queue* hold from Γ_{FSys} in $\Lambda_{M(FSys)}$ under the interpretation.

The controller theory of Γ_{FSys} is interpreted by the module C in $\Lambda_{M(FSys)}$, each actuator theory by A, each subcontroller theory by SC and the queue attributes are interpreted in the Q module. The theory $\Lambda_{M(FSys)}$ of the interpretation of an RSDS specification **Sys** is defined to be the theory Λ_{main} of the main module of **M(FSys)**.

The interpretation of the fine-grain RSDS system theory Γ_{FSys} into the theory $\Lambda_{M(FSys)}$ is defined by:

1. *Q.event_queue : seq(EventToken)* for holding the sequence of tokens representing pending events is interpreted by *Q.qu:array 1..k of iTok* (an array in SMV) where *k* represents the integer of the maximum size of the array at any time during computation, and *iTok* is an enumerated set isomorphic to the set *EventToken* of all internal (generated) events in the system.
2. *Q.event_queue* = [] for checking whether the queue of internal events is empty, is interpreted by boolean variable *Q.is_empty:boolean*, where if *Q.is_empty = 1* the queue (array in SMV) is empty.
3. The sensor state s of a sensor S, is the i-th projection $proj_i(C.stable\_state)$ of the controller state, if S is S_i.

4. *tr : Trans*$_s$ for a sensor S is interpreted as the disjunction ($C.CT1$ = *1)* ∨ ... ∨ ($C.CTn$ = *1)* where the $CT1, ..., CTn$ represent all the controller transitions which depend on *tr* (in the RSDS model). We will use the shorthand given in SMV for $C.CT1$ to mean $C.CT1 = 1$, similarly for all boolean or DEFINE variables.
5. The state attribute *st* for the controller is interpreted by *C.stable_state*.
6. A transition *tr : Trans* of the controller is interpreted by the predicate *C.CTj=1* where *C.CTj* is the corresponding DEFINE variable in the controller module.
7. The sensor (external) events *ext : Ext_events* are interpreted by *C.sensor_event* = ext_{Tok} in the main controller where ext_{Tok} is the name for *ext* listed in the enumerated type of *C.sensor_event*.
8. The state attribute *sub_st* of a subcontroller is interpreted by *SC.sub_st*.
9. A transition *sub_tr* of a subcontroller is interpreted by the predicate $SC.Tk$ *=1* where $SC.Tk$ is the DEFINE variable in the SMV module for the subcontroller.
10. The state attribute *state* of an actuator is interpreted by *A.state*.
11. A transition *act_tr* of an actuator is interpreted by the predicate *A.ATj=1* where *A.ATj* is the DEFINE variable in the SMV module for the actuator.
12. The guard g of an actuator transition is interpreted by the predicate $G = 1$, which is described in the definitions of actuator transitions $A.ATi$. It is composed of a conjunction of references to other actuator states, for example : *A2.state = g1 & A3.state = g2*, which refers to the state of actuator $A2$ and $A3$.
13. A transition *tr* that changes the elements in the queue is interpreted as the disjunction $C.CT1 = 1 \ \vee \ ... \ \vee \ C.CTj = 1 \ \vee \ SC.T1 = 1 \ \vee \ ... \ \vee \ SC.Tk = 1 \ \vee \ A.AT1 = 1 \ \vee \ ... \ \vee \ A.ATq = 1$ where $C.T1, ..., C.CTj$ and $SC.T1, ..., SC.Tk$ and $A.AT1, ..., A.ATq$ represent all the transitions (controller, subcontroller and actuator) which invoke *tr* in the RSDS model.

The component and system axioms of Γ_{FSys} hold in $\Lambda_{M(FSys)}$ under this interpretation:

Controller

CG1 The axiom defining the initial state of Sm:

$$BEG \ \Rightarrow \ sm = init_{sm}$$

where Sm is a controller statemachine and sm is its current state, is interpreted in $\Lambda_{M(FSys)}$ as

$$BEG \ \Rightarrow \ C.stable\_state = init_{stable\_state}$$

which is immediately true. *C.stable_state* is a tuple of sensor states, which means that its initial state corresponds to the initial values of all the sensor modules $init_{S1}\_...\_init_{Sn}$, where $S1, .., Sn$ are the sensor modules of a system.

CG2 That at most one sensor event of Sm can occur in a step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of distinct events of Sm. This axiom is interpreted as

$$\neg(C.sensor\_event = \alpha \;\wedge\; C.sensor\_event = \alpha')$$

which holds because of the definition of sensor event: *C.sensor_event :Type*, where Type is an enumerated set, and *C.sensor_event* can have only one value at a time.

CG3 The state transition behaviour of Sm:

$$sm = s \;\wedge\; \alpha \;\Rightarrow\; tr \tag{5.1}$$

$$tr \Rightarrow \alpha \tag{5.2}$$

$$sm = s \;\wedge\; tr \;\Rightarrow\; AX(sm = t) \tag{5.3}$$

for each sensor transition tr of Sm with source s, target t and trigger event α. Sensor transitions are interpreted as the disjunction of controller transitions. Therefore, axiom 5.1 is interpreted as

$$(C.stable\_state = s1 \vee ... \vee C.stable\_state = sn) \wedge C.sensor\_event = \alpha \;\Rightarrow\; C.CT1 \vee ... \vee C.CTn$$

were $s1, ..., sn$ are all the controller states where one of the elements in the tuple is the same i.e. corresponds to the sensor source of the sensor transition, and $C.CT1, ..., C.CTn$ refers to the controller transitions that are depended on tr. This axiom holds because of the definition of controller transitions:

$$C.CTj := C.stable\_state = si \;\&\; C.sensor\_event = \alpha \;\&\; Q.is\_empty = 1$$

which are in terms of $C.stable\_state$ values, sensor events and the $event_queue$ is empty (i.e. no internal events).

The axiom 5.2 is interpreted as

$$C.CT1 \vee ... \vee C.CTn \;\Rightarrow\; C.sensor\_event = \alpha$$

which holds, because of the definitions of the controller transitions.

The axiom 5.3 is interpreted as:

$$(C.stable\_state = s1 \vee ... \vee C.stable\_state = sn) \wedge (C.CT1 \vee .... \vee C.CTn) \Rightarrow$$
$$AX(C.stable\_state = t1 \vee ... \vee C.stable\_state = tn)$$

whereby $s1, ..., sn$ are the controller state values that contain the sensor state s in their tuple, and $t1, ..., tn$ are the controller state values that contain t in their tuple. This axiom follows from the definition of *next* for $C.stable\_state$:

```
ASSIGN
  next(stable_state):=
  case
  CT1 :  t1;
   ...
  CTk :  tn;
  1:  stable_state;
  esac
```

which is interpreted as:

$$AG(C.stable\_state = s1 \;\Rightarrow\; (C.CT1 \;\Rightarrow\; AX(C.stable\_state = t1)))$$
$$\vdots$$
$$AG(C.stable\_state = s1 \;\Rightarrow\; (\neg C.CT1 \wedge \neg C.CT2 \wedge ... \wedge C.CTk \;\Rightarrow\; AX(C.stable\_state = tn)))$$

and similarly for all values of $C.stable\_state$.

CG4 That at most one sensor transition of Sm can occur in the step:

$$\neg(tr \wedge tr')$$

for each pair of distinct transitions of Sm. This axiom is interpreted as:

$$\neg((C.CT1 \vee ... \vee C.CTn) \wedge (C.CTj \vee ... \vee C.CTk))$$

where $(C.CT1 \vee ... \vee C.CTn)$ are the controller transitions invoked by tr, and $(C.CTj \vee ... \vee C.CTk)$ are the controller transitions invoked by tr'. An assumption of RSDS specifications is that the events are unique, and each sensor transition is defined in terms of events. Therefore, this axiom is true because each sensor transition will invoke a different set of controller transitions depending on the sensor event.

CG5 That a sensor transition can only occur if Sm is in its source state:

$$tr \Rightarrow sm = source_{Sm}(tr)$$

This axiom is interpreted as:

$$C.CT1 \vee ... \vee C.CTn \Rightarrow C.stable\_state = s1 \vee ... \vee C.stable\_state = sn$$

where the source state is an element in the tuple of controller states $s1, ..., sn$. Since the controller state is an amalgamation of sensor states, a sensor transition can correspond to a number of controller transitions (maximum number = number of sensor modules) because although the event and the source state are the same, the states of the other sensors can be different. This axiom is true because of the definition of controller transitions:

$$C.CTi := C.stable\_state = s \;\&\; C.sensor\_event = \alpha \;\&\; Q.is\_empty = 1;$$

which is in terms of the source state for $C.stable\_stable$.

CG6 The locality notion requires that there is no visible change to the value of the attribute sm when no transition is taken:

$$\neg tr_1 \;\wedge ... \wedge\; \neg tr_n \wedge sm = s \Rightarrow AX(sm = s)$$

for each $s \in States_{Sm}$, where the tr_i are all the transition action symbols of Sm.

This axiom is interpreted as:

$$\neg C.CT1 \wedge ... \wedge \neg C.CTk \wedge C.stable\_state = s \Rightarrow AX(C.stable\_state = s)$$

which is true because of the default case in the case statement within the *next* clause that defines the state changes of *C.stable_state*:

```
ASSIGN
  next(stable_state):=
  case
  CT1 :  s2;
   ...
  CTk :  sk;
  1:  stable_state;
  esac
```

which is interpreted as:

$$AG(C.stable\_state = s_0 \;\Rightarrow\; (\neg C.CT1 \wedge ... \wedge \neg C.CTk \;\Rightarrow\; AX(C.stable\_state = s_0)))$$

and similarly for all values of *C.stable_state*.

CG7 We assume that only one sensor event can occur in each step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of events α of Sm and α' of Sm' where Sm and Sm' are distinct sensor components. This axiom is interpreted as:

$$\neg(C.sensor\_event = \alpha \wedge C.sensor\_event = \alpha') \wedge Q.is\_empty = 1$$

which holds because the type of *C.sensor_event* is an enumerated set, and by definition only one value of this set can be true in each SMV step. The start of a fine-grain reaction cycle is when the event queue is empty and that is the only time that a sensor event can occur.

Subcontrollers/Actuators

CG1 The initial state of the actuator statemachine Sm:

$$\mathbf{BEG} \;\Rightarrow\; sm = init_{Sm}$$

where $init_{Sm}$ is the initial state of Sm. This axiom is interpreted as

$$\mathbf{BEG} \;\Rightarrow\; A.state = init_{Sm}$$

which is immediately true.

CG2 That at most one actuator event of Sm can occur in a step:

$$\neg(\alpha \wedge \alpha')$$

for each pair of distinct events of Sm. Actuator events are those that are at the head of the queue, therefore, the axiom is interpreted as:

$$\neg(Q.qu(1) = \alpha \wedge Q.qu(1) = \alpha')$$

which is immediately true because of definition of queue as an array that only has a single value at the first position (i.e. qu(1)).

CG4 That at most one actuator transition of Sm can occur in a step:

$$\neg(tr \wedge tr')$$

for each pair of distinct transitions of Sm. This axiom is interpreted as:

$$\neg(A.AT1 \wedge A.AT2)$$

for each pair of distinct transitions of actuator Sm. This axiom holds because of actuator definitions which state that actuator transitions are triggered by an event at the head of the queue:

$$ATi := A.state = a1 \;\&\; Q.qu[1] = \alpha \;\&\; G;$$

where G is the guarding condition whose default value is true. Since there is only one such event at a time, and the definition of actuator transitions is unique (combination of sctuator state and event), therefore, only one such transition is triggered at a time.

CG5 That a transition can only occur if Sm is in its source state:

$$tr \Rightarrow sm = source_{Sm}(tr)$$

This axiom is interpreted as:

$$A.ATi \Rightarrow A.state = a1$$

where $a1$ is the source state of transition $A.ATi$ and is immediately true because of the definition of an actuator transition:

$$ATi := A.state = a1 \;\&\; Q.qu[1] = \alpha \;\&\; G;$$

where G is the guarding condition whose default value is true.

CG6 The locality notion requires that there is no visible change to the value of the attribute sm when no transition is taken:

$$\neg tr_1 \;\wedge ... \wedge\; \neg tr_n \wedge sm = s \Rightarrow AX(sm = s)$$

for each $s \in States_{Sm}$, where the tr_i are all the transition action symbols of Sm. This axiom is interpreted as:

$$\neg A.AT1 \wedge \neg A.AT2 \wedge ... \wedge \neg A.ATn \wedge A.state = s \Rightarrow AX(A.state = s)$$

This is represented in the translation by the default case in the case statement within the *next* clause that defines the state changes to $A.state$, the actuator state:

```
ASSIGN
  next(state):=
  case
  AT1 :  a2;
   ...
  ATn :  ak;
  1:  state;
  esac
```

which is interpreted as:

$$AG(A.state = a1 \;\Rightarrow\; (\neg A.AT1 \wedge ... \wedge \neg A.ATn \;\Rightarrow\; AX(A.state = a1)))$$

which is immediately true. Similarly for all values of $A.state$.

CG8 The state transition behaviour of Asm:

$$sm = s \;\wedge\; \alpha \;\wedge\; g_{tr} \Rightarrow\; tr \tag{5.4}$$

$$tr \Rightarrow \alpha \;\wedge\; g_{tr} \tag{5.5}$$

$$sm = s \;\wedge\; tr \;\Rightarrow\; AX(sm = t) \tag{5.6}$$

for each transition tr of Sm with source s, target t and trigger event α. The first axiom is interpreted as:

$$A.state = s \wedge Q.qu(1) = \alpha \wedge G \;\Rightarrow\; A.ATi$$

where G is the guard. This axiom is immediately true because of the definition of the actuator transition which is given in terms of: the event at the head of the queue (element at array index 1), the actuator state and the guard.

Axiom 5.5 is interpreted as:

$$A.ATi \Rightarrow Q.qu(1) = \alpha \wedge G$$

which is immediately true because of the definition of the actuator transition.

The final axiom 5.6 is true because of the *next* assignment for the actuator state:

```
ASSIGN
  next(state):=
  case
  AT1 :  a2;
   ...
  ATn :  an;
  1:  state;
  esac
```

which is interpreted as:

$$\begin{aligned}
&AG(A.state = a1 \;\Rightarrow\\
&\qquad (A.AT1 \;\Rightarrow\; AX(A.state = a2)) \;\wedge\\
&\qquad (\neg\; A.AT1 \;\wedge\; A.AT2 \;\Rightarrow\; AX(A.state = a3)) \;\wedge\\
&\qquad \vdots\\
&\qquad (\neg\; A.AT1 \;\wedge\; \dots \;\wedge A.ATn \;\Rightarrow\; AX(A.state = an))))\\
&\qquad (\neg\; A.AT1 \;\wedge\; \dots \;\wedge \neg A.ATn \;\Rightarrow\; AX(A.state = a1))))
\end{aligned}$$

showing that the cases when an actuator transition occurs (all, except for the last one) brings about a change to the actuator state.

Complete System

CG11 An axiom for the guard is :

$$tr_C \;\wedge\; G_A \Rightarrow\; tr_A$$

where G_A is the guard for an actuator transition, and tr_C and tr_A are controller and actuator transitions respectively. This axiom is interpreted as:

$$C.CTi \wedge G \Rightarrow A.ATi$$

If we assume that $C.CTi \wedge G$, then because of the definition of $A.ATi$, we can reduce the axiom to:

$$C.CTi \Rightarrow Q.qu(1) = \alpha \wedge A.state = a1$$

where α is the event that triggers $A.ATi$ and $a1$ is the state in which $A.ATi$ occurs. The guard definitions on either sides of the implication are cancelled out. Let's assume that $C.CTi$ generates events, whereby α is one of them. Then, because of the *next* definition for $Q.qu(1)$ which is interpreted as:

$$AG(Q.qu(1) = x \Rightarrow (C.CTi \Rightarrow AX(Q.qu(1) = \alpha)))$$

the CG11 axiom holds. If a controller transition does not generate events, CG11 does not hold.

CG12 The axiom that asserts that each controller transition generates events received by subcontroller or actuator components:

$$AG(tr_C \;\Rightarrow\; \rho_1 \wedge \rho_2 \wedge ... \wedge \rho_p)$$

where tr_C is a controller transition and $\{\rho_1, ..., \rho_p\}$ are the subcontroller or actuator events that it generates. This axiom is interpreted as:

$$AG(C.CTi \Rightarrow Q.qu(1) = \rho_1 \wedge Q.qu(2) = \rho_2 \wedge ... \wedge Q.qu(p) = \rho_p$$

which is immediately true because of the *next* definitions for each element in the queue (Q.qu(i)). For example, the interpretation of the *next* definition for $Q.qu(1)$ is:

$$AG(Q.qu(1) = x \Rightarrow (C.CTi \Rightarrow AX(Q.qu(1) = \rho_1)))$$

where $x \in Type$ of generated events. Similarly for all possible values of $Q.qu(1)$ and similarly for all p positions in the array.

The axioms on *event_queue* of Γ_{FSys} hold in $\Lambda_{M(FSys)}$ under this interpretation:

FG1 The initialisation axiom:

$$BEG \;\Rightarrow\; event\_queue = []$$

is interpreted in two ways:

1. $BEG \Rightarrow Q.is\_empty = 1$ which is immediately true.
2. $$\begin{aligned} BEG &\Rightarrow Q.qu(1) = null \\ BEG &\Rightarrow Q.qu(2) = null \\ &\vdots \\ BEG &\Rightarrow Q.qu(n) = null \end{aligned}$$

   for each position in the $Q.qu$ (n = last index of array). All the elements in the array must be *null* for the queue to be empty, which is immediately true.

FG2 An sensor event $\alpha \in Ext_{FSys}$ that triggers a transition t may only be responded to if the queue is empty:

$$t \Rightarrow event\_queue = []$$

which is interpreted by:

1. $C.CTi \Rightarrow Q.is\_empty = 1$ which is immediately true because of the definition of controller transitions:

   $$C.CTi := C.stable\_state = s1 \;\&\; C.sensor\_event = \alpha \;\&\; Q.is\_empty = 1;$$

2. $$\begin{aligned} C.CTi &\Rightarrow Q.qu(1) = null \\ C.CTi &\Rightarrow Q.qu(2) = null \\ &\vdots \\ C.CTi &\Rightarrow Q.qu(n) = null \end{aligned}$$

   These axioms are immediately true because $C.CTi$ can only occur if the event queue is empty (definition of $C.CTi$), which means that all the elements in the array representing the queue are null.

FG3 When an sensor event α is received by the system and it triggers a sensor or controller transition t, its generations become the new event queue:

$$t \Rightarrow AX(event\_queue = generations(t))$$

which is interpreted by:

1. $C.CTi \Rightarrow AX(Q.is\_empty = 0)$ which holds because of the *next* definition which is interpreted as:

   $$AG(Q.is\_empty = 1 \Rightarrow (C.CTi \Rightarrow AX(Q.is\_empty = 0))$$

2. $$\begin{aligned} C.CTi &\Rightarrow AX(Q.qu(1) = \rho_1) \\ C.CTi &\Rightarrow AX(Q.qu(2) = \rho_2) \\ &\vdots \\ C.CTi &\Rightarrow AX(Q.qu(n) = \rho_n) \end{aligned}$$

   These are true becuase of the next definition for each element in the queue (represented by array). For example, for the first element in the queue, the next clause is defined as follows:

   ```
   ASSIGN
     next(qu[1]):=
     case
     CTi :  ρ1;
      ...
     1:  qu[1];
     esac
   ```

   which is interpreted as:

   $$AG(Q.qu(1) = null \Rightarrow (C.CTi \Rightarrow AX(Q.qu(1) = \rho_1))$$

   which is immediately true. Similar interpretations for all possible values of $Q.qu(1)$ (i.e. on the LHS of the first implication). Similarly for all array indexes.

FG4 An internal event found in the queue can only be processed if it is at the head of the queue:

$$\beta \Rightarrow event\_queue \neq [] \wedge name(\beta) = head(event\_queue)$$

where β is the event at the head of the queue and *name* is defined above. This axiom is intepreted as:

$$A.ATi \Rightarrow Q.is\_empty = 0 \wedge Q.qu(1) = \beta$$

where $A.ATi$ is some actuator or subcontroller transition which is triggered by event β, which is at the head of the queue (that is how events are processed). This axiom is immediately true because of the definition of the actuator/subcontroller transitions :

$$ATi \;:=\; Q.is\_empty \;=\; 0 \;\&\; Q.qu[1] \;=\; \beta \;\&\; A.state \;=\; a1;$$

FG5 The head of the queue is removed when the corresponding event is processed and replaced with some interleaving ρ of the generations (set of internal events) of all the transitions triggered by the occurrences of β, the event at the head of the queue:

$$\beta \ \wedge \ q = event\_queue \Rightarrow AX(event\_queue = \rho \frown tail(q))$$

If there are no generations ($\rho \ = \ []$), then the event β at the head of the queue is simply removed. This axiom is interpreted by:

1.

$$\begin{aligned} \beta \quad \wedge \quad & (Q.qu(2)' = Q.qu(2) \wedge ... \wedge Q.qu(k)' = Q.qu(k)) \Rightarrow \\ & AX((Q.qu(1) = \rho_1 \wedge ... \wedge Q.qu(n) = \rho_n) \wedge \\ & (Q.qu(n+1) = Q.qu(2)' \wedge ... \wedge Q.qu(max) = Q.qu(k)') \end{aligned} \tag{5.7}$$

where $Q.qu(1)', ..., Q.qu(k)'$ refer to the current values of the array, k refers to the number of generated events in the queue and n is the number of generations that are added at the head of the queue. There are two cases to consider:

(a) The case where β is a sensor (external) event: the queue is empty when a sensor event occurs, therefore the generated events are added to the queue. This is interpreted as:

$$C.sensor\_event = \beta \Rightarrow AX(Q.qu(1) = \rho_1 \wedge ... \wedge Q.qu(n) = \rho_n)$$

which holds because the sensor event will trigger a controller transition that generates the events $\rho_1, .., \rho_n$. These events are added to the queue by the definition of *next* for the array elements representing the queue. For example, for $Q.qu(1)$ the *next* definition is interpreted as:

$$AG(Q.qu(1) = x \Rightarrow (C.CTi \Rightarrow (AX(Q.qu(1) = \rho_1)))$$

and similarly for all possible values of $Q.qu(1)$. Similarly for all array indexes.

(b) The case where β is a generated event: the generated event triggers a subcontroller or actuator transition that could generate more events to be placed in the queue. In the axiom definition 5.7 for the interpretation of FG5, β is changed to $A.ATi$ which is some actuator transition that is triggered by β, which is how events are processed in SMV. This axiom holds because of the *next* definitions for the array elements and for $Q.is\_empty$.

i. The status variable for the queue must indicate that once the transition $A.ATi$ occurs, the queue is not empty (because there are generated events) which follows from the *next* definition that is interpreted as:

$$AG(is\_empty = 1 \Rightarrow (A.ATi \Rightarrow AX(is\_empty = 0)))$$

Similarly for when $is\_empty = 0$.

ii. Each array element, representing the queue elements, must be updated to show that the generations are added into the queue. If there are n generations (that are less than the size of the array) with values $\rho_1, ..., \rho_n$, then for positions 1 to n in the array, *next* is defined as:

```
ASSIGN
  next(qu[1]):=
  case
  A.ATi :  ρ1;
   ...
  1:  qu[1];
  esac
```

which is interpreted as:

$$AG(Q.qu(1) = x \Rightarrow (A.ATi \Rightarrow AX(Q.qu(1) = \rho_1)))$$

Similarly for all values of $Q.qu(1)$ and similarly for all array positions with indexes 1 to n.

The events that are already in the queue are shifted down the queue. For example, if n generations are added to the queue, then the existing events are placed in positions $n+1$ to max, where max is the size of the array. The *next* definitions for array positions are defined as:

```
ASSIGN
  next(qu[i + n - 1]):=
  case
  A.ATi :  qu[i];
   ...
  1:  qu[i + n - 1];
  esac
```

where i is the array index of the event in its original position, n is the number of generated events. This is interpreted as:

$$AG(Q.qu(i+n-1) = x \Rightarrow (A.ATi \Rightarrow AX(Q.qu(i+n-1) = Q.qu(i))))$$

where $i > 1$ because the first element in the queue is removed. Similarly for all values of $Q.qu(i+n-1)$ and similarly for all array indexes greater than n.

2. If there are no generations then the interpretation is as follows:

$$\beta\wedge \Rightarrow AX(Q.qu(1) = Q.qu(2) \wedge Q.qu(2) = Q.qu(3) \wedge ... \wedge Q.qu(n) = null)$$

The event β is an actuator event (usually the latter) which will trigger a transition that will process it. Lets call this transitions XTi that can be either a subcontroller or actuator transition. This holds because of the *next* definitions for $Q.is\_empty$ and for the queue elements.

(a) The status variable $Q.is\_empty$ is updated (using the *next* clause) when XTi occurs:

$$AG(Q.is\_empty = 1 \Rightarrow (XTi \Rightarrow AX(Q.is\_empty = 0)))$$

Similarly for when $Q.is\_empty = 0$. If β is the last element in the queue, then the queue is empty, which is interpreted as:

$$AG(Q.qu.is\_empty = 0 \wedge Q.qu(2) = null \Rightarrow (XTi \Rightarrow AX(Q.is\_empty = 1)))$$

which checks to see that the second element in the queue is null, then in the next step the queue will be empty (the event at the head of the queue is processed).

(b) The *next* definition for the queue elements is:

```
ASSIGN
  next(qu[i]):=
  case
  XTi :  qu[i+1];
   ...
  1:  qu[i];
  esac
```

which is interpreted as:

$$AG(Q.qu(1) = x \Rightarrow (XTi \Rightarrow AX(Q.qu(1) = Q.qu(2)))$$

and similarly for all possible elements of $Q.qu(1)$. Similar interpretations for all elements in the array.

The two properties that should be proven for a system are (asserted as properties under the SPEC clause in SMV):

FG6 That the system responds to every waiting event (if the event queue is non-empty, the event at its head must eventually be processed.) This is a weaker property than FG7 and is expressed for every possible event that occurs at the head of the queue as:

$$AG(Event\_queue.is\_empty\ =\ 0\ \&\ Event\_queue.Q[1]\ =\ e\ \rightarrow\ AF(Tr1\ /\ Tr2\ ...\ /\ Trn))$$

where $Tr1, Tr2, ..., Trn$ are all the possible transitions that can be taken in a system. Therefore, there are many properties to be verified.

FG7 Eventually all events in the event queue will be processed: this is expressed in SMV as:

$$AG(Event\_queue.is\_empty\ =\ 0\ \rightarrow\ AF\ Event\_queue.is\_empty\ =\ 1)$$

and is verified for each specification automatically.

- All system invariants are valid in the RSDS controller module, so remain true in the SMV translation.

□

Theorem 3. *The translation is sound if we can show that: if* $\Gamma_{FSys} \vdash \varphi$ *then* $\Lambda_{M(FSys)} \vdash \xi(\varphi)$

Proof. The theorem follows because the rules of deduction are the same in the theories (quantifier free CTL in object calculus) and deduction steps [Eme90] in Γ_{FSys} are preserved by ξ. For example, for Modus Ponens:

If

$\Gamma_{FSys} \vdash \varphi$

and

$\Gamma_{FSys} \vdash (\varphi \implies \psi)$

then by induction on the length of the proof

$\Lambda_{M(FSys)} \vdash \xi(\varphi)$

and

$\Lambda_{M(FSys)} \vdash \xi(\varphi \implies \psi)$

which means

$\Lambda_{M(FSys)} \vdash \xi(\varphi) \implies \xi(\psi)$

so

$\Lambda_{M(FSys)} \vdash \xi(\psi)$

□

Therefore, with this theorem we proved the soundness of the translation. In order to show the completeness of the translation, we use the following lemma.

Lemma 4. For each trace h of the SMV model $M(Sys)$ there is a corresponding trace $r(h)$ in the RSDS model Sys which satisfies the same formulae with respect to the translation ξ.

Proof. If h consists of a sequence $h_1, ..., h_n$ of n Kripke nodes (where RSDS states and events are represented by variables), then $r(h)$ is defined as a sequence of n states and events as follows.

The settings of the sensors at state i of $r(h)$ are given by the value of *C.stable_state* in h_i. The settings of other components are given by their values in h_i. The sensor event which occurs to move from node i to node $i+1$ is given by *C.sensor_event* in h_i. The response actuator events and transitions are given by those disjunctions of conditions used in $M(Sys)$ to define their occurrence, which are true (i.e. their value is 1) in h_i. The settings of the queue are given by *Q.qu:array 1..max* and *Q.is_empty* in h_i

These two traces have the property that:

$h \vdash_{\Lambda_{M(Sys)}} \xi(\varphi) \Leftrightarrow r(h) \vdash_{\Gamma_{Sys}} \varphi$ □

Hence, if h is an SMV counter-example to an RSDS property ψ, i.e.

$h \vdash_{\Lambda_{M(Sys)}} \neg\xi(\psi)$

the $r(h)$ is an RSDS counter-example to ψ:

$r(h) \vdash_{\Gamma_{Sys}} \neg(\psi)$

Let us consider the same example system described in the completeness proof for the coarse-grain view 4.4 that contains: two sensors, $S1$ with states $\{s11, s12\}$, and $S2$ with states $\{s21, s22, s23\}$; a controller C and an actuator $A1$ with states $\{a1, a2, a3\}$. If

$$\varphi = C.CT1\ \&\ Q.is\_empty = 1\ \Rightarrow\ AF(A.state = a3)$$

then a possible counter-example h produced for this system could be:

h_1 $C.stable\_state = s11\_s21$
$C.sensor\_event = \alpha$
$A.state = a1$
$C.CT1 = 1$
(No actuator transitions are true)
$Q.is\_empty = 1$
$Q.qu[1] = null$
$\vdots$
$Q.qu[n] = null$ where n is the size of the array

h_2 $C.stable\_state = s12\_s21$
$C.sensor\_event = \alpha$
$A.state = a1$
(No controller transitions are true)
$A.AT1 = 1$
$Q.is\_empty = 0$
$Q.qu[1] = \rho_1$
$Q.qu[2] = null$
$\vdots$
$Q.qu[n] = null$

h_3 $C.stable\_state = s12\_s21$
$C.sensor\_event = \beta$
$A.state = a2$
$C.CT2 = 1$
(No actuator transitions are true)
$Q.is\_empty = 1$
$Q.qu[1] = null$
$\vdots$
$Q.qu[n] = null$

where in h_3 $A.state = a2$ instead of $A.state = a3$. The corresponding counter-example $r(h)$ in the RSDS model Sys is:

$r(h_1)$ $S1 = s11$
$S2 = s21$
$stable\_state = s11\_s21$
$sensor\_event = \alpha$
$A1 = a1$
$tr_C = \alpha/\rho_1$
(No actuator transitions are true)
$event\_queue = []$

$r(h_2)$ $S1 = s12$
$S2 = s21$
$stable\_state = s12\_s21$
(No sensor event occurs)
$A1 = a1$
(No controller transition occurs)
$tr_A = \rho_1$
$event\_queue(1) = \rho_1$

$r(h_3)$ $S1 = s12$
$S2 = s21$
$stable\_state = s12\_s21$
$sensor\_event = \beta$
$A1 = a2$
$tr_C = \beta/\rho_2$
(No actuator transitions are true)
$event\_queue = []$

This example illustrates the close correspondence between h and $r(h)$.

The same reasoning as in 6 also shows the completeness of the translation.

Theorem 4. *The translation is complete if we can show that: if* $\Lambda_{M(Sys)} \vdash \xi(\varphi)$ *then* $\Gamma_{Sys} \vdash \varphi$.

Proof. We prove the contra-positive: if $not(\Gamma_{Sys} \vdash \varphi)$ then $not(\Lambda_{M(Sys)} \vdash \xi\varphi)$ For every trace s of an RSDS system it is simple to construct a trace h of the corresponding SMV system for which $s = r(h)$. If

$not(\Gamma_{Sys} \vdash \varphi)$

there is a counter-example trace

$s \vdash_{\Gamma_{Sys}} \neg\varphi$

But then

$h \vdash_{\Lambda_{M(Sys)}} \neg(\xi(\varphi))$

and

$not(\Lambda_{M(Sys)} \vdash \xi\varphi)$ which proves completeness. □

5.3 Applying Decomposition Techniques to SMV Models

As for coarse-grain SMV models, decomposition techniques can be applied in a natural way to fine-grain SMV models to help reduce the state space for the purpose of model checking. In fact, the way in which these approaches are applied is very similar, that is, subsystems are defined as separate SMV programs so that model checking is performed locally. However, in order to discuss these, we need to first show how the fine-grain view of decomposed systems is translated into

SMV modules. Each decomposition approach is considered independently. In practice it is often the case that systems are decomposed using a combination of these, and thus a combination of translation rules must be applied, one for each approach.

We have already shown how subcontrollers are translated into SMV modules as these are similar to actuator components. Their transitions are triggered by internal events and, unlike actuator transitions, they may generate events that are added to the event queue. The introduction of subcontrollers in the SMV models of the fine-grain is useful as opposed to the coarse-grain, as the sequence of internal events generated, which includes those generated by the subcontrollers, in the fine-grain must be shown to be correct. In this section, we give for each decomposition technique any additional translation rules that are required and briefly describe how the algorithm at the pre-processing phase calculates the maximum size of the array to be used for the event queue.

5.3.1 Hierarchical composition of controllers

The hierarchical decomposition approach decomposes the system controller into a number of subcontrollers, managed by an overseer controller.

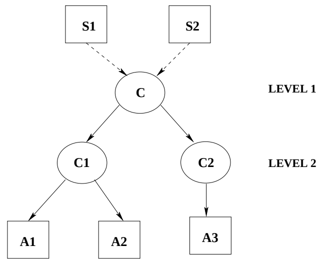


Figure 5.6: There are two levels of controllers in this hierarchically decomposed system.

The algorithm for calculating the maximum size of the array used to represent the event queue considers the events generated at each level. When a hierarchical approach is applied once to a system, as in Figure 5.6, two levels of controllers are produced.

In a hierarchically decomposed system, subsystems can be verified independently in separate SMV programs with the introduction of virtual sensors that act as interfaces between one subsystem and another. These are introduced in the same way as described in section 4.5.1 for the coarse-grain.

5.3.2 Horizontal composition of controllers

In a horizontally decomposed system, events are broadcast from the outer-level controller to separate controllers that processes events independently. There is no overseer controller for managing subcontrollers. Each subcontroller is considered as a separate subsystem. An example of such a system is illustrated in figure 5.7.

Each subsystem in a horizontally decomposed system can be expressed as a separate SMV

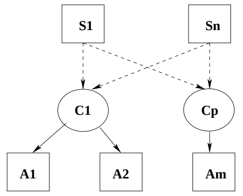


Figure 5.7: The DCFD for the horizontally decomposed system.

program and be model checked independently. This will thus help reduce the state space. Small or medium systems can be represented in a single SMV program. There are two cases that we need to consider when translating these into SMV:

1. The first case considers subcontrollers that process events concurrently. In this case, we introduce separate event queues for each subcontroller i.e. subsystem. Translation rule 15 shows how the separate queues are defined in the *main* module. Each queue module is generated using the translation rules 1-6. Translation rule 16 is used when we want to reduce the state space of large systems by avoiding the unnecessary parameter passing of modules, i.e. each queue defined for each system has parameters for all the system modules. Therefore, the separate queues are defined in the same queue module (as defined normally by translation rules 1-6) and can have access to all the system modules.

Rule 16: *For each subsystem SS1...SSn defined in the main module:*
```
VAR
 QSS1 :  QueueSS1(C, Act);
 ...
 QSSn :  QueueSSn(C, Act);
```
where n is the maximum number of subsystems,
C is the subsystem's controllers, and Act is the subsystem's actuators.

OR

Rule 17: *Define an array for each subsystem in the Queue module:*
```
VAR
 QSS1 :  array 1..maxSS1 of {s1e1, s1e2, ...,s1em, null};
 ...
 QSSn :  array 1..maxSSn of {sne1, sne2, ...,snek, null};

 is_emptySS1 :  boolean;
 ...
 is_emptySSn :  boolean;
```
where n is the number of subsystems in a system.

For a horizontally decomposed system consisting of $C_1, ..., C_p$ independent controllers, a separate event queue is generated for each controller $QSS_1, ..., QSS_p$ and each of these is represented by a separate module in SMV. The actuators are parameterised with the queue with which they interact, that is any event queue **i** of $QSS_1, ..., QSS_p$, where $1 \leq \mathbf{i} \geq p$. The *main* module is responsible for linking each controller with its corresponding event queue and defining which actuators belong to which subsystem i.e. to the event queue and controller. If the subcontrollers share actuators, then event queues for each controller will be updated by transitions of the shared actuators. The translation schema for the main module of this system is as follows.

```
MODULE main
VAR
 C1 :  Controller1(Event_queue);
  ...
 CP : ControllerP(Event_queue);
 Act :  Actuator(Event_queue);
 ...
 Event_queue :  Queue1(C1, ..., CP, Act);
```

The algorithm that calculates the maximum size of the event queue (array in SMV) is applied to each subsystem, that has its own queue.

2. Alternatively, the second case considers subcontrollers processing interleaved events. A strict order in which the events are processed is imposed over the entire system i.e. an event is processed from the one subcontroller and then an event is processed from another subcontroller and so on until all sensor events have been processed. Typically, this case is used for RSDS systems not optimised by decomposition. The queue is defined as normal using the translation rules discussed, and the internal events in the queue can be any internal event for any subcontroller.

5.3.3 Phase composition of controllers

For a system decomposed using the phase decomposition technique, a separate controller is specified for the control reactions to be carried out in each mode or phase of the system. Phase decomposition is similar to the hierarchical decomposition, that is it consists of a single overseeing controller with a number of subcontrollers. The key difference is that in the phase decomposition the system can react to events only in one mode at a time, therefore only one subcontroller can compute reactions to events at a time, while in the hierarchical decomposition this is not so. Each active mode corresponds to a separate subsystem, for example in Figure 5.8 there are three subsystems, where two controllers C2 and C3 share actuator A3.

We assume that there are restraints on the overseeing controller to send at each step events only to one subcontroller or mode as only one mode is active at a time (the activeness of subcontrollers is mutual exclusive). A subcontroller is active when it receives events that fire/enable its transitions, which in turn generate actuator events which update the actuator states of actuators that it manages. In other words, only one subcontroller can respond to sensor events at a time in this decomposition approach. An active mode is determined by a disjunction of sensor states, i.e. the state of the overseeing controller (in Figure 5.8 the overseeing controller is C). Subcontrollers can share actuators, however this does not lead to any safety violation as they cannot send contradictory commands simultaneously to the common actuator as the subcontrollers are active at different times.

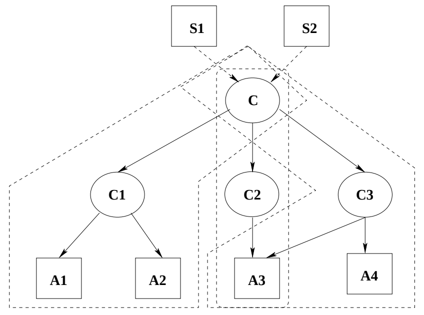


Figure 5.8: An example of a system with three modes (subsystems).

The SMV code for the controller module is generated by using the standard translation rules presented. The controller transitions determine which mode and thus which subcontroller is active as they are responsible for updating the sensor states ingrained in the controller state. A separate variable could be used to identify the system modes, but we choose to keep the state space to a minimum as we can determine the modes by the controller transitions and controller state.

The SMV modules for subcontrollers and actuators are similar and do not differ much from their standard module definition. The main difference lies in the definition of subcontroller or actuator transitions where they can only occur if the system is in the right mode. The translation rule below describes the definition of the transitions for these modules.

Rule 18: *For each subcontroller or actuator transition:*
```
DEFINE
 AT0 := C.CT1 & q.is_empty = 0 & q.Q[1] = ie1 & act = sc0;
```
where AT0 is a subcontroller or actuator transition, CT1 is a controller transition, act is the subcontroller or actuator state and ie1 is an internal event.

Actuators can be shared by different subcontrollers which means that they can receive contradictory commands from different subcontrollers as only one subcontroller is active at a time. Therefore, different system modes can have controller transitions that generate the same or different internal events to be received by the shared actuator. Lets assume that in Figure 5.8 subcontroller C2 is active with controller transitions CT1 and CT3 while C3 is active with controller transitions CT2 and CT4, but CT1, CT2 and CT4 generate the internal event a1 for actuator A3, while CT3 generates the internal event a2 for actuator A3. In the first mode the subcontrollers generate the same internal event a1, while in the second mode subcontroller C2 generates a different internal event. Both cases describe valid system behaviour.

The modules are linked in the usual way. Invariants for this decomposition technique have the standard form $AG(AG(Env) \Rightarrow T)$, but T is usually given in terms of modes:

$$C.stable\_state = st1 \ \Rightarrow \ Q$$

where $C.stable\_state$ is the controller state that determines which mode is currently active and Q describes constraints on the actuator states in that mode. For example, a possible mode of a system with two sensors $s1, s2$ (that have two states: on, off) could be when $s1 = on$. Therefore, this mode would be active if $C.stable\_state = on\_off$ and $C.stable\_state = on\_on$. Q would then be the constraints on actuator states for these controller states when $s1 = on$. T can also have any static or temporal invariant form presented in section 3.1.1.

The algorithm for calculating the maximum size of the array is applied to each subsystem and then the maximum length of alternate subcontrollers is calculated.

Systems structured according to the phase decomposition approach can be expressed using numerous SMV programs, one for each subsystem with an active controller. Therefore, properties of active controllers are verified independently. If subsystems depend on each other in any way, then virtual sensors must be introduced as in hierarchically decomposed systems (see example given for coarse-grain in section 4.5.3 as this is similar to the fine-grain).

5.3.4 Annealing

There are no clear benefits of using the annealing decomposition approach for a system that is to be translated into SMV. This is because the encapsulation of the repeated control sequences in a separate module will only increase the state space of the SMV model as all the modules will require read-access to the module and vice versa.

5.3.5 Standard controllers

There are two standard controllers: the AND controller and the priority controller. The fine-grain translation is similar to the coarse-grain with the inclusion of the event queue.

The AND controller system

The AND controller system in Figure 5.9 consists of two sensors, a controller and an actuator. The controller sets an actuator to *on* only if both sensors generate the event *swon*. The actuator state is updated analogous to the truth table produced for a logical *and* operator.

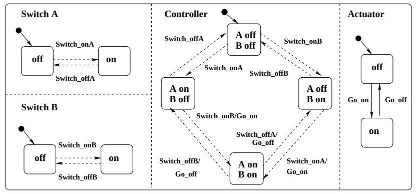


Figure 5.9: The SRS statemachine for the AND controller.

In the fine-grain SMV representation of the AND controller system, an event queue is introduced. However, since there is only one actuator in this system, an event queue is not required since there will be a maximum of one event generated at a time. Therefore, a variable q of enumerated type {*swon, swoff, null*} is used to represent the internal event at the head of the queue and no variable is needed to indicate whether the queue is empty. The AND controller system can be

part of a larger system with many actuators which would require an event queue to organise the order of the internal events issued. The queue module is generated using the standard translation rules.

The AND behaviour of the system is implemented by the generations of the internal event *swon*, i.e. only controller transition CT1 and CT5 generate the internal event *swon* which consequently sets the actuator state to *on*. The actuator transitions remove the event at the head of the queue, therefore in this case they simply set the variable q to null.

The SMV code generated for the queue module is as follows.

```
MODULE Queue(C, A)
VAR
 q :  {swonA, swoffA, swonB, swoffB, null};

ASSIGN
 init(q) := null;
 next(q):=
  case
   C.CT1 | C.CT5 :  swon;
   C.CT2 | C.CT3 | C.CT4 | C.CT6 | C.CT7 | C.CT8 :  swoff;
   A.AT1 | A.AT2 | A.AT3 | A.AT4 :  null;
   1:q;
  esac;
```

The controller module is generated using the standard translation rules. In this case, owing to the system having a single actuator and thus just a single variable for representing the internal event generated, the controller transition occurs only if no internal event is generated, i.e. $Q.q = null$. The controller state changes that the transitions cause correspond to updating the sensor states, since the controller state is an amalgamation of sensor states. The SMV code for the controller is generated as follows.

```
MODULE Controller(Q)
VAR
 stable_state :  {Aon_Boff, Aon_Bon, Aoff_Boff, Aoff_Bon};
 sensor_event :  {swonA, swonB, swoffA, swoffB };

DEFINE
 CT1 := Q.q = null & sensor_event = swonA & stable_state = Aoff_Bon;
 CT2 := Q.q = null & sensor_event = swonA & stable_state = Aoff_Boff;
 CT3 := Q.q = null & sensor_event = swoffB & stable_state = Aoff_Bon;
 CT4 := Q.q = null & sensor_event = swoffB & stable_state = Aon_Bon;
 CT5 := Q.q = null & sensor_event = swonB & stable_state = Aon_Boff;
 CT6 := Q.q = null & sensor_event = swonB & stable_state = Aoff_Boff;
 CT7 := Q.q = null & sensor_event = swoffA & stable_state = Aon_Bon;
 CT8 := Q.q = null & sensor_event = swoffA & stable_state = Aon_Boff;

ASSIGN
 init(stable_state) := Aoff_Boff;
 next(stable_state):=
  case
   CT1 | CT5 :  Aon_Bon;
```

```
    CT2 | CT4 :  Aon_Boff;
    CT3 :  Aoff_Boff;
    CT6 | CT7 :  Aoff_Bon;
    CT8 :  Aon_Boff;
    1:stable_state;
  esac;
```

The actuator module has read-only access to the queue module as its transitions are triggered by internal events found in the queue, in this case indicated by the variable *q*. The sole responsibility of the actuator module is to update the actuator state according to the commands issued by the controller. Therefore, the queue module determines from the controller transitions when the actuator state is set, and the actuator module sets its state accordingly. The SMV code generated for the actuator module is as follows.

```
MODULE Actuator(Q)
VAR
 act :  {on, off};

DEFINE
 AT1 := Q.q = swon & act = off;
 AT2 := Q.q = swon & act = on;
 AT3 := Q.q = swoff & act = off;
 AT4 := Q.q = swoff & act = on;

ASSIGN
 init(act) := off;
 next(act):=
  case
   AT1 | AT2 :  on;
   AT3 | AT4 :  off;
   1:act;
  esac;
```

The priority controller system

The priority controller system in Figure 5.10 consists of two sensors, a controller and two actuators. Its objective is to ensure that priority is given to actuator A, i.e. actuator B cannot be set to *on* unless actuator A is *on*. There are no priority restrictions when the actuators are switched off.

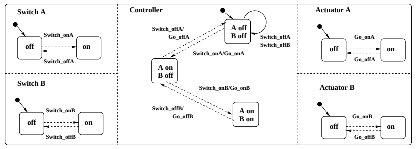


Figure 5.10: The SRS statemachine for the priority controller.

As usual, the fine-grain SMV representation of this system introduces a queue module for declaring the queue of internal events generated by controller transitions. However, only one internal event is generated and therefore we can use a single variable for the internal event as we

did for the AND controller. For a priority controller system that is embedded in a larger system, a queue of events would be required.

```
MODULE Queue(C, A)
VAR
 q :  {swon, swoff, null};

ASSIGN
 init(q) := null;
 next(q):=
  case
   C.CT2 :  swonA;
   C.CT3 | C.CT4 :  swoffB;
   C.CT5 :  swonB;
   C.CT8 :  swoffA;
   A.AT1 | A.AT2 | A.AT3 | A.AT4 :  null;
   B.BT1 | B.BT2 | B.BT3 | B.BT4 :  null;
   1:q;
  esac;
```

The controller module is mainly the same as that of the AND controller system with the exception that in the *next* clause CT6 and CT7 do not update the controller state to the undesirable state of *Aoff_Bon*. This is because of the priority requirement of actuator A over actuator B that is determined by the appropriate reaction to the controller transitions. The controller state must never reach the state *Aoff_Bon* as this will violate the priority requirement. Moreover, the controller transition CT1 must never occur for the same reason.

The actuator modules have transitions that are triggered by the internal event found at the head of the event queue. As for the AND controller system, the actuator module is responsible for updating the actuator state according to the events generated by the controller. The SMV code for the actuator modules is generated using the standard translation rules and is as follows.

```
MODULE ActuatorA(Q)
VAR
 act :  {on, off};

DEFINE
 AT1 := Q.q = swonA & act = off;
 AT2 := Q.q = swonA & act = on;
 AT3 := Q.q = swoffA & act = off;
 AT4 := Q.q = swoffA & act = on;

ASSIGN
 init(act) := off;
 next(act):=
  case
   AT1 | AT2 :  on;
   AT3 | AT4 :  off;
   1:act;
  esac;
```

```
MODULE ActuatorB(Q)
VAR
 act :  {on, off};

DEFINE
 BT1 := Q.q = swonB & act = off;
 BT2 := Q.q = swonB & act = on;
 BT3 := Q.q = swoffB & act = off;
 BT4 := Q.q = swoffB & act = on;

ASSIGN
 init(act) := off;
 next(act):=
  case
   BT1 | BT2 :  on;
   BT3 | BT4 :  off;
   1:act;
  esac;
```

5.4 Discussion

For both the coarse-grain and the fine-grain translations, we have decided not to translate the sensor components as separate SMV modules because the sensor state and events are represented in the controller module. We have shown that our approach still represents all the information provided by the sensors and that it preserves both the coarse-grain and fine-grain semantics of SRS statemachines. The following table gives the performance and BDD statistics for model checking the SMV models generated using the coarse-grain and fine-grain translations for the gas burner system.

nuSMV resources used	Coarse-grain	Fine-grain
User time:	0.06 seconds	0.08 seconds
System time:	0.04 seconds	0.04 seconds
Virtual data size (bytes allocated):	6357K	6536K
BDD nodes allocated:	733	8021
BDD cluster size:	59	570

We considered an alternative translation approach for the coarse-grain and fine-grain that only translates the sensor and actuator components because derived controllers of large systems with many sensors can become very complicated. We discuss what SMV models they generate.

In the alternative coarse-grain translation, explicit SMV modules are described for sensors, and the actuator modules have read-only access to the sensors, instead of the controller, in order to synchronise actuator transitions with sensor transitions. Therefore, sensor transitions are repeatedly defined in the various actuators because several actuators can respond to the same sensor transition. Also, controller guards (reference to sensor states) are defined only in the actuator modules as they already have read-access to the states of all other sensor modules. We can apply the decomposition techniques to the same extent as with the translation without sensors, that is, no explicit representation of subcontrollers, but uses the structure of the system to divide it into separate SMV programs accordingly.

In the alternative fine-grain translation, explicit SMV modules are described for the sensors and actuators. Sensor transitions with controller guards are described in the queue module as they are used to generate the events that will be added to the queue. Without the explicit representation of controllers and subcontrollers, it would not make sense to apply the decomposition techniques to systems modelled in this way. We can however make use of the ways in which the SMV model is divided into separate SMV programs depending on which decomposition technique is applied. This would require that we model the system with controllers (no more than one level) in the RSDS specification and apply the decomposition techniques, but then only translate the sensors and actuators as separate SMV programs respectively.

In the following table we present the resources used for model checking the SMV models (see Appendix B.2 for SMV code) generated by applying the alternative translation approach to the RSDS specification of the gas burner system.

nuSMV resources used	Alternative Coarse-grain	Alternative Fine-grain
User time:	0.05 seconds	0.08 seconds
System time:	0.04 seconds	0.05 seconds
Virtual data size (bytes allocated):	6366K	6509K
BDD nodes allocated:	506	5607
BDD cluster size:	53	569

The BDD nodes for the alternative fine-grain translation is higher than that of the alternative coarse-grain translation because in the fine-grain more states (which are represented by BDD nodes) are required to model the event queue. The performance statistics for the alternative translations are slightly lower than those for the coarse-grain and fine-grain translations presented in this thesis (without sensors), meaning that their performance is slightly faster. For large systems this difference could be significant, especially with the fine-grain translation as the BDD nodes allocated are almost half of the BDD nodes of the translation we presented.

5.5 Related Work

RSDS is unique with respect to its two semantic views of statemachines. Consequently, there is no related work to be found in the literature regarding model checking different levels of semantic views of statemachines. However, these different semantic views of statemachines lead to the definition of RSDS specifications at two different specification levels, comparable to B machines and implementation level. An approach is presented in [BCJ02, BDJK00, JMM99] for combining model checking (SPIN) and B for verifying temporal properties. However, model checking is only applied at the first level, that is to the B machines, and not at the implementations.

We also compare the fine-grain translation with the translations [CH00, CAB+98] discussed (with respect to the coarse-grain translation) in section 4.6. In the fine-grain, the most important behaviour that must be modelled in SMV is the order in which the generated events occur. These events are defined in the statemachines as actions of controller transitions and modelled using an array in SMV. We consider how actions are handled by the other translations and whether such an order is modelled.

In [CH00], generated events that are described as actions of transitions cannot occur in a sequence. For example, the transition in the Controller SRS statemachine in Figure 5.11 Ta : $\alpha/\beta \frown \gamma \frown \delta$ can be expressed in the STATEMATE language as three transitions: $T1 : \alpha/\beta$, $T2 : \beta/\gamma$, $T3 : \gamma/\delta$ and $T4 : \delta$, as illustrated in Figure 5.12. However, it is not always the case in a specification that β generates γ or γ generates δ. There might be another transition in an SRS statemachine $tb : \theta/\beta \frown \delta$ which can be expressed in the STATEMATE language as $T5 : \theta/\beta$, $T6 : \beta/\delta$, leading to an incorrect representation of the SRS transition as when $T5$ is executed, $T2$ and $T6$ will be triggered in the next step, instead of just $T6$. Somehow the generated events need to be linked back to the sensor event and this is not easily done without: either introducing a new event to differentiate between β generated by $T1$ and that generated by $T5$; or introducing a number of dummy states between Cstate1 and Cstate2 whose event is true. Therefore, if we use these solutions to model the fine-grain and then translate into SMV for model checking them, the SMV program obtained has unnecessary states or events which consequently increases the state space. There are similar issues with the translation presented in [CAB+98].

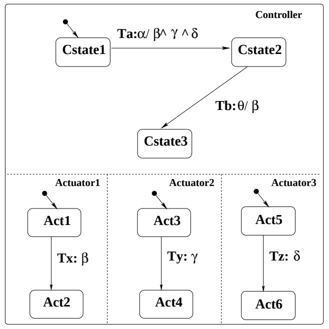


Figure 5.11: SRS statemachines

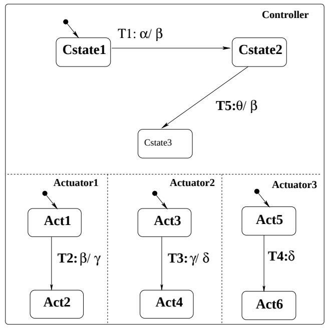


Figure 5.12: STATEMATE statecharts

5.6 Summary

In this chapter, we described how to translate fine-grain RSDS specifications into SMV, using translation rules and schemas. As for the coarse-grain translation, each statemachine module is translated into a single SMV module, except for sensor statemachines that are implicitly represented by the controller. A fine-grain step corresponds to an SMV step. The order in which generated events occur is vital in the fine-grain. It is maintained in the SMV model by a first-in-first-out queue of events (implemented using an array), which has also been used in the definition of the semantics of fine-grain statemachines. All the translation rules for translating fine-grain RSDS specifications into SMV are summarised in Table 5.1.

Table 5.1: Summary of translation rules for the fine-grain

Rules	RSDS	SMV

Rule 1:	An event queue.	`MODULE main`
		 `VAR`
		  `Event_queue :  Queue(Cont, Act1,...,Actp);`
		where *Cont* is the controller module and *Act1...Actp* are the p actuators of a system.

Rule 2:	Each event queue Q with a set of possible generated events $(ge_1, ge_2, ..., ge_m)$.	`VAR`
		 `Q : array 1..max of {ge1, ge2, ...,gem, null};`
		where *max* is the maximum length of the array.

Rule 3:	To determine whether the queue is empty.	`VAR`
		 `is_empty :  boolean;`

Rule 4:	*Initialising the queue*	`ASSIGN`
		 `init(is_empty) := 1;`
		 `init(Q[1]) := null;`
		 ...
		 `init(Q[max]) := null;`
		Each element is set to null and *is_empty* to true.

Rule 5:	For indexes $i : 1..p$ of a sequence of generated events *ge* produced, the transition *tr* that is added to the head of the queue.
	
	The elements at the tail of the queue are moved to their new array positions j where $j = i + p - 1$ where:
	 i is the index of the event in the tail of the queue and,
	 p is the number of generated events for a transition.
		`ASSIGN`
		 `next(Q[i]) :=`
		  `case`
		   –For each transition tr that generates events
		   `tr :  ge(i);`
		   ...
		   –Default case, value in the array stays the same.
		   `1:Q[i]`
		  `esac;`
		
		 `next(Q[j]) :=`
		  `case`
		   –For each transition tr that generates events
		   `tr :  Q[i];`
		   ...
		   –Default case, value in the array stays the same.
		   `1:Q[j]`
		  `esac;`
		
		 `next(is_empty) :=`
		  `case`
		   –If a transition that generates events
		   –occurs then the queue is not empty
		   `tr :  0;`
		   ...
		   –Default case, value in the array stays the same.
		   `1:is_empty`
		  `esac;`

continued on next page

continued from previous page

Rules	RSDS	SMV

Rule 6:	Each event that is processed by an actuator transition *ATr*	`ASSIGN`
		 –For each element in the queue where x is the array index:
		 `next(Q[x]) :=`
		  `case`
		   –The queue is shifted to the left as the event
		   – at the head is removed.
		   `ATr :  Q[x+1];`
		   ...
		   –Default case, value in the array stays the same.
		   `1:Q[x]`
		  `esac;`
		
		 j= i+p-1 where:
		 i is the index of the event in the tail of the queue and
		 p is the number of generated events for a transition.
		 `next(Q[j]) :=`
		  `case`
		   –For each transition tr that generates events
		   `tr :  Q[i];`
		   ...
		   –Default case, value in the array stays the same.
		   `1:Q[j]`
		  `esac;`
		
		 `next(Q[max]) :=`

continued on next page

continued from previous page

Rules	RSDS	SMV

		`case`
		–the last element in the queue is set to null
		`ATr : null;`
		...
		–Default case, value in the array stays the same.
		`1:Q[max]`
		`esac;`
		
		`next(is_empty) :=`
		`case`
		–If a transition occurs and the second element
		–in the queue is null, then the queue is empty.
		`ATr & Q[2]=null : 1;`
		...
		–Default case, value in the array stays the same.
		`1:is_empty`
		`esac;`

Rule 7:	Each state defined in the controller.	`VAR`
		`stable_state : {stb1, stb2, ...,stbn};`
		where $stb_1, stb_2, ..., stb_n$ is the set of possible stable states.

Rule 8:	Initialising the controller state	`ASSIGN`
		`init(stable_state) := init_stb;`
		where init_stb is the initial stable state.

Rule 9:	Sensor events ex	`VAR`
		`sensor_event : {ex1,ex2,...,exp};`
		defined in the controller module.

Rule 10:	Each controller transition	`DEFINE`
		`CT0 := stable_state = stb1 & sensor_event = ex1 &`
		`q.is_empty = 1;`
		where q refers to the module with the event queue.

Rule 11:	The controller state	`ASSIGN`
		`next(stable_state) :=`
		`case`
		–For each controller transition CTi
		`CTi : stb2;`
		...
		–For the default case
		`1 : stable_state;`
		`esac;`

Rule 12:	Each actuator or subcontroller state.	`VAR`
		`st : {st1, st2, ...,stq };`
		where $st_1, st_2, ..., st_q$ is the set of possible actuator or subcontroller states.

Rule 13:	Each actuator or subcontroller transition.	`DEFINE`
		`AT0 := st = st1 & Ma.act & q.is_empty = 0 &`
		`q.Q[1] = e2;`
		where q refers to the event queue module and $Ma.act$ is the guard. The guard is a condition that refers to other actuator states.

Rule 14:	Each actuator or subcontroller state.	`ASSIGN`
		`next(st) :=`
		`case`
		–For each actuator or subcontroller transition ATi
		`ATi : st2;`
		...
		–For the default case
		`1 : st;`
		`esac;`

Rule 15:	Each temporal invariant T	`SPEC`
		`AG(AG(Env) -> T)`
		where Env is the environmental assumptions.

Rule 16:	Each subsystem $SS1...SSn$ defined in the main module.	`VAR`
		`QSS1 : QueueSS1(C, Act);`
		`...`
		`QSSn : QueueSSn(C, Act);`
		where n is the maximum number of subsystems, C is the subsystem's controllers, and Act is the subsystem's actuators.

	OR	

Rule 17:	Define an array for each subsystem in the Queue module.	`VAR`
		`QSS1 : array 1..maxSS1 of {s1e1, s1e2, ...,s1em,`
		`null};`
		`...`
		`QSSn : array 1..maxSSn of {sne1, sne2, ...,snek,`
		`null};`
		
		`is_emptySS1 : boolean;`
		`...`
		`is_emptySSn : boolean;`
		where n is the number of subsystems in a system.

Rule 18:	Each subcontroller or actuator transition.	`DEFINE`
		`AT0 := C.CT1 & q.is_empty = 0 & q.Q[1] = ie1 &`
		`act = sc0;`
		where $AT0$ is a subcontroller or actuator transition, $CT1$ is a controller transition, act is the subcontroller or actuator state and $ie1$ is an internal event.

The correctness of the fine-grain translation was formally shown by proof. Therefore, we guarantee that the SMV model generated preserves the semantics of a fine-grain RSDS specification. In addition, we showed how the results produced from model checking the SMV model are interpreted in the corresponding RSDS specification.

As for the coarse-grain translation, design decomposition techniques have been used to manage the state space explosion problem of model checking. We propose using a separate SMV program for each subsystem and verifying properties locally where possible. Table 5.2 summarises how the decomposition approaches are applied in SMV.

We conclude with a discussion of an alternative translation that could have been used instead and the reasons for choosing the one presented. Furthermore, we consider other possible translations in the literature that could have been used, and discuss why they were not suitable to translate RSDS specifications. In the next chapter we evaluate the RSDS method (with the integration of model checking) against two widely used methods: SCR and PVS.

Table 5.2: Summary of translation of decomposition approaches

Decomposition	**SMV Approach**	**Reduce state space explosion problem**
Hierarchical	The translation rules in Table 5.1 can be applied. The algorithm for calculating the maximum size of the array, checks events at each level.	Virtual sensors can be used for verifying global properties of a system, whose subsystems are defined as separate SMV programs.
Horizontal	The translation rules in Table 5.1 can be applied. Each subsystem can be translated into a separate SMV program.	Each subsystem verifies its properties independently.
Phase	Each phase can be translated into a separate SMV program using the translation rules in Table 5.1.	For global properties, virtual sensors are used. Properties for each active controller are verified independently.
Annealing	There are no benefits in translating this approach	N/A
Standard controllers: priority and AND controller	Translated using the translation rules in Table 5.1.	N/A

CHAPTER 6

Comparison of RSDS against SCR and PVS using the autopilot system

The autopilot system was chosen as a case study to be developed using RSDS with the motive of evaluating RSDS. We compare its performance against the Software Cost Reduction (SCR) method and Prototype Verification System (PVS) which have also been used to specify the autopilot system in [BH96] and [But96] respectively. Although all of these approaches are capable of specifying and verifying reactive systems, we are interested in determining the suitability of these approaches to developing reactive systems by discussing their strengths and weaknesses. First, we develop an RSDS specification of the autopilot system from the set of requirements described in [But96] and show how it is translated into SMV. Then, we evaluate the development process with SCR and PVS.

6.1 The Autopilot Specification

The Boeing 737 autopilot system consists of a mode control panel as in Figure 6.1 with four buttons or switches to allow the pilot to engage or disengage a mode, and three displays and dials for entering the desired values for the *altitude* (ALT), *calibrated air speed* (CAS) and *flight path angle* (FPA). One of the three modes: ATT (*attitude control wheel steering*), ALT, FPA should be engaged at all times independently from the fourth mode CAS, that can be engaged at the same time as the other modes. The displays show the current values of ALT, CAS and FPA unless a desired value is entered by the pilot via the dials. The aim of the autopilot system is to achieve the desired value dialed in by the pilot and maintain the set of modes that are currently active by not violating any of the safety and operational invariants that describe the normal behaviour of the aircraft.

The complete requirements are given informally in English in [But96] and, from these, we use the RSDS method to systematically develop the autopilot specification. There are numerous problems with these requirements which we hope the RSDS development approach will identify (i.e. by applying consistency checking, model checking and the B theorem prover). These problems are summarised as follows:

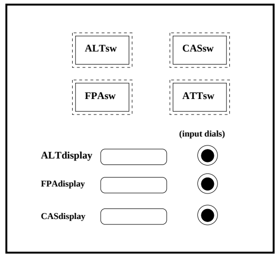


Figure 6.1: Control mode panel

- The initial state of the system is not given.
- Some cases are missing from the specification, for example, the FPA switch is described as a toggle, but there is no definition of what state the system is in after it is toggled off. If such problems are identified, we need to work out as best as we can what the intended behaviour is.
- It is not clear when certain modes are disengaged such as ATT mode. ATT mode is disengaged when another mode is selected.
- There are usability problems with the display, such that it is not clear whether the display shows the current or desired value, and the current mode is not displayed at all. This type of problem is common in the aviation industry and has been generally classified as *mode confusion*.
- Contradictory requirements: this usually occurs when a general case is first given, and then later on, a more specific case is given that contradicts it.

The complete informal requirements as given in [But96] are as follows.

R1 The mode control panel contains four switches for selecting modes and three displays for dialing in or displaying values. The modes are: attitude control wheel steering (ATT), flight path selected (FPA), altitude engage (ALT) and calibrated air speed (CAS). Exactly one of the first three is engaged at any time. CAS can be engaged with any of these. The pilot engages a mode by pressing the corresponding button.

R2 The three displays are: altitude (ALTdisplay), flight path angle (FPAdisplay) and calibrated air speed (CASdisplay). These usually show current values (ALTcurrent, FPAcurrent, CAScurrent) of the aircraft. The pilot can enter a new value into a display by dialing in a value (ALTdesired, CASdesired and FPAdesired) using the dial next to the display. This sets the display to show the target or desired value that the pilot wishes the aircraft to attain. For example, to climb to 25000 feet, the pilot dials in 25000 via ALT dial and then presses ALT switch to engage ALT mode. Once the target value is achieved or the mode is disengaged, the display reverts to showing the current value.

R3 If the pilot dials into ALT an altitude more than 1200 feet above the current altitude and then presses the ALT switch, the ALT mode does not directly engage. Instead altitude engage mode is set to "armed" and the FPA mode is engaged. The pilot must then dial into FPA the desired flight path that is to be followed until the aircraft attains the desired altitude. The FPA mode remains engaged until the aircraft is within 1200 feet of the target altitude and then the ALT mode is automatically engaged.

R4 Target CAS and FPA values need not be selected before corresponding modes are engaged, instead the current values displayed are used. However, the target altitude must be selected before the ALT switch is pressed, otherwise the command is ignored.

R5 The CAS switch and FPA switch toggle on and off every time they are pressed. However pressing the ATT switch when ATT mode is engaged has no effect and similarly for ALT switch.

R6 Whenever a mode other than CAS is engaged, all other displays should return to current.

R7 If the pilot dials in a new altitude while ALT mode is engaged, or if altitude engage is "armed", then ALT mode is disengaged, and ATT mode is engaged. If altitude engage is armed, then the FPA mode is disengaged as well.

6.1.1 Formalising the requirements

The requirements are formalised by converting them into formal invariants in the form accepted by RSDS. Each invariant has a label written besides it that refers to the requirement(s) that it is derived from.

The invariants describe the system behaviour by refering to the states and events of this system. The states are specified for each sensor, controller and actuator and consist of: $\{$*CASmode, CASdisplay, mode, FPAdisplay, ALTdisplay, ALTengage, Alt*$\}$. The following set of events are specified in RSDS: $\{$*CASpressed, FPApressed, ATTpressed, ALTpressed, ALTdialed, FPAdialed, CASdialed, ALTreached, ALTgetsNear, CASreached, FPAreached*$\}$. These events occur as a result of the pilot interacting with the mode-control panel (i.e. pressing buttons, dialing in values), and also from sensors that monitor when the airplane has reached the desired Altitude value, FPA value and so on.

The following invariants describe how the CAS mode changes depending on when the switch is pressed.

$$\mathit{CASpressed}\ \&\ \mathit{CASmode} = \mathit{false} \Rightarrow \mathit{AX}(\mathit{CASmode} = \mathit{true})\ \textbf{(R1, R5)} \tag{6.1}$$

$$\mathit{CASpressed}\ \&\ \mathit{CASmode} = \mathit{true} \Rightarrow \mathit{AX}(\mathit{CASmode} = \mathit{false})\ \textbf{(R5)} \tag{6.2}$$

The next three invariants describe what is displayed on CASdisplay depending on whether CAS mode is engaged, if a CAS value has been dialed or the CAS switch pressed or if the desired

CAS value is reached.

$$CASpressed \;\&\; CASmode = true \;\Rightarrow\; AX(CASdisplay = CAScurrent)\;\;(\mathbf{R2}) \tag{6.3}$$

$$CASdialed \;\Rightarrow\; AX(CASdisplay = CASdesired)\;\;(\mathbf{R2,\ R4}) \tag{6.4}$$

$$CASreached \;\Rightarrow\; AX(CASdisplay = CAScurrent)\;\;(\mathbf{R2}) \tag{6.5}$$

The invariants that follow describe the system changes brought about by events associated with the FPA mode. For example, invariant 6.6 ensures that if a FPA value is entered via the dial, the FPA display will show the new target value and invariant 6.9 ensures that when the desired FPA value is reached by the aircraft, then the FPA display shows the current value of FPA.

$$FPAdialed \;\Rightarrow\; AX(FPAdisplay = FPAdesired)\;\;(\mathbf{R2,\ R4}) \tag{6.6}$$

$$FPApressed \;\&\; mode = FPA \;\Rightarrow\; AX(mode = ATT \vee mode = ALT)\;\;(\mathbf{R5}) \tag{6.7}$$

$$\begin{aligned} FPApressed \;\&\; mode \neq FPA \;\Rightarrow\;& AX(FPAdisplay = FPAcurrent \;\&\\ & mode = FPA)\;\;(\mathbf{R4,\ R6}) \end{aligned} \tag{6.8}$$

$$FPAreached \;\Rightarrow\; AX(FPAdisplay = FPAcurrent)\;\;(\mathbf{R2}) \tag{6.9}$$

Invariants 6.10, 6.11 and 6.12 describe the system changes attained when the ATT switch is pressed. These either set the *mode* to ATT or set the ALT and FPA displays to show their current value respectively.

$$ATTpressed \;\Rightarrow\; AX(mode = ATT)\;\;(\mathbf{R5}) \tag{6.10}$$

$$ATTpressed \;\&\; mode \neq ATT \;\Rightarrow\; AX(ALTdisplay = ALTcurrent)\;\;(\mathbf{R2,\ R6}) \tag{6.11}$$

$$ATTpressed \;\&\; mode \neq ATT \;\Rightarrow\; AX(FPAdisplay = FPAcurrent)\;\;(\mathbf{R2,\ R6}) \tag{6.12}$$

The values appearing on the ALT display are influenced by invariants 6.13 and 6.14: if a new value for ALT is entered via the dial then the desired value will be displayed or if the desired altitude is reached by the aircraft, the current value will be displayed.

$$ALTdialed \;\Rightarrow\; AX(ALTdisplay = ALTdesired)\;\;(\mathbf{R2}) \tag{6.13}$$

$$ALTreached \;\Rightarrow\; AX(ALTdisplay = ALTcurrent)\;\;(\mathbf{R2}) \tag{6.14}$$

For the system to be in the FPA mode and the altitude engage to be armed, the ALT switch must be pressed and the ALT display must show the desired value, the current system *mode* is not ALT and the desired value must be greater than the current value plus 1200 (invariant 6.15). If ALT switch is pressed while the mode is ALT, the command has no effect (invariant 6.16).

$$\begin{aligned} & ALTpressed \;\&\\ & ALTdisplay = ALTdesired \;\&\\ & mode \neq ALT \;\&\\ & ALTdesired > ALTcurrent + 1200 \;\Rightarrow\; AX(mode = FPA \;\&\\ & \qquad ALTengage = armed)\;\;(\mathbf{R3}) \end{aligned} \tag{6.15}$$

$$ALTpressed \;\&\; mode = ALT \;\Rightarrow\; AX(mode = ALT)\;\;(\mathbf{R5}) \tag{6.16}$$

Invariant 6.17 ensures that the system is in ALT mode if the airplane approaches the desired altitude (specified by the event *ALTgetsNear*) within the appropriate range, and the current mode is set to FPA and the altitude engage is armed.

$$\begin{aligned} & ALTgetsNear \;\&\; mode = FPA \;\&\\ & ALTengage = armed \;\Rightarrow\; AX(mode = ALT \;\&\\ & \qquad FPAdisplay = current)\;\;(\mathbf{R3}) \end{aligned} \tag{6.17}$$

These two invariants (6.18 and 6.19) describe the events that occur for the mode to be set to ATT, that is the desired ALT value must be entered with the current mode is set to ALT or altitude engage is armed.

$$ALTdialed \;\&\; mode = ALT \;\Rightarrow\; AX(mode = ATT)\;\;(\mathbf{R7}) \tag{6.18}$$

$$ALTdialed \;\&\; ALTengage = armed \;\Rightarrow\; AX(mode = ATT)\;\;(\mathbf{R7}) \tag{6.19}$$

All the invariants presented for this system are given in the form of operational or action invariants because of the way in which the requirements are expressed. These types of invariants are not recommended as the sole form of invariants for specifying a system as they do not express clearly which states are reachable. For example, in this system any combination of *CASmode* and *CASdisplay* is possible. Nevertheless, these facilitate the translation into SMV.

Furthermore, we have adopted the abstraction techniques involving integers recommended in [But96] that are applied to the display actuators and the altitude sensor. These techniques help keep the verification of the SMV model computable as integers increase the state space immensely. For the display actuators, the actual value of the integers is not significant for verification purposes but whether the display shows the current or the desired value. Therefore, we abstract from the integer value and only have two possible states for each display actuator: *current* and *desired*, thus keeping the state space small.

The values of the altitude sensor **Altitude** are given as integers ranging from 0 to greater than 1200, where no limit is defined. To model this sensor as a statemachine, it would require a state for each possible value. The system is not however interested in all these values, instead it focuses on the relationship between the current value and the desired value i.e. how far is the actual value with respect to the desired value. Therefore, these values are abstracted by grouping them into three ranges: *near, away, at.* The following invariants model this relationship, describing precisely what these ranges mean and are considered as environmental assumptions:

$$\begin{aligned} & (desiredALT \leq actualALT + 1200)\\ & \;\&\; desiredALT \neq actualALT \;\Rightarrow\; Alt = near \end{aligned} \tag{6.20}$$

$$desiredALT > actualALT + 1200 \;\Rightarrow\; Alt = away \tag{6.21}$$

$$desiredALT = actualALT \;\Rightarrow\; Alt = at \tag{6.22}$$

There are a number of invariants that are formalised and used for verification. One of them aims to ensure that if altitude engage is armed, then the FPA mode must be engaged too.

$$ALTengage = armed \;\Rightarrow\; mode = FPA\;\;(\mathbf{R3}) \tag{6.23}$$

The following invariants are taken from the SCR implementation of the autopilot [BH96]. These ensure that if the FPA mode is disengaged, then the FPA display reverts to showing the current value, and similarly for the ALT mode.

$$mode = FPA \;\Rightarrow\; AX(mode \neq FPA \;\Rightarrow\; FPAdisplay = current) \;\; (\mathbf{R2}) \tag{6.24}$$

$$mode = ALT \;\Rightarrow\; AX(mode \neq ALT \;\Rightarrow\; ALTdisplay = current) \;\; (\mathbf{R2}) \tag{6.25}$$

A different type of invariant checks to see if a state is reachable. For example, invariant 6.26 checks if the state where the ATT mode is engaged and the two displays are showing the desired values is reachable.

$$EF(mode = ATT \;\&\; ALTdisplay = desired \;\&\; FPAdisplay = desired) \tag{6.26}$$

It is clear from the data dependencies of the invariants that the autopilot system consists of two independent subsystems. The DCFD in Figure 6.2 shows the horizontal decomposition approach applied to split the control system into the *CAS controller* for the CAS mode and the *Altitude controller* for the other modes of the system (FPA, ALT and ATT). Notice that controller attributes such as Mode and ALTengage are expressed as actuators.

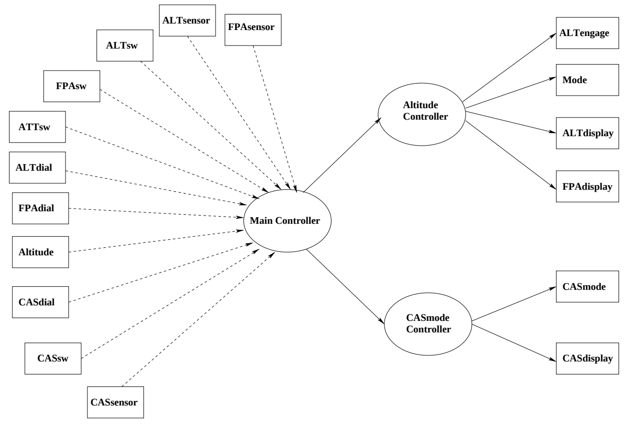


Figure 6.2: The DCFD of the autopilot system showing the horizontal decomposition of controller.

6.1.2 Producing the SRS statemachines

Each component in the DCFD in Figure 6.2 is described in terms of a SRS statemachine module. Since there are two independent subsystems, we will present the statemachines for each subsystem separately. The coarse-grain and fine-grain statemachines are visually the same, just the meaning of a step is different for each.

The CASmode Subsystem

The *CASmode* subsystem consists of the following components: the sensor CASsw that records when the switch is pressed, the sensor CASdial for detecting when the desired CAS value is entered, the actuator CASdisplay that displays the current or desired value and an internal attribute CASmode, that is modelled as an actuator, determines if the CAS mode is engaged. Figure 6.3 illustrates the statemachines for all of these components.

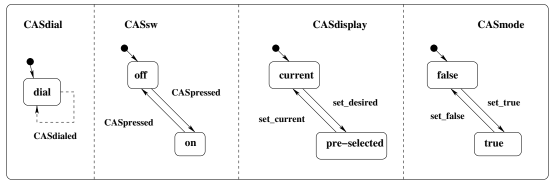


Figure 6.3: SRS statemachines for all the components in the CASmode subsystem.

The statemachine for the controller of the CASmode subsystem is synthesised from its sensor statemachines and from the invariants 6.1 to 6.5 that define what effect the sensor events have on the actuators i.e. what actuator events are generated. The state of the controller is comprised of the tuple of sensor states: (CASdial, CASsw). Since CASdial is a single state statemachine for detecting change, its state is not explicitly represented in the controller statemachine, but its events are. This abstraction is aimed at reducing the state space as we have model checking in mind for verification. Moreover, the CASsw statemachine is isomorphic (a one-to-one correspondence between two graphs, in this case the statemachines) to the CASmode statemachine and these are therefore fused together. Therefore, Figure 6.4 illustrates the controller and the two actuators (CASdisplay and CASmode) of the CASmode subsystem and it is clear from the statemachines that if CASmode is removed, its behaviour is represented by the controller. This is another kind of abstraction that also helps reduce the state space. The first abstraction technique can be automated, however the second would require the user to identify which components describe an internal attribute and since there might be more than one isomorphic statemachine, identify which statemachines to fuse.

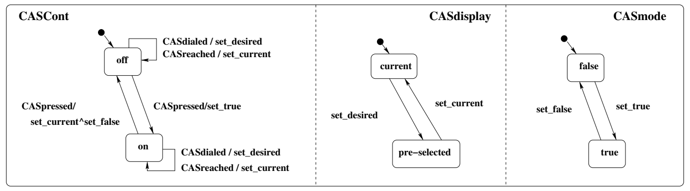


Figure 6.4: SRS statemachine for CAS controller and its actuators.

The Altitude Subsystem

Figure 6.5 illustrates all the components in the Altitude subsystem with a state space of $3^2 \times 2^3 = 72$ states. The process for synthesising the Altitude controller is similar to that of the CASmode controller, i.e. using the amalgamation of sensor states and transitions and the invariants 6.6 to 6.18 for determining the valid transitions. The state of the controller consists of the tuple *(Altitude, FPAsw, ALTsw, ATTsw, ALTsensor, FPAsensor, ALTdial, FPAdial)* but since most of the sensors are single state statemachines, the tuple is reduced to: *(Altitude)*. All the sensor events appear as events triggering controller transitions in the controller statemachine.

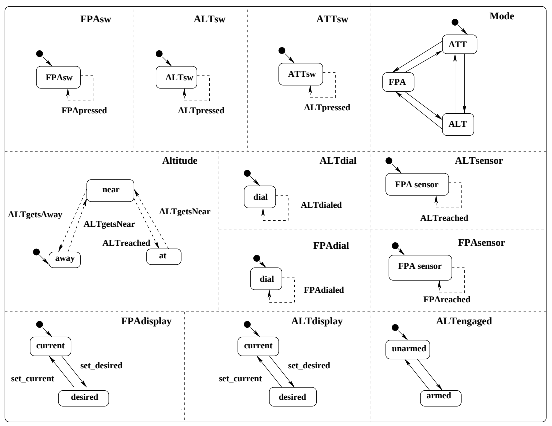


Figure 6.5: SRS statemachines of components of the Altitude subsystem.

There are two internal attributes that are neither sensors or actuators: *Mode* and *ALTengage*. *Mode* indicates the system mode that is currently engaged independently of the CAS mode i.e. either FPA, ALT or ATT is engaged at any one time. ALTengage is used for the case when the system does not directly engage into ALT mode, but will be *armed* and in FPA mode until the aircraft is within 1200 of the actual altitude (from **R3**). There is no direct way in which to represent these attributes in an RSDS specification. The ideal representation of these attributes would be to combine them together with *Altitude* to form the statemachine for the controller, as these attributes refer to the control behaviour of the system. However, in RSDS, the controller is composed from sensors only and therefore, these attributes cannot be included in the controller. Instead, we represent these as separate statemachines composed using AND as illustrated in Figure 6.6, that are treated as actuators. The states where ALTengage is armed and the Mode is in ATT or ALT is not shown in Figure 6.6 because these states are never reached.

6.1.3 Analysis of invariants

RSDS analyses the specification (i.e. the invariants and the statemachines) to detect completeness and consistency problems. This analysis is not automated as yet and we perform it by inspection

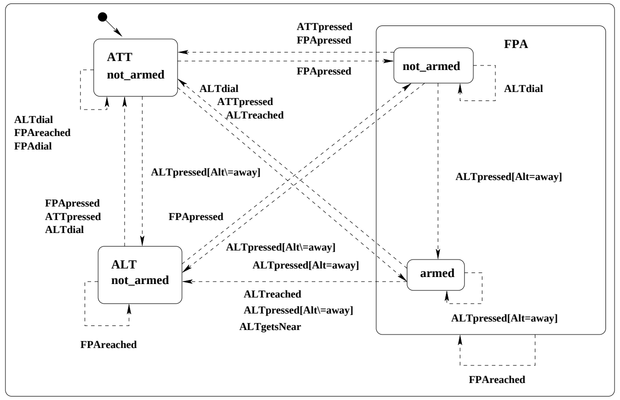


Figure 6.6: SRS statemachine produced by the amalgamation of Mode and ALTengaged.

and, by using B and SMV. We discuss what was detected for the autopilot system. The following completeness problems were identified:

- No initial state is given by the requirements. RSDS statemachines insist upon an initial state. We adopt the initial state used in [But96].
- ALTengage is never set to *not_armed*. To correct this we assume that *ALTengage = armed* means that the system is in the process of reaching the desired altitude, i.e. it is temporarily in FPA mode. Thus, we add the following invariants:

$$\begin{aligned} \mathit{ALTdialed} \;\&\; \mathit{Mode} = \mathit{ALT} \;&\Rightarrow\; \mathit{AX}(\mathit{ALTengage} = \mathit{not\_armed}) \;\; \textbf{(R7)} && (6.27)\\ \mathit{ALTdialed} \;\&\; \mathit{ALTengage} = \mathit{armed} \;&\Rightarrow\; \mathit{AX}(\mathit{ALTengage} = \mathit{not\_armed} \;\&\\ &\qquad \mathit{FPAdisplay} = \mathit{current}) \;\; \textbf{(R7)} && (6.28)\end{aligned}$$

  and the RHS of invariant 6.17 is updated:

$$\begin{aligned} &\mathit{ALTgetsNear} \;\&\; \mathit{mode} = \mathit{FPA} \;\&\\ &\qquad \mathit{ALTengage} = \mathit{armed} \;\Rightarrow\; \mathit{AX}(\mathit{mode} = \mathit{ALT} \;\&\\ &\qquad\qquad \mathit{FPAdisplay} = \mathit{FPAcurrent} \;\&\; \mathit{ALTengage} = \mathit{not\_armed})\end{aligned}$$

  and similarly for the RHS of invariants 6.7 and 6.12.
- It is apparent that some cases are missing from the specification of events ALTpressed, ALTgetsNear, ALTgetsAway, when we analyse the disjunction of the conditions on the LHS of the invariants:
  - For ALTpressed these cases are not specified:

1. $ALTdisplay \neq ALTdesired$
2. $ALTdisplay = ALTdesired \wedge mode = ALT$
3. $ALTdisplay = ALTdesired \wedge mode \neq ALT \wedge Alt \neq away$

The default null action seems correct for the first two cases, but the third seems to be a genuine missing requirement. To correct this we add the invariant:

$$\begin{aligned} & ALTpressed \ \& \\ & ALTdisplay = ALTdesired \ \& \\ & mode \neq ALT \ \& \ Alt \neq away \quad \Rightarrow \quad AX(mode = ALT \ \& \\ & \qquad FPAdisplay = FPAcurrent \ \& \\ & \qquad ALTengage = not\_armed) \ \ \textbf{(R3)} \end{aligned} \qquad (6.29)$$

– ALTgetsNear is missing a specification of what happens if the event occurs when:

1. $mode \neq FPA$
2. $mode = FPA \wedge ALTengage \neq armed$

We assume a null action was intended for both cases.

– ALTgetsAway is missing a behaviour specification. We assume that a null action was intended.

- FPAsw is described as a toggle in the requirements, but there is no definition of what state the system is in after it is toggled off. We assume that ATT mode is engaged when FPAsw is toggled off.

- Usability problems with the display have also been identified such as it is not clear whether the display shows the current or desired value or that the display does not show the current mode. This is a serious problem in aviation systems and is classified as *mode confusion*. In [LC99], the authors show the advantages of using model checking over theorem provers for verifying the mode confusion properties. The Nagoya crash [Nag] is an example of a catastrophe that arose from the mode confusion problem. Nevertheless, we choose to implement the specification based mostly on the existing implementation for consistency.

In conclusion, the invariants must be defined for all possible event combinations with guards if required on the LHS of the implication and on the RHS,and a value for each actuator must be provided.

RSDS detects the following inconsistencies:

- **R3** and **R7** are contradictory. **R3** says that FPA should remain engaged until altitude becomes “near” its target. **R7** says mode can go to ATT instead. This is typical of many requirements where a general case is stated and then a special case contradicting it is given later on. We assume **R7** should apply.

- **R6** suggests that the ALT display should be reset when we enter FPA mode in **R3**, or enter ATT mode in **R7**. Also that CAS display should be reset if FPA, ATT or ALT is entered.

There is also inconsistent behaviour of the interface from a user perspective, in that sometimes other displays are reset when we enter a mode, and sometimes not. This could be misleading for operators and lead once again to the mode confusion problem. In [Lev97], an approach is presented for detecting errors induced due to automated control of software systems (especially aviation), that can be used for designing the operator interface.

6.1.4 Verifying the autopilot system

RSDS specifications are verified by the B theorem prover and the SMV model checker. A translation [LBA99, LAK00, LAC00] already exists from RSDS specifications to B AMN machines and in this thesis we have defined one from RSDS specifications to SMV. We recall briefly what is considered as the input of the translations for the autopilot system.

The RSDS specification of the autopilot system consists of invariants and statemachines. The structure of the autopilot system is depicted in Figure 6.7 where the horizontal decomposition approach has been applied and the system is divided into two independent subsystems that do not share any sensors or actuators. The controllers for the subsystems are: CAScont for the CASmode subsystem and ALTcont for the Altitude subsystem, that process distinct events independently of each other. SRS statemachines are used to describe the dynamic behaviour of each component in the subsystem. The controller statemachines are generated automatically by combining the sensor statemachines. The invariants that formalise the requirements are used to identify the valid transitions in the controller statemachines. Some abstractions have been applied on the statemachines that have already minimised the state space and we summarise them:

1. Sensor states whose behaviour is described with single state statemachines are not part of the state of the derived controller. In the autopilot system, both subsystems contain a number of single state sensor statemachines that are ignored in the controller, for example, the ALTcont is derived from the altitude sensor and the other seven sensors’ states are ignored. However, the sensor events are not ignored and appear in the controller transition definitions.

2. The second abstraction that was applied involves the CASmode subsystem. The statemachine for the CASsw component is isomorphic to that of the CASmode component. Since CASmode is an internal attribute that is implemented as an actuator and CASsw determines its changes, we have decided to fuse the two statemachines, by implementing only the CASsw in the CAScont controller, thus avoiding any unnecessary repetition.

Using B for verification and validation

The B specification for the autopilot is derived from the statemachine modules and invariants that make up the RSDS specification. For each event *e*, all of the invariants that concern it are collected and their effects are combined to form the definition of the B operation for *e*. For example, the invariants for the event *CASpressed* are 6.1, 6.2 and 6.3 and their effect are combined to form the following B operation.

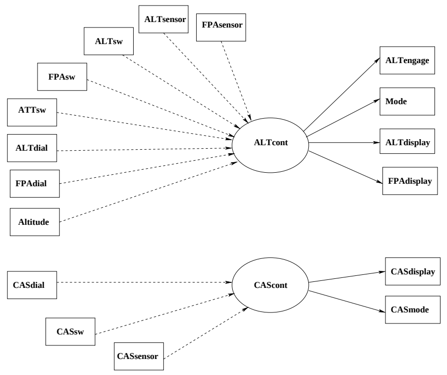


Figure 6.7: The autopilot system is horizontally decomposed resulting in two independent subsystems.

```
CASpressed =
IF
 CASmode = false
THEN
 CASmode := true
ELSE
 CASmode := false ||
 set_current_CASdisplay
END;
```

The rest of the operations are formulated similarly. The structuring of the B machines illustrated in Figure 6.8 follows the receivers hierarchy defined by the DCFD diagram and it respects the horizontal composition of controllers. The complete B source can be found in Appendix C.

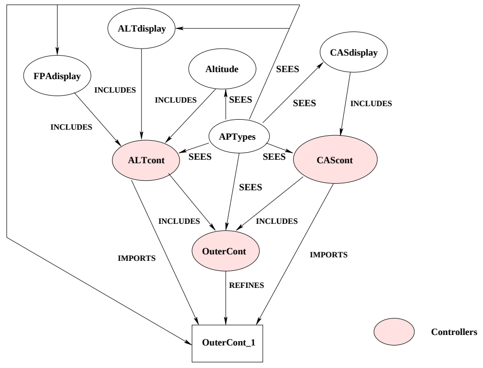


Figure 6.8: The B Development Architecture of the Autopilot.

The B translation allows us to check that the expected static invariant 6.23

$$ALTengage = armed \Rightarrow mode = FPA$$

is actually provable from the formalisation of the requirements, or to identify where it fails. The proof obligations generated for these B machines by the BToolkit reveal that the invariant does not hold: $ALTengage = armed$ can occur with $mode = ATT$ as a result of *ATTpressed*, if *ALTengage* is armed with *FPA* mode active before this event. To correct this we add the extra invariants 6.23 and 6.24 and correct invariants 6.7, 6.12, 6.17 to ensure that *ALTengage* is set to *not_armed*.

Verification of temporal invariants is not supported in B without some additional enhancements to the method. However, B is capable of using the environmental assumptions in the specification to define the data abstraction in the outer-level controller. The outer-level (interfacing) controller is responsible for detecting events by polling, or sampling the environment and broadcasting the events to the main controller one at a time. It is implemented in B as a separate B machine **OuterCont** which is refined into the implementation **OuterCont_1**. A single operation *cycle* describes the polling of the environment for events. The environmental assumptions, i.e. the invariants 6.20, 6.21 and 6.22 in the autopilot system defining the altitude ranges *away, at* and *near*, are used as follows to check the relationship between the actual and the desired altitude value and to broadcast the appropriate events.

```
IF
 desiredALT <= 1200 + actual
THEN
 ALTgetsNear
ELSE IF
 desired = actual
 THEN
  ALTreached
 ELSE
  AltgetsAway
 END
END;
```

Moreover, B toolkit provides animation facilities to explore user scenarios to check that the formalised invariants accurately capture the intended behaviour of the system. For example, a possible scenario of normal behaviour for the Altitude subsystem could be given as in Figure 6.9. The altitude values in bold indicate which value is currently being displayed by the actuator *ALTdisplay*.

Model Checking the Autopilot System

SMV code is generated from the RSDS specification of the autopilot system using the translation rules described in chapter 4. There are two semantic views of RSDS specifications: one for the coarse-grain view and one for the fine-grain. For the fine-grain, an order of actuator events must be described in the requirements and formalised in the invariants. The invariants of the autopilot do not define any precedence of actuator events over others. Therefore, we believe that there are no benefits in implementing the fine-grain view for this system, although this can be easily done.

Event	Desired	Current	Altitude	Old Mode	New Mode
ALTdial	30000	**5000**	away	ATT	ATT
ALTpressed	**30000**	5100	away	ATT	FPA & armed
FPAdial	**30000**	5200	away	FPA & armed	FPA & armed
FPAreached	**30000**	10500	away	FPA & armed	FPA & armed
ALTgetsNear	**30000**	28800	near	FPA & armed	ALT
ALTreached	**30000**	30000	at	ALT	ALT
ALTaway	30000	**30100**	near	ALT	ALT

Figure 6.9: An example scenario of normal behaviour for the autopilot system.

The complete listing of the SMV code generated for the coarse-grain view of the autopilot can be found in the appendix C.

In SMV, the main module shows the dependencies between the different system components, as illustrated by the DCFD, that run in parallel (representing the concurrency of SRS statemachines). In this system, it is clear that the CAScont controller and its actuator are independent from ALTcont controller and its actuators. Also, the dependencies between the modules of the components in a subsystem are evident, for example all the actuators in the Altitude subsystem have read-only access to the ALTcont module as well as to the Mode actuator module.

```
MODULE main
VAR
 –For the CASmode subsystem
 c1 :  CAScont;
 cas :  CASdisplay(c1);

 –For the Altitude subsystem
 c2 :  ALTcont;
 mo :  Mode(c2, alt, en);
 en :  ALTeng(c2, mo, alt);
 fpa :  FPAdisplay(c2, mo, en, alt);
 alt :  ALTdisplay(c2, mo);
```

The SMV code for each module is derived from the statemachines. Note that the internal attributes Mode and ALTengage are represented in RSDS as actuator statemachines and are translated into SMV modules using the translation rules described in Chapter 4. Therefore, the states of SMV modules for these attributes will change value depending on the controller transitions, and, any actuator in the system can access these states if needed by guard definitions in their transitions.

The action invariants used to formalise the requirements are well-represented in the SMV model. For example, lets consider invariant 6.8:

$$FPApressed \,\&\, mode \neq FPA \;\Rightarrow\; AX(ALTdisplay = ALTcurrent \,\&\, mode = FPA)$$

The event on the LHS of the implication is represented by the controller transition CT4, CT5, and CT6 since the current altitude range value has not been specified.

```
MODULE ALTcont
```

```
VAR
 state :  {near, away, at};
 event :  {ATTpressed, FPApressed, ALTpressed, ALTgetsNear,
  ALTreached, FPAreached, ALTdial, FPAdial};

DEFINE
 CT1 :=event = ATTpressed & state = near;
 CT2 :=event = ATTpressed & state = away;
 CT3 :=event = ATTpressed & state = at;
 CT4 :=event = FPApressed & state = near;
 CT5 :=event = FPApressed & state = away;
 CT6 :=event = FPApressed & state = at;
 ...
```

The effect of this invariant, expressed by the RHS of the implication, is represented in the local transitions MT2 of the Mode actuator and AT1, AT2 of the ALTdisplay actuator.

```
MODULE Mode(OC, a, En)
VAR
 mode :  {ATT, ALT, FPA};

DEFINE
 MT1 := !(mode = ATT) & (OC.CT1 | OC.CT2 | OC.CT3) ;
 MT2 := !(mode = FPA) & (OC.CT4 | OC. CT5 | OC.CT6) ;
 ...

ASSIGN
 init(mode) := ATT;
 next(mode) := case
   MT1 :  ATT ;
   MT2 :  FPA ;
   ...
   1 :  mode;
 esac;
```

```
MODULE ALTdisplay(C, M)
VAR
 ast :  {desired, current};

DEFINE
 AT1 := (C.CT4 | C.CT5 | C.CT6) & !(M.mode = FPA) & ast = desired ;
 AT2 := (C.CT4 | C.CT5 | C.CT6) & !(M.mode = FPA) & ast = current;
 ...
ASSIGN
 init(ast) := current;
 next(ast):= case
   AT1 | AT2 :  current ;
   AT3 | AT4 :  desired ;
   ...
   1 :  ast;
 esac;
```

Note that *mode ≠ FPA* is a guarding condition that forces the actuator components to have read-only access to the mode actuator. This system is rich with guards which results in high interdependency of modules. Moreover, the actuator transition definitions are abridged for the purpose of presentation, such as the transition MT1 that is defined as:

```
MT1 := !(mode = ATT) & (OC.CT1 | OC.CT2 | OC.CT3);
```

where | is an OR connective. In the automated version there will be three transitions, one for each controller transition. The state space is not increased with the additional definitions of transitions.

The outer-level controller is not translated into SMV as the environment is modelled by under-specification where the events occur non-deterministically and the system must be able to react to all possible events. We abstract from the details of how the events are received from the environment. These invariants will have to be ensured by the developer or in B.

The RSDS specification is translated into SMV in order to verify its temporal properties, invariants 6.24, 6.25, that cannot be verified in B. These are described as follows in SMV.

```
SPEC AG(mo.mode = FPA → AX(!(mo.mode = FPA) → fpa.fst = current))
SPEC AG(mo.mode = ALT → AX(!(mo.mode = ALT) → alt.ast = current))
```

A counter-example was produced for 6.24, as the model was incomplete. The RHS of invariant 6.7 must be updated to include the condition that sets the FPA display to current. Furthermore, two more invariants must be added since 6.13 and 6.14 are not enough. These set the FPA display to current when *ALTdialed* and *ALTreached* occur. Additional transitions were added to update the SMV model.

$$ALTdialed \,\&\, ALTengaged = not\_armed \quad \Rightarrow \quad FPAdisplay = FPAcurrent \qquad (6.30)$$

$$ALTreached \,\&\, mode = FPA \quad \Rightarrow \quad FPAdisplay = FPAcurrent \qquad (6.31)$$

Alternatively, model checking can check if a state is reachable. To check if a state such as

$$P\!: mode = ATT \;\&\; ALTdisplay = desired \;\&\; FPAdisplay = desired$$

is reachable, we can try to prove AG(!P). This is equivalent to !EF(P), the negation of invariant 6.26. If P is reachable a counter example sequence will be produced by SMV. In this system we get the counter example sequence:

ALTdialed; FPAdialed

which establishes P. In this case almost all possible states are reachable, and therefore the actuator settings are not at all predictable from the mode. Again this is evidence of poor human factors and design of the system.

Any invariant that is verified using B, can be verified in SMV, such as invariant 6.23. It is expressed in SMV as:

SPEC AG (en.Alt_armed = armed) → *AX (mo.mode = FPA))*

In addition, the invariants used to formulate the invariants can be checked in SMV. For example, in the CAS mode subsystem, the following invariant is true of the autopilot model.

SPEC AG (c1.cas_event = CASdial → *AX (cas.st = desired))*

The resources used to model check the autopilot system with NuSMV are as follows. Since the values for user and system time, and virtual data, vary each time the model is checked, we have taken the average for these usage values for ten runs of the autopilot model with SMV.

User time:	0.06 seconds
System time:	0.04 seconds
Virtual data size (bytes allocated):	6409K
BDD nodes allocated:	1816
BDD cluster size:	178

6.1.5 Discussion

The autopilot system was chosen to be developed using RSDS primarily for the purpose of comparison. However, this is the first system developed by RSDS from the aviation domain. It shows that in general RSDS is suitable for developing systems from this domain.

The development of the different aspects of the autopilot system are well-supported by RSDS. For example, not only is the system's complete reaction to its environment specified, but also the system's interface with its environment is specified by the outer-level controller in B where environmental assumptions can be ensured. The SMV language is particularly well-suited to modelling the action invariants and these are used solely for formalising the system's requirements. The disadvantage of using action invariants is that it is unclear which states are reachable. We overcome this disadvantage by verifying reachability properties in SMV that aid the detection of incompleteness and inconsistencies in the specification. Moreover, the process of formulating the invariants is difficult and can result in missing cases and contradictions that are detected by the verification tools and RSDS checks.

The drawback of using this approach for the autopilot system concerns the construction of the controller from the sensors, as this is not very meaningful. In this particular system there are two internal attributes (Mode and ALTengage) that are modelled as actuators which result in a high level of interdependency between these modules. This suggests that Mode, and possibly Mode combined with ALTengage as in Figure 6.6 are good contenders for being the controller of this system. It would be very good if we could extend RSDS to be able to represent internal attributes in controllers, while still ensuring that development occurs systematically and is automated as much as possible. This partly motivates the object oriented version of RSDS described in Chapter 7, where the controller is modelled as a class that may have a number of attributes.

With respect to the translation process, we have previously mentioned [Day98] that highlights the problems with translations and [KG02] that has identified factors that pertain to the quality of a translation. We comment on the extent to which these have been met or dealt with, based on the experience gained from implementing the autopilot specification.

In [Day98], three main problems are identified and we respond as follows:

1. **There is no guarantee that the translation preserves the semantics of the source language.** For each translation that was described in Chapters 4 and 5, a proof of correctness was also given showing that the translations are semantic-preserving. If these translations are to be automated, then we need to ensure that they are correctly implemented. This is especially important for safety-critical systems. In [PSS98], a method for validating the

implementation of the translation is described. For the autopilot system, we manually translated the RSDS specification into SMV, applying the translation rules given. We know that the rules are correct, because of the proof of correctness, and therefore we can focus on ensuring that the rules are applied correctly. Since the SMV model produced is not very big, we can easily check by inspection that we applied the rules correctly.

2. **The results produced from some processing performed on the target notation are not represented in the source notation.** In this case, the type of processing performed on the target notation is verification, using B and SMV. The B translation has not presented a way of interpreting its results onto the RSDS specification, this remains to be tackled in future work. For the translations to SMV, Chapters 4 and 5 describe how a counter-example is interpreted in the RSDS specification. Some counter-examples were produced for the autopilot specification. The counter-examples were manually interpreted on the RSDS specification. Any necessary corrections were made to the RSDS specification and then it was re-translated into SMV and its properties were model checked. This process was repeated until the results were true for all of the properties.

3. **The translation is notation-specific and hence limited to the processing capabilities of that notation.** We don't consider this to be a limitation because we want to make use of a notation's additional capabilities over our own method's, i.e. model checking, theorem-proving. We believe that it is better to integrate existing tools that are actively maintained and familiar, rather than building new tools to perform the same capabilities. For the autopilot system, the different verification techniques that were applied identified different problems with the specification that might not have been detected if a single verification system was used.

The factors (semantic, syntactic and efficiency) for evaluating the quality of the translations are described in [KG02]. The semantic factors consider whether the elements in the source language correspond to elements in the target language. It is not always the case that all elements in the source language can be adequately represented in the target language. The syntactic factors consider the desired properties of the target model such as modularity and readability. Since one of the target languages is a model checker, the efficiency factors question how well the state space explosion problem is dealt with in the target language. We comment on these based on our experience from developing the autopilot system using RSDS.

- **Semantic Factors:** The translation presented from an RSDS specification (coarse-grain) to SMV is not direct. We use all the information that we already know about coarse-grain RSDS specifications in order to translate. This is because we want to ensure that a step in RSDS corresponds to a single step in SMV. This is achieved by not translating the explicit generation of internal events. Instead, the actuator transitions are defined in terms of controller transitions, thus ensuring that a coarse-grain step corresponds to a single SMV step. Moreover, the translation is semantics-preserving as a proof of correctness is given.

  For the autopilot system, not all elements can be translated directly. The internal attributes Mode and ALTengage, that should be part of the controller, have to be represented as

  actuators because RSDS specifications cannot represent internal attributes. However, this limitation belongs to the expressivity of RSDS specifications and not to the translation or target notation (SMV). In Chapter 7, an object oriented version of RSDS is presented, namely RSDS/UML, that allows for the representation of internal attributes.

- **Syntactic Factors:** The modularity of RSDS specifications is maintained in the SMV model, i.e. each statemachine modules is translated into a separate SMV module. The structure of the SMV model usually corresponds to the structure illustrated by the DCFD diagram, showing the dependencies of the modules. In the autopilot system, two additional actuator modules are added in the DCFD in order to specify the internal attributes, but these are all translated into SMV in the standard way.

- **Efficiency:** The SMV model that is produced by the translation should be small enough so that it can be model checked automatically by the tool. There are no problems with size of the SMV model produced for the autopilot system, as it can be model checked in a fraction of a second.

6.2 Comparison of RSDS, SCR and PVS

We base our comparison on the development steps in an ideal design strategy for supporting the entire development process. We aim to answer the following questions with respect to each development phase.

Requirements analysis and specification development: How well has the specification been formulated from the set of requirements? Was there a set of guidelines that were followed? Was the system specified at the right level of abstraction ?

Design: Can the system be decomposed further and how successful is its architecture ?

Validation: Does the development method provide ways of checking that the system does what the user intended ?

Verification: Is the system modelled correctly ?

Another two aspects that we are interested in comparing among the methods, are:

Abstraction techniques for reducing the state space: Can we apply some abstraction technique to reduce the state space ? We are interested in these techniques for scalability and for model checking purposes. Even though PVS does not use model checking, it may still benefit from abstractions.

Tool support: Is there a tool that supports the development process and to what extent ?

Before we compare the different approaches for specifying and verifying the autopilot system, we provide an overview of each method. Since the SCR method is very similar to RSDS, we examine it in more detail and compare it with RSDS.

6.2.1 Overview of SCR

SCR is a set of techniques used for developing systematically formal specifications from a set of requirements. It was developed from a collaboration between David Parnas and Constance Heitmeyer, and other researchers from the U.S. Naval Research Laboratory (NRL) in the late 1970's. It consists of two formal models [Hei02]: the Four Variable Model and the SCR requirements model. In [HBGL95] a set of software tools were developed to analyse documents with SCR requirements.

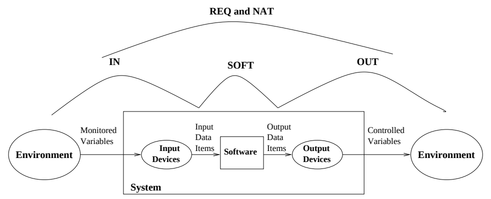


Figure 6.10: The Four Variable Model.

The Four Variable Model, illustrated in Figure 6.10, defines the required system behaviour as a set of relations on four sets of variables: *monitored* variables that describe the system environment (non-deterministic), *controlled* variables that describe the required system behaviour (deterministic), *input* and *output data items* that are resources available to examine the monitored and controlled quantities respectively. The relations are: NAT, REQ, IN and OUT. **NAT** describes the physical constraints on the system behaviour as assumptions and is defined as a set of possible values, and **REQ** describes the required relation between the monitored and controlled quantities maintained by the system. In RSDS, environmental assumptions can be compared to NAT and the RSDS system model to REQ. The IN and OUT relations specify the rigour required for measuring the values of monitored quantities and the computability of the controlled quantities as mappings from monitored or controlled variables to input or output variables. In RSDS, the sensor and actuator variables in the statemachines correspond to the input and output variables in SCR. Figure 6.11 delineates the relationship between the RSDS system defined by the DCFD diagram and the system described by the Four Variable Model. The labels in brackets refer to the corresponding SCR elements.

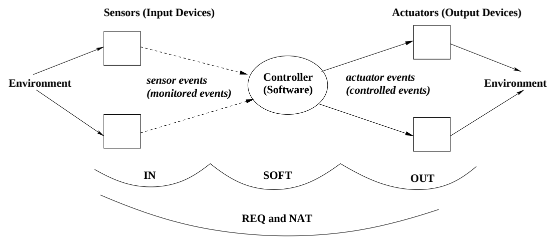

Sensors (Input Devices)
Actuators (Output Devices)
Environment
sensor events (monitored events)
Controller (Software)
actuator events (controlled events)
Environment
IN
SOFT
OUT
REQ and NAT

Figure 6.11: Visual comparison of the DCFD with the Four Variable Model.

The SCR requirements model [HJL96] is a special case of the Four Variable Model that provides precise semantics by representing the system as a statemachine and focusing on the REQ and NAT relations. As in the RSDS method, the system responds to a single *input event* from the environment by changing state (initial state specified) and possibly changing some controlled variables by producing one or more *system outputs*.

REQ is specified by using the following constructs: *mode classes*, *terms*, *conditions* and *events*. A mode class is a state machine with *system modes* as states and *transitions that are triggered by events*[1]. A term is any function of input variables, modes or other terms that are used in making the specification more concise. A *condition* is a predicate defined on one or more input or output variable, mode or term at any particular time. An *event* occurs when either variables (input or output, mode or term) changes value. There are some special types of events like the *input event* that occurs when an input device (sensor) changes value, and the *conditioned event* that occurs when a specified condition becomes true. A conditioned event is defined as:

$$@T(c)\ \ WHEN\ \ d \longrightarrow\ \neg c\ \wedge\ c'\ \wedge\ d$$

where c and d are evaluated in the old state and the primed condition c is evaluated in the new state. Basically it means that whenever a transition for the event $@T(c)\ WHEN\ d$ occurs, the old state of c is false and the new state of c is true and d was true in the old state but in the new state can be either true or false. There is also another notation $@F(c)$ that is defined by $@F(c)\ =\ @T(\neg c)$. The conditioned event is similar to a transition in the SRS statemachine in RSDS

$$t : e \wedge [G] \setminus \alpha_1 \frown ... \frown \alpha_n$$

where c can be mapped to a condition that describes the current state the system is in, and d can be mapped to the logical guard G (see Figure 6.12). The SCR transition does not have a particular event name, but it is unique because of the conditions that define it. Therefore, in $@T(c)$ in SCR can be mapped to the event name e in RSDS. The actions $\alpha_1 \frown ... \frown \alpha_n$ are not explicitly given in SCR, but are modelled as *term* variables whose values are expressed in the event and condition tables.

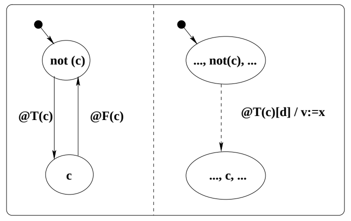


Figure 6.12: The SCR transitions expressed using SRS notation

There are many similarities between the SCR requirement model and the SRS statemachine semantics of RSDS system specifications. In SCR, monitored and controller variables are assigned

[1]This can be compared to the state design pattern that has a separate class for each mode [GHJV95] as well as the phase decomposition approach in RSDS.

discrete values, and RSDS develop specifications for only discrete systems. They both assume that only a single monitored or sensor event will trigger a state transition at a time and in SCR this is called the *One Input Assumption*. The *Synchrony Assumption* of SCR requires the system to process one monitored event completely before going on to process the next monitored event and this corresponds to the fine-grain model assumptions in RSDS for a reaction cycle. No circular dependencies of state variables are allowed in the definitions of state variables in SCR which applies for RSDS as well.

In SCR, a tabular notation is used for writing specifications for these relations. Three types of tables are used: mode transition tables, event tables and condition tables, each defining the variables mode, event, condition as a function. The mode transition table describes the system modes in terms of the system states and how they are changed when transitions occur. Then event tables are used to describe the changes to the variables depending on the events that occur and also which mode the system is in (the mode does not always appear in the event tables). Condition tables describe how the values of the output variables or terms change as a function of a mode and condition. These changes are similar to the changes brought about as a result of an actuator transition in an RSDS specification. The tables spell out the steps of execution, showing what the values of variables are and how they change when transitions occur and which events become true. This level of detail is comparable to that described by the *fine-grain semantics*[2] for RSDS. The system properties must be satisfied by the information held in each table.

The invariants to be verified by model checking are written as logical formulae in both SCR and RSDS methods. However SCR concentrates on two types of invariants [BH99]: *state invariants* that have the same form as static invariants in RSDS, and *transition invariants* that have the same form as action invariants in RSDS.

The SCR tool [HBGL95] provides a specification editor to allow the user to create, modify or display a specification as well as a consistency checker, simulator and verifier. For verification it uses mainly the SPIN model checker but there exists a translation to SMV as well. TAME (a friendly front end version of PVS) or salsa [BS00a] is available for use when the state space explosion problem can hinder verification. Currently, the SCR has better tool support capabilities than RSDS and has been used in industry to specify many critical reactive systems.

6.2.2 Overview of PVS

PVS is a general verification system that was developed by SRI International [SRI]. It consists of a specification language based on higher-order logic and enhanced with a rich type system that is integrated with support tools and a theorem prover. It focuses on providing verification support for non-finite or finite systems with a powerful interactive theorem prover as opposed to the SCR and RSDS methods that focus on designing specifications from requirements for finite systems. Its expressive specification language enables users to develop specifications from any application domain and using any development approach. SCR and RSDS are considered restricted compared to PVS that is flexible by supporting various specification styles. However, since there is no

[2]The fine-grain semantics separates the steps within the reaction and represents these as successive steps in the tables.

	Similarities
1.	Both assume: only one sensor event will trigger a transition at a time.
2.	Both process one event completely before processing another.
3.	Both describe systems with discrete values.
4.	Both distinguish sensor and actuator values, (known as monitored and controlled variables in SCR.
5.	Both dont allow circular dependencies of state variables
6.	The level of detail in SCR tables is comparable to the fine-grain semantic view of RSDS.
7.	Both integrate existing tools for verificaion (by translation).

Figure 6.13: The similarities between RSDS and SCR.

	RSDS	**SCR**
1.	Notation: statemachines and invariants.	Tabular notation.
2.	More types of invariants are defined.	Only two types of invariants: state invariants and transition invariants.
3.	Tool still in the initial stages of development.	Very good tool support.
4.	Integrated with SMV and B.	Integrate with PVS, SPIN and SMV
5.	Statemachines suffer from the state explosion problem.	Specifications have many tables that can be hard to follow.
6.	A number of decompositions can be applied.	Mode decomposition is always applied.

Figure 6.14: The difference between RSDS and SCR.

systematic method given for producing specifications, it relies heavily on human ingenuity and this could be time consuming and may lead to ill-designed specifications. It can be incorporated into another method that can use its powerful theorem prover for verification, for example it is part of the techniques associated with the SCR method used for verification.

The PVS development process consists of the following:

1. A specification in the **PVS specification language** is produced using any style of specification, such as declarative, axiomatic and algorithmic. The specification is described as a collection of theories.
2. The specification is **parsed automatically** to identify syntax errors.
3. The specification is **type-checked** to identify any semantic errors, such as ambiguous types and undeclared names. Since type-checking is undecidable, proof obligations are produced and most are discharged automatically. This type-checking allows for PVS to provide simple solutions to issues that are considered difficult in other systems, such as accommodating partial functions. Moreover, it enforces strong checks on consistency and other properties in a reliable manner.
4. The **theorems are proved**, automatically or interactively. For interactive proofs, a number of atomic commands can be used for each deductive step, such as induction, quantifier reasoning and automatic conditional rewriting. Each proof is divided into a number of subgoals and is constructed by prompting the user to give a suitable command to prove a given subgoal. The execution of the command can lead to the generation of further subgoals or complete the proof of the subgoal and move to another unproven subgoal. The user can define proof strategies that are applied as commands to enhance automation of proofs. Model checking has also been incorporated into PVS for automatically verifying finite-state specifications.
5. The PVS tool is used to **produce documentation** of the specification.

6.2.3 Evaluation of SCR, PVS and RSDS with the autopilot system

The autopilot was specified in [BH97a] using the SCR tabular notation, with the relation REQ describing the required system behaviour by expressing the relationship between the monitored and controlled variables. A mode transition table is used to describe the transitions that occur when the system mode changes from ALT, ATT or FPA. A number of event and condition tables are used to describe how the term variables are changed when the various events occur depending on the system mode. For example, an event table is defined for the term *tCASmode* (of type boolean) that represents the CAS mode and it shows when its value is set to $TRUE$ or $FALSE$ depending on the transitions that occur.

In [But96], the autopilot system was specified in the PVS language. Two modelling techniques were used: one involved representing the system as a statemachine and the other embraced the SCR modelling approach. The specification that was modelled on statemachines consists of states for each system mode and events for inputs initiated by the pilot or from the sensors. It consists of a

number of theories defining the types of the states and events, for describing the possible transitions and for initialising the system and stating the invariants that must be true for a valid specification. The states are specified as a tuple of variables contained in the abstract data type RECORD which is equivalent to the amalgamation of all states in the RSDS specification. The specification that was generated from adopting the SCR modelling approach decomposed the specification by mode, rather than by event. Each of the sensors and actuators are described in isolation from each other. In this comparison we evaluate the specification modelled on statemachines, as we are already evaluating the SCR modelling approach.

The RSDS specification of the autopilot system resembles the PVS specification using the statemachine approach, but there are some significant differences. These are summarised in the following table:

	Comparison points	**RSDS**	**SCR**	**PVS**
Requirements analysis and specification development	*Formulation of specification*	DCFD, invariants & statemachines & set of guidelines. Development process partly automated. Also, models environmental assumptions.	Tabular notation & set of guidelines. Also, models environmental assumptions.	PVS language - theories. Very expressive. No guidelines for modelling. Uses data abstraction technique.
	Level of abstraction	High & low Coarse-grain & Fine-grain	Low	High
Design	*System decomposition*	Horizontal decomposition	Always applies mode decomposition. Many tables, hard to follow.	No decomposition applied, any can be used.
Validation	*Simulation & checks*	Simulation, (animation in B) inconsistency & incompleteness checks	Simulation, inconsistency & incompleteness checks	Validation by verification only
Verification	*Proof support*	B theorem prover & SMV	PVS & SPIN	PVS
Abstraction techniques	*Model checking*	Optimisation via syntax and decomposition approaches	Two reduction principles	N/A
Tool support	*Tools used*	RSDS tool, SMV and BToolkit	SCR tool, SPIN PVS with TAME as front end	PVS theorem prover

In the following sections, we consider in more detail the similarities and differences among the specifications of the autopilot system. The results are organised according to the aspects singularised for the purpose of comparison.

Requirements analysis and specification development

The RSDS and SCR method provide a set of guidelines for applying a particular modelling approach for developing specifications. Some might consider this as being restrictive, even though their aim is to focus on a specific application domain enriching the development process with experience obtained from this domain. RSDS in particular, aims to use this experience to automate the process for facilitating the development of similar systems. PVS, on the other hand, does not provide a set of guidelines for developing specifications because it is a verification system that supports systems from different application domains, and any modelling approach can be adopted for generating a specification in its language. This however, is not always an advantage as it is much more difficult to produce the specification and can be time consuming as we cannot assume that the developer is aware of the different modelling approaches.

The comprehensibility of the notation or language used for describing the specification is important as it aids validation. The SCR method provides a tabular notation that is not very intuitive and can get complicated when representing large systems. The variable dependency graph, used to show visually the relationship between the variables in SCR, suggests the need for a simple graphical view of the system. In [BH97a], this limitation is recognised and the tabular notation has been improved by reducing the number of tables for this simple example from nine to just five. The RSDS method uses statemachines, a familiar notation for engineers and can encourage communication between developers and non-software engineers, and it can be argued that they are more comprehensible than SCR's tabular notation. However, statemachines mostly for the controllers can get very complicated for large systems, but RSDS overcomes this limitation by automatically deriving the controller statemachines. The statemachines for the sensors and actuators are usually very simple. PVS provides a very rich specification language and uses abstract data types that are familiar to software engineers. Its language caters for the concise description of specifications but it is not very intuitive and difficult to communicate with people who have no or little experience with PVS.

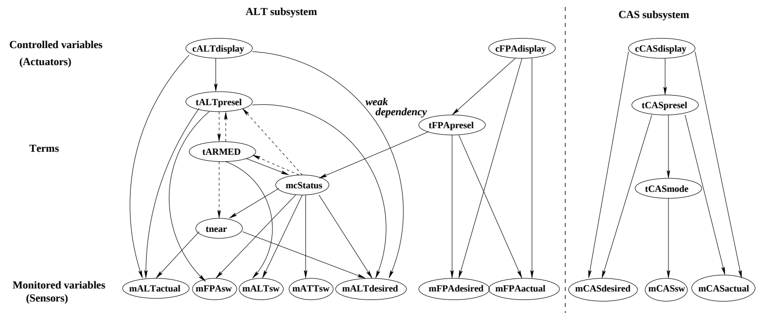


Figure 6.15: The SCR variable dependency graph.

When developing system specifications from their requirements, some of the features have not been finalised and a modelling language should support a specification at a more abstract level. The specific details of the features can be introduced once they are known. PVS and RSDS specify system at a more abstract level than SCR. In fact, RSDS provides two levels of abstraction: the coarse-grain view and the fine-grain view. The level of abstraction of the PVS model (that

implemented the autopilot using statemachines) is similar to the coarse-grain view, because all the state changes occur in a single step, i.e. the response to an event occurs in a step. The level of abstraction of the SCR model is similar to the fine-grain view, because the order in which the events and the state changes occur is specified. PVS can support refinement, although it is not needed for the autopilot system. SCR claims that the requirements will not be represented accurately in a more abstract model and can be misinterpreted by developers. During verification, they obtain an abstraction of the specification in order to verify a smaller model.

The authors in [But96] have not thought to model the physical environmental constraints imposed in PVS. They could be used, as in RSDS, for eliminating states that are never reached by adding them as assumptions when proving the properties. Both RSDS and SCR specifications include these constraints and use them for fault detection.

Design

The SCR method always assumes a decomposition by control mode for all of the example systems that have been specified ([BH97a], [HB00]). Butler [But96] pointed out that mode decomposition was not suitable for specifying the autopilot system as there are many interactions between the modes that lead to complicated tables. The tabular specification of the autopilot originally consisted of nine tables and later optimised to five, while there are only seven simple statemachine modules in RSDS. The problem that arises with a specification with many tables is that it is difficult to follow one action. Moreover, some systems cannot always be expressed in terms of system modes. For example, in the Fault-Tolerant Production Cell [Lot96], the only possible modes that could be identified are: normal, failed and recovering. These provide little insight on how to specify the system and will possibly lead to a large and complex tabular specification for each mode.

RSDS, guided by the dependencies denoted by the invariants, chose horizontal decomposition as a suitable approach even though mode decomposition was an option. This choice was made because the variables for CAS do not depend on any other variables in the system as illustrated by the variable dependency graph of SCR in Figure 6.15. RSDS is equipped with a number of decomposition approaches as we have already mentioned. PVS does not explicitly provide decomposition techniques, but if these are known they can be used to produce the specification. PVS has already shown how it can specify the autopilot using two different approaches, one that applies mode decomposition and another that is event-driven. It must be possible for PVS to apply all of RSDS's decomposition approaches, and many others too.

Validation

In SCR the system is validated in two ways: first a consistency checker is executed that checks for proper syntax, missing cases, nondeterminism and type correctness, and then a simulator can be used to execute the requirements and discuss the results with the client to see if the system behaves as intended. For the autopilot system, SCR and RSDS identified more cases of inconsistencies than PVS. Moreover, SCR insists upon certain information in order to formulate the specification and this is also the case for RSDS. For example, in RSDS the statemachines require the definition of initial states and SCR specifications require these too.

In RSDS inconsistencies or incompleteness are detected by analysing the invariants and this can be automated. Also, the RSDS specification can be automatically translated into B and the B animator can be used to execute the specification.

PVS relies on human inspection for detecting inconsistencies and incompleteness as well as for determining whether the specification satisfies the intended behaviour. Consistency can only be checked via proof in PVS but its success depends on the properties devised.

Verification

Verification is PVS's major strength. The specification must satisfy the properties that are presumed to be true of the system. These properties (or system invariants) are extracted from the requirements as for RSDS and SCR specifications. A predicate is used to describe the union of the system invariants and commands are applied for proving that this property is satisfied. For the autopilot, a single command (GRIND) is only used to prove by induction and its performance is compared to those of model checkers. There are different ways in which the proof can be formulated and these require ingenuity and experience for deciding which commands to use. The version of PVS used to specify the autopilot did not support the verification of temporal properties and thus these were never mentioned. However, model checking has been integrated into PVS so that temporal properties for finite state systems can be verified.

SCR relies on PVS and the model checker SPIN for verification. For the autopilot system, it identified two additional properties than the PVS specification described. One of these was a temporal property, known as a transition invariant in SCR that describes the system in terms of the next step, similar to action invariants in RSDS. The other property is known as a state invariant for determining the reachable states in the system, similar to static invariants in RSDS. PVS is used by SCR in the same way that B is used in RSDS. The main difference is that in B the modular structure of RSDS is preserved so that proof can be localised and the specification can be refined further in order to produce code automatically. Although PVS is a very comprehensive language, B is a more comprehensible language for RSDS users as the coarse-grain view corresponds to abstract B machines and the fine-grain view to B implementations. SCR has had to provide their own interface TAME for understanding PVS comments. The temporal property is proven automatically using SPIN. RSDS uses SMV for verifying temporal properties and identified further properties for reachability. These properties are aimed at ensuring that an undesired state is never reached. We do not compare the resources as it would be unfair as the specification of the machines and the performance of the model checkers used have improved remarkably since the papers were written. The translation to SMV from RSDS is modular and properties can be proven locally.

An additional invariant is defined in PVS than in RSDS because of the way in which the variables are defined for the modes. A variable is defined for each mode and is either: enganged, off or armed and therefore invariants are needed to ensure that only one of the three modes (ATT, FPA, ALT) is engaged at any one time, independently from the CAS mode. In RSDS, the decomposition strategy chosen, ensures that this invariant is true and thus does not need to be explicitly proven.

Abstraction techniques

Since both SCR and RSDS use model checking for verification, it is reasonable to assume that some kind of abstraction techniques is applied to both in order to reduce the state space of the model. This allows the verification of some system properties that would otherwise be impossible. SCR uses the two reductions principles (one for eliminating irrelevant entities and the other applies abstraction of monitored variables) of [BH97b, BH99] to obtain an abstract specification which is translated to the language of SPIN. If a counter-example is produced, this is run with the simulator that highlights the cause of the violation in SCR. A disadvantage of using SPIN is that it mostly produces long counter-examples, while SMV always guarantees to produce the shortest path. Moreover, the language of SPIN (Promela) does not cater for the expression of both "new" and "old" values of variables. To overcome this, SCR [BH99] explicitly represents each value with separate variables, thus increasing the state space. This problem is not evident in SMV.

The RSDS specification is translated into SMV for verification. For the autopilot system, a number of optimisations have been applied on the statemachines when generating the controller that contribute to the reduction of the state space. Eminently, the decomposition approaches aid verification. For example, the autopilot is decomposed horizontally and thus can be translated into separate SMV programs for each subsystem to verify the properties locally.

PVS is not impaired by the state space explosion problem. In the development of the autopilot system, they introduce a data abstraction technique for abstracting the details of the possible values of variable. For example, the range of values for the altitude sensor is: *at, near, away* instead of the integer values and similarly for the displays whose possible values are: *current and desired*. This technique corresponds to the second reduction principle in SCR for removing detailed monitored variables. We have applied this abstraction technique to the RSDS specification, as this facilitates the translation into SMV as SMV is not efficient at representing integers.

The SCR specification of the autopilot consists of twenty variables (including monitored, controlled, mode class and terms), compared to only ten in the PVS specification, one for each mode, display and altitude sensor, current event and current state. The SMV model for the RSDS specification consists of only nine variables as the modes are represented by two variables. The state space of the SMV model for RSDS specification and the PVS specification is the same.

Tool support

Both SCR and RSDS have some tool support for supporting the development of a specification from its requirements. They have embodied a collection of tools and provide translations to the languages of the various tools. SCR aims to seamlessly integrate these tools into the SCR toolkit and it provides interfaces for interpreting the results. The advantages lie in that developers minimise the amount of new languages and tools they need to learn.

RSDS provides automated translations whose correctness has been verified but does not provide an interface for interpreting the results obtained from the tools. It provides a number of templates to facilitate development and aims to automate as much of the development process as possible, more than SCR, for example the automatic derivation of the controller statemachine from the sensors. The SCR tool has been used extensively in industry [Hei02] while the RSDS tool is still in its early stages of development. PVS is also a well-established interactive verification system to support proof of systems specified in their rich language. It provides no support for the development of specifications.

6.3 Related Work

In [DT96], a method for developing embedded software systems by using B is introduced that can be compared to RSDS. The set of requirements of a system are expressed at an abstract level formally in B and then by applying a number of refinement steps, the specification is structured and expressed in the form required for translation into code. The disadvantage of introducing all these refinement steps in B is that refinement is difficult to produce and depends on the skills of the developers (the process cannot be easily automated). This can introduce errors into the specification because of human interaction, and can be time consuming. The advantage of this method is that difficult proofs can be decomposed into a number of smaller proofs that can be easily proven. RSDS structures the specification at the DCFD and statemachine stage which is easier and does not introduce any refinement steps in B.

RSML [LHHR94] is a requirements specification language that uses a modified notation of statemachines ([HN96], [PS91]) as well as other techniques like AND/OR tables, to specify each component. It is limited with respect to the structuring mechanisms that it provides: like SCR, it only decomposes the controller by modes. In [CAB$^+$98], RSML specifications are translated into SMV for model checking. In particular, they translated a portion of the Traffic Alert and Collision Avoidance System (TCAS II) whose state space size is 1.4×10^{65}, which is much larger than that of the autopilot specification. The translation presented does not represent the modular structure of statemachines in SMV, i.e. no SMV modules are defined. However, they do translate more complex states (not in the semantics of RSDS) such as nested OR states and nested AND states, but by flattening them first. Also, they translate timing constraints which are not defined in RSDS.

DOVE (Design Oriented Verification and Evaluation) [Dov] is a tool with a graphical editor for drawing statemachines, an animator for simulating the execution and a prover for verifying the critical properties. The tool does not provide structuring mechanisms and guidelines and does not check the liveness property because it represents only finite configurations. However, it can check only the progress property i.e. a state can be reached after a number of transitions.

iState [SZ01] is a tool for translating statecharts (a restricted version of UML statecharts) into programming languages, namely Pascal, Java and AMN, for validation. Its not a method, as it does not provide a set of guidelines or a systematic way in which to specify systems, so it cannot be directly compared to RSDS, SCR, and RSML. However, the translation to AMN is comparable to the RSDS translation to AMN. They both take an event-centric translation approach, meaning that the main structure of the code is that of events. The translation from RSDS to SMV is state-centric, meaning that the main structure of the code is that of states. The statechart notation in [SZ01] differs from RSDS statemachines, for example. it allows for states (including initial) to have several outgoing transitions. Moreover, a proof of correctness for the translations in [SZ01] has not been given.

STATEMATE is a tool for specifying and analysing systems. Its modelling language is a statechart variant. In [CH00], a translation is described from STATEMATE statecharts to SMV, that is modular, i.e. it preserves the hierarchy structure of the statecharts in the SMV code generated. Although RSDS translations are also modular, the STATEMATE statecharts are more expressive than RSDS statemachines as they contain nested AND and OR states. The translation in [CH00] produces large and complicated SMV code, containing boolean variable definitions for each event and generated event, and the SMV modules contain a large number of parameters for passing these events around, both of which increase the state space significantly.

The algorithm for performing this translation [CH00] has been used in the IFADIS toolkit [LH02], whose aim is to support the analysis of dependable interactive systems by model checking. It is aimed at practising engineers in the avionics industry that are not familiar with SMV. A statechart model of the system is first developed using the STATEMATE toolkit [Sta]. Then this model is imported into the IFADIS toolkit and automatically translated into SMV using [CH00]. The engineer then chooses the kind of properties to be analysed from either property specification patterns or a list of templates for usability properties, and instantiates them with appropriate values from the system model. The properties are checked by SMV and the results are presesnted in an enhanced tabular view (i.e. detailing the steps of the trace for counter-examples). RSDS differs from IFADIS in two ways. It provides a specific statemachine notation (rather than using STATEMATE etc.) that supports the specification of reactive systems in a systematic way (for example, it provides decomposition approaches, type of modules to be used). Secondly, if counter-examples are produced from SMV, they can be easily corrected on the RSDS model. IFADIS does not present a direct mapping of the counter-example onto the statechart model, but just presents the counter-example in an accessible way.

6.4 Summary

The control panel of a simple autopilot system was developed using RSDS for the purpose of comparison with SCR and PVS that have previously developed it. An overview of the issues discussed in the comparison are given in the following table.

	Comparison points	**RSDS**	**SCR**	**PVS**
Requirements analysis and specification development	*Formulation of specification*	DCFD, invariants & statemachines	Tabular notation	PVS language
	Level of abstraction	High & low	Low	High
Design	*System decomposition*	Horizontal decomposition	Mode decomposition	No decomposition
Validation	*Simulation & checks*	Simulation, inconsistency & incompleteness checks	Simulation, inconsistency & incompleteness checks	Validation by verification only
Verification	*Proof support*	B theorem prover & SMV	PVS & SPIN	PVS
Abstraction techniques	*Model checking*	Optimisation via syntax and decomposition approaches	Two reduction principles	N/A
Tool support	*Tools used*	RSDS tool, SMV and BToolkit	SCR tool, SPIN PVS with TAME as front end	PVS theorem prover

We conclude from this experience that a method supported by a collection of tools is far better at developing specifications of reactive systems rather than just a verification system. The development method should be flexible but provide guidelines or templates of design approaches to facilitate development. It is clear that both model checking and theorem proving are required for verification since model checking is not suitable for modelling the environment.

CHAPTER 7

Model Checking RSDS/UML Specifications

The RSDS/UML method supports the development of systems in an object oriented way. At present, we focus on specifying reactive systems (in particular control systems) with RSDS/UML but we aim to widen the application domain to include financially-critical systems[1]. RSDS/UML uses a restricted subset of UML notations (class diagrams, object diagrams and statecharts) for specifying the behaviour of systems. The advantage of using UML is that it provides notations that are used in mainstream software development and are thus familiar to most software engineers. In addition, UML is scalable for large systems. However, UML lacks precise semantics which consequently makes formal verification of UML specifications impossible. Therefore, RSDS/UML provides formal semantics for this subset of UML and as a result the properties of RSDS/UML specification can be verified. The semantics for the class diagrams and object diagrams are more or less standard, while the semantics for the statecharts correspond to an object oriented variant of SRS statemachines with coarse-grain and fine-grain views.

The problem that we address in this chapter is how to provide verification support for control systems developed with RSDS/UML. Since model checking was used successfully for verifying the temporal properties of RSDS specifications, we apply model checking to RSDS/UML specifications as well. Therefore, we define translation rules for mapping elements of an RSDS/UML specification into elements of the SMV input language. We also discuss how suitable SMV is for expressing object oriented concepts as SMV does not have high-level data types or constructs that are commonly used for specifying software.

By providing verification support for RSDS/UML specifications, we not only provide an object oriented formal approach to developing safety-critical systems, but also provide a way for verifying UML specifications. Several tools and methods already exist that provide verification support for UML system models. However, we find that they have the following shortcomings: they provide translations to the input language for a model checker, theorem prover or other formal methods verification system, but they do not provide a proof of correctness with respect to the UML

[1]Financially-critical systems are systems whose failure may result in a devastating economic loss.

semantics. Furthermore, they either verify the static (e.g class diagrams) or dynamic aspects (e.g. statecharts) of UML, but not both. For class diagrams, the constraints supported do not reason over the correctness of the dynamic creation or deletion of class instances or on the dynamic creation or deletion of associations between classes. We address these issues in our translation.

In this chapter we first present the subset of UML notations that constitute an RSDS/UML specification and formally describe their semantics. We consider to what extent the object oriented constructs can be represented in the SMV language and describe numerous translation rules for mapping elements of coarse-grain RSDS/UML specifications to elements of the SMV language. To ensure that the translation preserves the semantics of RSDS/UML, we formally prove its correctness. RSDS/UML is currently being revamped, so we discuss how the translation can be adapted and we also test the efficiency of particular representations that we chose to implement in SMV. Finally, we evaluate our work with respect to other approaches.

7.1 RSDS/UML Specifications

RSDS/UML uses a restricted subset of UML class diagram, object diagram and statechart notation to specify the static and dynamic behaviour of control systems. These models present different views of a system. A class diagram is used in the place of the DCFD to model the relationship of system components with classes and associations. The instances of classes and associations are visualised using object diagrams. Invariants are defined in the class diagram using LOCA (Logic of Objects, Constraints and Associations) [Lan05], asubset of OCL. The dynamic behaviour of instances is described using statemachines by presenting their internal states and their reaction to external events. The decomposition techniques can be applied to RSDS/UML specifications as well, however we do not consider these techniques in the work defined in this chapter.

7.1.1 Class Diagrams

A class diagram is composed of classes and associations. Classes are encapsulations of state and operations that model its behaviour by updating and querying the state. They are depicted as rectangles that enclose definitions of their state and operations. On the top RHS corner of a class there is usually an integer, we call the class multiplicity, that indicates the maximum number of instances that can be created for that class. These instances can be created and destroyed dynamically (at run-time) as needed. Associations model the interaction between the instances of the classes involved, where this interaction results in the instance of one class requiring the interaction of another in order to perform its operations. They are depicted as solid lines between two classes. Cardinality can be defined on both ends of an association that indicate the number of instances of each class that are related. If no cardinality is provided, it is assumed that it is 1, i.e. one instance of a class is related to one instance of the other class that is associated.

For control systems, RSDS/UML represents each controller and actuator by a separate class and their relationships are represented by associations. These relationships could not be modelled with the DCFD and neither could the class state and operations. Figure 7.1 illustrates a class diagram produced for the most basic type of control system, that is one with a single controller

and actuator. The controller class has only one instance and it always exists. It is derived from the sensors by combining their states and represents the sensor events and transitions. Actuator classes have a finite number of possible instances, whereby the maximum is expressed by the class multiplicity. We explicitly define the class multiplicity because class instances representing control system components correspond exactly to physical components and any omission of instances could lead to a serious violation of properties. Actuator classes are never associated with each other.

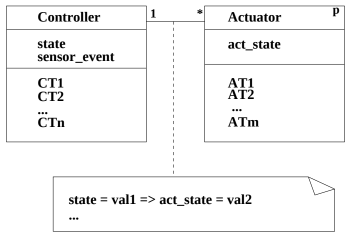


Figure 7.1: A typical class diagram for a basic control system developed using RSDS/UML.

7.1.2 Invariants

In UML class diagrams, constraints are used to impose restrictions on the system and/or the environment and are expressed in LOCA [Lan05, LCA02d], a subset of OCL 2.0 specification notation. The syntax for this subset is illustrated in Table 7.1. LOCA uses the OCL types: *OclAny* that is a supertype of booleans, numerics and strings, and the parameterised types: *Collection* $\langle T \rangle$, *Set* $\langle T \rangle$ and *Sequence* $\langle T \rangle$. We further restrict this subset for RSDS/UML to include only discrete integer values (i.e. no integers of type Real). The mapping of LOCA to syntax to the standard OCL syntax is illustrated in Table 7.2 and e' denotes the translation of e. The numeric opererators $/, +, *, <, >, <=, >=$ on numeric expressions have the same form in both OCL and LOCA. For this version of RSDS/UML, we only use the following LOCA syntax to describe invariants: *OclAny*, *Integer*, *Boolean*, *Sequence* and *Other*.

Usually, constraints are attached to classes as class invariants, or to operations as pre and post conditions. Invariants are true before and after an operation, i.e. they can be temporarily broken during an operation. The meta-model definition of UML allows constraints to be attached to any model element. RSDS/UML adopts this approach and allows constraints to be attached, not only to classes and operations, but to associations as well. The advantage of this approach is that one can specify how the states of attributes of one object relate to those of other objects in an abstract way, without defining explicitly which classes are responsible for enforcing the constraints. The invariants can be described as constraints on states of specific objects or can be generalised to apply to all objects (see *Other* in Table 7.2 for LOCA syntax). An example of an invariant described as the latter is illustrated in Figure 7.1: $state = val1 \Rightarrow act\_state = val2$ that is interpreted as:

$$\forall x, y \cdot (x, y) \in Controller\_Actuator \ \wedge \ x.state = val1 \ \Rightarrow \ y.act\_state = val2$$

$\langle$ *value* $\rangle$	::=	$\langle$ *ident* $\rangle \mid \langle$ *number* $\rangle \mid$ $\langle$ *string* $\rangle \mid \langle$ *boolean* $\rangle$
$\langle$ *objectref* $\rangle$	::=	$\langle$ *ident* $\rangle \mid$ $\langle$ objectref$\rangle.\langle$ ident$\rangle \mid$ $\langle$ objectref$\rangle \mid (\langle$ expression$\rangle)$
$\langle$ *arrayref* $\rangle$	::=	$\langle$ *objectref* $\rangle \mid$ $\langle$ *objectref* $\rangle [\langle$ *value* $\rangle]$
$\langle$ *factor* $\rangle$	::=	$\langle$ *value* $\rangle \mid \{\langle$ *valueseq* $\rangle\} \mid$ *Sequence* $\{\langle$ *valueseq* $\rangle\} \mid$ $\langle$ *objectref* $\rangle \mid \langle$ *arrayref* $\rangle \mid$ $\langle$ *factor* $\rangle$ op1 $\langle$ *factor* $\rangle$
$\langle$ *expression1* $\rangle$	::=	$\langle$ *factor* $\rangle$ op2 $\langle$ *factor* $\rangle$
$\langle$ *expression* $\rangle$	::=	$\langle$ *expression1* $\rangle \mid$ $(\langle$ *expression* $\rangle) \mid$ $\langle$ *expression1* $\rangle$ op3 $\langle$ *expression* $\rangle$
$\langle$ *invariant* $\rangle$	::=	$\langle$ *expression* $\rangle \mid$ $\langle$ *expression* $\rangle \Rightarrow \langle$ *expression* $\rangle$
valueseq		is a comma-separated sequence of values
op1		is a factor-level operator such as $+, -, *, /, \frown, /\backslash$or$/\backslash$
op2		is a comparator: $=, /, =, >, >=, <, <=, :, / :, <:$ or$/ <:$
op3		is $\&, or$

Table 7.1: LOCA syntax from [Lan05].

	LOCA	**OCL**	**Note**
OclAny			
	$x = y$	$x' = y'$	
	$x/ = y$	$x' <> y'$	
Integer			
	$x + y$	$x' + y'$	Likewise for $-, /, *$
	$x < y$	$x' < y'$	Likewise for $>, <=, >=$
	$\{n_1, ..., n_m\}.max$	$n'_1.max(n'_2.max(...n'_m...))$	Numeric expressions $n_1, ..., n_m$
	$\{n_1, ..., n_m\}.min$	$n'_1.min(n'_2.min(...n'_m...))$	Numeric expressions $n_1, ..., n_m$
	$\{n_1, ..., n_m\}.sum$	$n'_1 + ... + n'_m$	No duplicates in $n_1, ..., n_m$
	$x\ div\ y$	$x'.div(y')$	Likewise for *mod*
String			
	$str.size$	$str'.size()$	
	str1 + str2	*str1'.concat(str2')*	
Boolean			
	$b1\ \&\ b2$	$b1'\ \&\ b2'$	
	$b1\ or\ b2$	$b1'\ or\ b2'$	
	$b1 \Rightarrow b2$	$b1'\ implies\ b2'$	
Collection			
	$c.size$	$c' \rightarrow size$	
	$x : c$	$c' \rightarrow includes(x')$	
	$x/ : c$	$c' \rightarrow excludes(x')$	
	$c1 <: c2$	$c2' \rightarrow includesAll(cl')$	
	coll — (P)	$coll' \rightarrow select(P')$	*coll* is a collection
	$c.sum$	$c \rightarrow sum()$	
Set			
	$s = \{\}$	$s' \rightarrow isEmpty()$	
	$\{x_1, ..., x_m\}$	$Set\{x'_1, ..., x'_m\}$	
	$set \setminus / set2$	$set' \rightarrow union(set2')$	
	$set /\backslash set2$	$set' \rightarrow intersection(set2')$	
	$set \backslash/ \{elem\}$	$set' \rightarrow including(elem')$	
	set - $\{elem\}$	$set' \rightarrow excluding(elem')$	
	set1 - *set2*	$set1' - set2'$	
Sequence			
	$Sequence\{x_1, ..., x_m\}$	$Sequence\{x_1, ..., x_m\}$	
	$seq[i]$	$seq' \rightarrow at(i')$	
	$seq.asSet$	$seq' \rightarrow asSet()$	
	$seq1 \frown seq2$	$seq1' \rightarrow union(seq2')$	
Other			
	C	C.allInstances()	
	self	self	

Table 7.2: Mapping of LOCA syntax to OCL from [Lan05].

where *Controller_Actuator* is the set of pairs of objects in the association between *Controller* and *Actuator*.

7.1.3 Object Diagram

Object diagrams represent instances of classes and associations depicted in a class diagram. There are many object diagrams for each class diagram. RSDS/UML specifications have finite number of possible object diagrams as the number of class instances is finite.

Let us suppose that the class diagram in Figure 7.1 represents a simple reactive system, that consists of two sensors (a switch and a fluid detector) and two actuators (valves) that are instances of class Actuator (therefore, p = 2). The aim of the system is that when the switch is pressed (state of switch = on), the valves must be set to open in order to allow some fluid to pass through. The fluid detector is used in order to determine when to close the valves and for fault detection. A possible object diagram is illustrated in Figure 7.2, showing the initial values for this system.

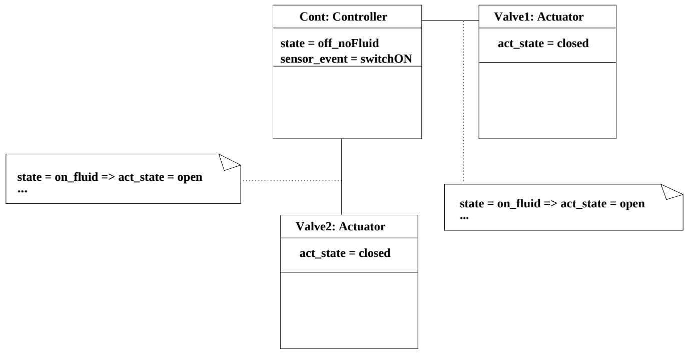


Figure 7.2: A possible object diagram for the class diagram in Figure 7.1.

7.1.4 Statemachines

A statemachine is produced for each class in the class diagram, representing the dynamic behaviour of the class. The statemachine notation employed is an object oriented variant of SRS statemachines notation used for describing RSDS specifications. It can also be considered as a restricted subset of UML statechart diagrams. The UML statecharts are restricted in the following ways:

1. Deferred events are not considered.
2. No nested states are represented. Hence, no history states (where a superstate can recall its last active substate) are modelled or interlevel transitions.
3. We do not allow non-deterministic transitions.

4. We do not deal with object oriented issues represented in UML statecharts such as inheritance or sub-behaviour in statecharts.

Moreover, RSDS/UML statemachines have two semantic views: the coarse-grain and the fine-grain view. Both views do however model the "run to completion" semantics of UML statecharts.

7.1.5 The gas burner system

Recall the gas burner system specified in section 3.5. It can be redesigned to contain a class Controller, a class Igniter and a class Valve. The classes Controller and Igniter have only one instance each, while Valve has two instances, one for the air valve and another for the gas valve. This design is conceptually coherent as the air and gas valve are similar in form and function. The invariants 3.1 to 3.7 (given in section 3.5) are attached to the associations between the classes. Figure 7.3 illustrates the class diagram for the gas burner with the invariants attached to associations. The class Controller is responsible for enforcing the constraints.

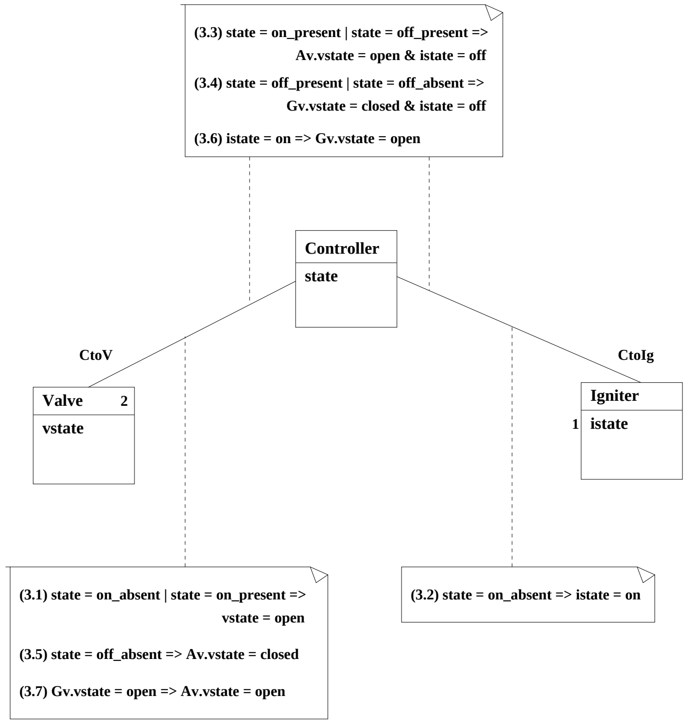

(3.3) state = on_present | state = off_present => Av.vstate = open & istate = off
(3.4) state = off_present | state = off_absent => Gv.vstate = closed & istate = off
(3.6) istate = on => Gv.vstate = open
Controller
state
CtoV
CtoIg
Valve 2
vstate
Igniter
1 istate
(3.1) state = on_absent | state = on_present => vstate = open
(3.5) state = off_absent => Av.vstate = closed
(3.7) Gv.vstate = open => Av.vstate = open
(3.2) state = on_absent => istate = on

Figure 7.3: The class diagram for the gas burner system.

7.2 RSDS/UML Semantic Foundations

We have described the expected form of the RSDS/UML system specification by describing some syntactic restrictions. In this section we describe the precise semantics for RSDS/UML specifications as first order CTL theories as published in [LCA02a, LCA02c]. We use first order CTL as we want to define the meaning of all instances of classes. The semantics described are more general than the expected form of RSDS/UML specification as we want to gradually evolve RSDS/UML to support more UML features without losing the benefits of using the control system structure for automating as much of the control algorithm as possible. These syntactic restrictions are gradually relaxed as RSDS/UML is applied to systems from different application domains. In the meantime, these syntactic restrictions can be considered as guidelines that assist the developer in modelling the system requirements simply and in a familiar notation.

7.2.1 Class Diagrams

A class diagram CD_{Sys} for a control system describes a set $\{C_1, ..., C_n\}$ of classes, and associations $\{r_1, ..., r_n\}$. Each association has a set of association ends where each end has a corresponding class and cardinality. The cardinality determines how many objects of a class are associated at a time. It is interpreted as a set of numbers, for example:

i) 1 is $\{1\}$,
ii) $0...p$ is $\{0, .., p\}$,
iii) $*$ is $\mathbb{N}$,
iv) $p...q$ is $\{p, .., q\}$. If no cardinality is provided, it is assumed to be 1. On the top RHS of a class, there is a cardinality that represents the maximum number of instances that can be created for this class. The cardinality is interpreted as a natural number (constant in $\mathbb{N}$).

A first order CTL theory Δ_{CD} is defined for a class diagram. This consists of the following type symbols, attributes and action symbols:

- Type symbols $@C$ for each class C that refers to all possible instances that can exist for class C, $@D_i$ for each end point i of associations, and for standard primitive types *bool*, *int*, etc;
- Attributes

  $$\overline{C} : \mathbb{F}(@C)$$

  for the set of all existing instances of each class C , and $\overline{r} : \mathbb{F}(@D_1 \times ... \times @D_k)$ for the set of all existing instances of each association r, with end points at classes D_1 to D_k;
- Attributes

  $$att : @C \to T$$

  for each attribute $att : T$ of class C where T is the attribute type. We may write $x.att$ for $att(x)$ to emphasise that these represent variables at the instance level. Moreover, the mode of C (that is the current state of its corresponding statemachine Sm) is included as an attribute $sm : @C \to States_{Sm}$;

- An attribute

  $$m_C : \mathbb{N}$$

  representing the class multiplicity for each class C.
- Action symbols

  $$op : @C \times T_1 \times ... \times T_p \to T$$

  for each operation $op(T_1, ..., T_p) : T$ of class C where $T_1, ..., T_p$ are the parameters of the operation, and T is the return type. The syntax $x.op(v_1, ..., v_p)$ can be used for $op(x, v_1, ..., v_p)$;
- Actions for creating and deleting instances:

  $$new_C : @C \to$$
  $$kill_C : @C \to$$
- Actions for linking and unlinking tuples of instances in relationships:

  $$link_r : @D_1 \times ... \times @D_k \to$$
  $$unlink_r : @D_1 \times ... \times @D_k \to$$
- Pseudo-attributes

  $$bs : @A \to \mathbb{F}(@B)$$
  $$as : @B \to \mathbb{F}(@A)$$

  for each binary relation r between classes A and B. The notations $y \in bs(x)$ and $x \in as(y)$ abbreviate $(x, y) \in r$.

The theory Δ_{CD} also consists of the following axioms:

RU1 Axioms defining the initial state of attributes:

$$new_C(x) \Rightarrow AX(att(x) = val)$$

where val is the initial value of att specified in the class diagram;

RU2 Axioms

$$AG(\overline{r} \subseteq \overline{D_1} \times ... \times \overline{D_k})$$

for each association r, with end points at classes D_1 to D_k;

RU3 Axioms defining the allowed cardinality of an association: if c_k is the cardinality of association r at end k, expressed as a set c_k of elements of $\mathbb{N}$, then for each particular tuple $(x_1, ..., x_{k-1}) \in \overline{D_1} \times ... \times \overline{D_{k-1}}$ at the other ends, the number of elements $x \in \overline{D_k}$ such that $(x_1, ..., x_{k-1}, x) \in \overline{r}$ must be in c_k. Similarly for each of the indexes;

RU4 Axioms for creating and deleting instances:

$$AG(\forall X : \mathbb{F}(@C) \cdot \overline{C} = X \ \wedge \ new_C(x)$$
$$\Rightarrow AX(\overline{C} = X \ \cup \ \{x\}))$$

$$AG(\forall X : \mathbb{F}(@C) \cdot \overline{C} = X \ \wedge \ kill_C(x)$$
$$\Rightarrow AX(\overline{C} = X \ - \ \{x\}))$$

for each association between D_1 to D_k.

RU5 The axioms for linking and unlinking tuples of instances in relationships are:

$$AG(\forall X : \mathbb{F}(@D_1 \times ... \times @D_k) \cdot \overline{r} = X \ \wedge \ link_C(x_1, ..., x_k) \\ \Rightarrow AX(\overline{r} = X \ \cup \ \{(x_1, ..., x_k)\}))$$

$$AG(\forall X : \mathbb{F}(@D_1 \times ... \times @D_k) \cdot \overline{r} = X \ \wedge \ unlink_C(x_1, ..., x_k) \\ \Rightarrow AX(\overline{r} = X \ - \ \{(x_1, ..., x_k)\}))$$

RU6 The axiom:

$$card(\overline{C}) \leq m_C$$

asserts that the number of existing instances of class C must be less than or equal to the class multiplicity. Similarly for all classes;

RU7 Axioms $AG(@E \subseteq @C)$ and $AG(\overline{E} \subseteq \overline{C})$ if E inherits C;

RU8 Locality axioms, stating that for each class C and attribute $att : T$ of C where the op_h are all the operations of C, if no operations occur in C then the attribute does not change:

$$AG(\forall att : @C \rightarrow T \cdot \forall x : @C \cdot \forall x : @\forall v : T \cdot \ att(x) = v \ \wedge \\ \forall x_1 \cdot \ \neg op_1(x_1) \wedge ... \wedge \forall x_h \cdot \ \neg op_h(x_h) \Rightarrow AX(att = v))$$

THEORY Δ_{CD}

Attribute symbols:
$\overline{C} : \mathbb{F}(@C)$ for the set of existing instances of class C
$\overline{r} : \mathbb{F}(@D_1 \times ... \times @D_k)$ for the set of existing instances of each association r
$att : @C \rightarrow T$ for each attribute
$sm : @C \rightarrow States_{Sm}$ for the corresponding statemachine
$m_C : \mathbb{N}$ for the class multiplicity
$bs : @A \rightarrow \mathbb{F}(@B)$ pseudo-attribute for the binary relationships
$as : @B \rightarrow \mathbb{F}(@A)$ pseudo-attribute for the binary relationships

Action symbols:
$op : @C \times T_1 \times ... \times T_p \rightarrow T$ for each operation
$new_C : @C \rightarrow$ for creating an instance of a class
$kill_C : @C \rightarrow$ for deleting an instance of a class
$link_r : @D_1 \times ... \times @D_k \rightarrow$ for linking tuples of instances in relationships
$unlink_r : @D_1 \times ... \times @D_k \rightarrow$ for unlinking tuples of instances in relationships

Axioms:
[RU1]
[RU2]
[RU3]
[RU4]
[RU5]
[RU6]
[RU7]
[RU8]

7.2.2 Invariants

RSDS/UML invariants have the same form as that of RSDS invariants, that is, static, operational and temporal. The key difference is that the invariants are not just attached to the controller

component (corresponding to the controller class in RSDS/UML), they can be attached to any component class as well as to any association between the classes. Moreover, the invariants are given without quantifiers but these are implicit as they quantify over all instances of classes whose attributes are referred to in the invariants. This is made evident in the semantic interpretation of these constraints.

The invariants that are attached to classes describe constraints on attributes local to their objects. Formally, a constraint ϕ on a class C has the semantic interpretation:

$$AG(\forall x : \overline{C} \cdot \phi(x))$$

where x denotes a class instance and is a parameter of attribute att of C in $\phi(x)$. An example of an invariant ϕ is $att \leq 10$ that imposes a restriction on a local attribute. It is interpreted by:

$$AG(\forall x : \overline{C} \cdot att(x) \leq 10)$$

The invariants attached to associations consist of constraints of attributes of objects that are related to each other via that association. Formally, a constraint on an association r between classes D_1 to D_k is interpreted by

$$AG(\forall x_1 : \overline{D_1}; ...; x_k : \overline{D_k} \cdot (x_1, ..., x_k) \in \overline{r} \ \Rightarrow \ \phi(x_1, ..., x_k))$$

where each attribute att of D_i occurring in ϕ is replaced by $att(x_i)$ in $\phi(x_1, ..., x_k)$. For example, the gas burner invariant 3.2 in Figure 7.3 is interpreted as:

$$AG(\ \forall z : \overline{Ig}; \ c : \overline{Controller} \cdot (c, z) \in \overline{Controller\_Ignitor} \Rightarrow \\ (state(c) = on\_absent \ \Rightarrow \ istate(z) = on))$$

where Ig refers to the ignitor component. Ig and $Controller$ are the classes at the end points of the association $Controller\_Ignitor$.

7.2.3 Object Diagrams

An object diagram displays a set of objects of classes and a set of links between the objects that are instances of associations in the class diagram. For an object diagram OD we define a first order CTL theory Ω_{OD}. This consists of type symbols and attributes:

- Type symbols $@C$ for each class C and for the primitive types;
- Attributes $\overline{C} \cdot \mathbb{F}(@C)$ for each class C, and $\overline{r} : \mathbb{F}(@D_1 \times ... \times @D_k)$ for each association r, with end points at classes D_1 to D_k;
- Attributes

  $att : @C \rightarrow T$

  for each attribute $att : T$ of class C;

The theory Ω_{OD} also consists of the following axioms:

RU9 Axioms

$$AG(\overline{C} = \{x_1, ..., x_p\})$$

where the x_i are all the existing objects in the diagram declared to be of class C or of a subclass of C;

RU10 Axioms

$$AG(\overline{r} = \{t_1, ..., t_q\})$$

where the t_j are all the tuples of existing objects corresponding to the links in the diagram declared to be the instances of r;

RU11 Axioms $AG(att(obj) = val)$ for each attribute equation $att = val$ listed in object obj.

THEORY Ω_{OD}

Attribute symbols:
$\overline{C} \cdot \mathbb{F}(@C)$ for each class C
$\overline{r} : \mathbb{F}(@D_1 \times ... \times @D_k)$ for each association r
$att : @C \to T$ for each class attribute att

Axioms:
[RU9]
[RU10]
[RU11]

7.2.4 Statemachines

Every class in the class diagram has a statemachine associated to it. The definition of the statemachine is given in terms of functions and corresponds to that of the original RSDS defined in section 3.2. The generations of transitions in the controller statemachine are defined as:

$generations_C : Trans_C \to$
$(@CC \to seq(Events_{R1} \times @CR_1 \cup ... \cup Events_{R_p} \times @CR_p))$ where the set of system receivers $receivers_{Sys}(C) = \{R_1, ..., R_p\}$ and the CR_i are the classes corresponding to the R_i statemachines, CC is the class corresponding to C.

There are two distinct semantics for the RSDS statemachines: the coarse-grain and the fine-grain. The semantics are described in first order CTL theories and are more general than the semantics used to describe the original RSDS. This is because the controller statemachine is not necessarily an amalgamation of the system's sensors which means that the developer can define any SRS statemachine for it.

Coarse-grain semantics

The *external events* of an RSDS system Sys are the sensor events:

$Ext_{Sys} = \bigcup_{i:1...n} Events_{S_i}$ where the set of sensors of Sys is $Sen = \{S_1, ..., S_n\}$. All other events of Sys are *internal*.

A theory Γ_{Sm} is defined for each component statemachine Sm for a class C. This has a single attribute

$sm : @C \to States_{Sm}$

and it has action symbols

$act : @C \to$ for each element act of $Trans_{Sm} \cup Events_{Sm}$.

The axioms of Γ_{Sm} include:

RU12 The state-transition behaviour of Sm, for all $x : @C$:

$$AG(sm(x) = s \wedge G(x) \wedge \alpha(x) \;\Rightarrow\; tr(x))$$
$$AG(tr(x) \;\Rightarrow\; \alpha(x))$$
$$AG(sm(x) = s \wedge tr(x) \;\Rightarrow\; AX(sm(x) = t))$$

for each transition tr of Sm with source s, target t, guard G, and trigger event α.

RU13 That at most one transition of Sm can occur in a step:

$$AG(\forall x, y : \overline{C} \cdot \neg(tr(x) \wedge tr'(y)))$$

for each pair of distinct transitions of Sm. Similarly for events.

RU14 That a transition can only occur if Sm is in its source state:

$$AG(tr(x) \;\Rightarrow\; sm(x) = s)$$

where $s = source_{Sm}(tr)$.

RU15 Locality:

$$AG(\forall v : State_{Sm} \cdot sm(x) = v \wedge$$
$$\neg tr_1(x) \;\wedge\; ... \;\wedge\; \neg tr_h(x) \;\Rightarrow\; AX(sm(x) = v))$$

Notice that events may happen in a state from which there is no transition for the event. The state remains unchanged in this case.

THEORY Γ_{Sm}

Attribute symbols:
$sm : @C \to States_{Sm}$ for the state of the statemachine

Action symbols:
$act : @C \to$ for the generated events

Axioms:
[RU12]
[RU13]
[RU14]
[RU15]

At the system level, we define the theory Th_{Sys} of the complete RSDS/UML specification Sys as the union of class diagram theory and each of the statemachine theories, together with the following global axiom:

RU16 For a controller transition tr with generations ρ:

$$AG(tr(x) \;\Rightarrow\; \rho(1)(y_1) \wedge ... \wedge \rho(p)(y_p))$$

where p is the $size(generations(tr)(x))$, $\rho(i)$ is the action symbol corresponding to event $first((generations(tr)(x))(i))$, and $y_i = second((generations(tr)(x))(i))$.

THEORY Th_{Sys}

Attribute symbols:
All attributes of Δ_{CD}, the class diagram theory
All attributes of all Γ_{Sm} statemachine theories (without duplicates).

Action symbols:
All action symbols of Δ_{CD}, the class diagram theory
All action symbols of Γ_{Sm} statemachine theories (without duplicates) .

Axioms:
All axioms of Δ_{CD}, the class diagram theory,
All axioms from all the Γ_{Sm} statemachine theories, and
[**RU16**]

Fine-grain semantics

The fine-grain semantics of Sys consisting of sensors $S_1, ..., S_n$, controllers $C_1, ..., C_m$ and actuators $A_1, ..., A_p$ can be expressed in terms of statemachines of the components together with axioms on the attribute

$event\_queue : seq(EventToken)$ of Th_{Sys} which holds a sequence of tokens representing pending events. $EventToken$ is isomorphic to the set $Event_{Sys}$ of all events of the system:

$$\begin{aligned} Events_{Sys} = \; & \bigcup_{i:1..n} Events_{S_i} \times @X_i \cup \bigcup_{i:1..m} Events_{C_i} \times @Y_i \cup \\ & \bigcup_{i:1..p} Events_{A_i} \times @Z_i \end{aligned}$$

where X_i is the class corresponding to S_i, Y_i is the class corresponding to C_i and Z_i is the class corresponding to A_i. Let $name : Events_{Sys} \rightarrow EventToken$ be this semantic isomorphism.

The global axiom (**RU16** defined at the system level) of the coarse-grain semantics is replaced by the following axioms on the event queue:

RU17 The event queue is initially empty : **BEG** $\Rightarrow$ $event\_queue = []$

RU18 An external event α can only be responded to if the queue is empty:

$$AG(t(x) \;\Rightarrow\; event\_queue \;=\; [])$$

for each transition t with trigger event some $\alpha \in Ext_{Sys}$.

RU19 When an external event is accepted by the system, generations of the triggered transition become the new event queue. All these events should be internal:

$$AG(tr(x) \;\Rightarrow\; AX(event\_queue \;=\; \rho))$$

where $tr(x)$ is the transition triggered by $\alpha(x)$, and

$$\rho_1 = generations(tr)(x)$$

and $\rho_i = name(\rho(i))$ for each i. If several transitions in different components are triggered by $\alpha(x)$ then some interleaving of their generations becomes the new event queue. It does not matter which interleaving.

RU20 An internal event can only be processed if it is at the head of the queue:

$$AG(t(x) \;\Rightarrow\; event\_queue \neq [] \;\wedge\; name(\beta(x)) \;=\; event\_queue(1))$$

for each controller or actuator transition $t(x)$ with the trigger $\beta(x)$.

RU21 The head of the queue is removed when the corresponding event is processed, and replaced with some interleaving ρ of the generations of all transitions triggered by this occurrence of β:

$$\forall q.AG(t(x) \wedge q = event\_queue \;\Rightarrow\; AX(event\_queue = \rho \frown tail(q)))$$

This models the "run to completion" semantics of statecharts in UML.

THEORY Th_{FSys}

Attribute symbols:
All the attributes of the theory Th_{Sys} of the coarse-grain, and,
$event\_queue : seq(EventToken)$

Action symbols:
All the action symbols of the theory Th_{Sys} of the coarse-grain.
However, events are divided into external and internal events.

Axioms:
[**RU17**]
[**RU18**]
[**RU19**]
[**RU20**]
[**RU21**]

7.3 Model Checking RSDS/UML Specifications

RSDS/UML specifications should be analysed and verified. Verification can be performed by translating into B as shown in [LAC03, LCA04]. However, B cannot verify the systems temporal properties and therefore we apply model checking as with RSDS. The RSDS translations to SMV presented in Chapter 4 and 5 cannot be used for RSDS/UML specifications as the object oriented concepts are not covered. Therefore, we present a new set of translation rules and translation schemas for translating the coarse-grain specifications of RSDS/UML into SMV. The fine-grain translation is not finalised yet. The object oriented features included in the translation are: dynamic creation and deletion of class instances, dynamic creation and deletion of associations between instances of different classes, encapsulation, class multiplicity, class inheritance and cardinality on association ends.

In the following sections, we discuss the suitability of SMV for modelling object oriented systems and present the translation rules for the coarse-grain RSDS/UML semantic view. The object oriented model of the gas burner system is used to illustrate the translation.

7.3.1 RSDS/UML vs SMV

A crucial issue that we need to consider is how suitable SMV is for verifying object oriented systems. SMV was originally developed for verifying hardware, therefore its language is very low-level (as it is based on finite transition systems) and does not provide any high-level data types or constructs that are common in languages used for specifying software. Furthermore, there is no dynamic memory allocation available and the state space explosion problem bounds SMV by memory. On the other hand, object oriented notations introduce unbounded behaviour. Nevertheless, we are still interested in model checking RSDS/UML specifications as verification is automatic and helps us automate as much of the development process of RSDS/UML as possible.

We consider each object oriented feature of RSDS/UML that will be translated into SMV and evaluate SMV's modelling capabilities of these features.

- **Dynamic creation and deletion of class instances:** SMV can only represent finite system specifications and is unable to dynamically create or delete modules or variables in its language. Therefore, it must imitate this dynamic aspect by defining the maximum number of instances that can exist at any one time for a class in the model and assert its existence by recording that the instance is alive. When the instance is deleted, the SMV model must record that the instance no longer exists.
- **Dynamic creation and deletion of associations between instances of different classes:** For the same reason as for class instances, associations cannot be modelled dynamically in SMV and its behaviour must be imitated in the SMV model. Arrays are used in SMV for recording which objects (denoted by the index) are associated. Only binary associations are considered.
- **Encapsulation:** The SMV language has no concept of encapsulation. This is apparent from the fact that modules can directly access the instance variables of other modules by passing them or the modules they belong to as parameters. The passing of parameters, especially modules, should be kept to a minimum as it increases the state space. Moreover, since the translation will be automated and the user will not have to be able to understand the SMV code generated, we are not interested in preserving this concept in the SMV generated.
- **Class multiplicity:** Since SMV models finite systems only, the maximum number of objects for each class that can be created must be known before translating. Therefore, the class multiplicity provides the maximum number of objects created for a class. If no class multiplicity is provided, it is assumed to be 1. The class multiplicity must be finite and not too large (the states of SMV model is greater than 10^{20} and can't be handled by the tool).
- **Class inheritance:** There is no construct available in SMV for modelling inheritance. There are two ways in which inheritance can be implemented. The first way simply copies all the contents of the superclass module into the subclass, resulting in the subclass having direct access to the attributes and operations of the superclass. The second way models inheritance as an association between two classes where at the association end of the subclass the cardinality is 0..1 and at the association end of the superclass the cardinality is 1 (see Figure 7.6).
- **Cardinality on association ends:** The allowed cardinality of an association refers to the number of instances that can be related to each other at any one time. In RSDS/UML specifications, the only associations defined are between controllers and actuators. The cardinality must be finite in order to be translated into SMV. Therefore, when cardinality is * (meaning many), the class multiplicity can be used as an upper bound. There are some implicit conditions that must be maintained when creating and deleting objects that are associated with cardinality.

We conclude that general object oriented concepts cannot be modelled precisely using SMV. We propose several restrictions to the object oriented notation to ensure translation to a valid SMV model.

1. A class multiplicity must be provided for all classes. If no multiplicity is provided, we assume it is 1. Moreover, the multiplicities provided must be within a reasonable range as this affects the feasibility of model checking a system. The reasonable range will be smaller for larger systems.
2. Attributes of type integer must also be within a reasonable range as they will increase the state space immensely. The data abstraction technique proposed in [But96] used in the autopilot system (see Chapter 6) can be applied to obtain a small range.
3. If an association has multiplicity *, then for the translation we assume it to be $0...p$, where p is the number of instances of the class at the association end where it is defined.
4. All of the possible instances of classes are modelled in SMV. They are created and deleted by changing the value of their boolean variable **alive**.

In the following sections, we present in detail the translation rules that show how to generate SMV models for RSDS/UML specifications.

7.3.2 Coarse-grain translation

We define the coarse-grain translation with a number of translation rules that map elements in RSDS/UML specifications to elements in the SMV input language. An algorithm that automates the translation can be easily implemented from these rules. In addition, translation schemas are provided that delineate the form of the SMV modules produced.

We assume that the input to the translation is an RSDS/UML specification that conforms to the structure presented for control systems and to the restrictions of the object oriented notation. These should be checked by the RSDS tool. An RSDS/UML specification consists of a single class diagram that describes the static structure of the system under development, a number of object diagrams and a statemachine for each class in a class diagram. An example of a class is given in Figure 7.4 with a number of possible objects and its corresponding statemachine that describes its dynamic behaviour. Each system component is modelled as a class in a class diagram, with the controller class associated to every other class. The controller is assumed to be static, that is it always exists, and is automatically generated by the translation algorithm (no details of the controller need to be given by the developer). As with the original RSDS derivation of the controller, we assume that the controller state is a combination of sensor states and that it receives all of the sensor events.

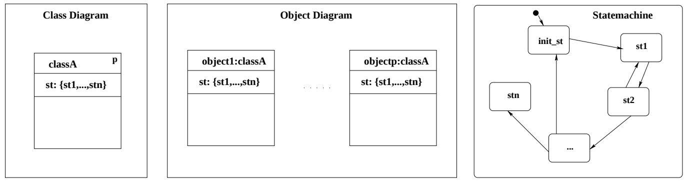


Figure 7.4: The different views for a single class in a RSDS/UML specification.

Translating the Actuator Classes

Each actuator class is translated into an SMV module and all of its possible objects are defined in the main module as unique variables of type module. The class multiplicity describes the maximum number of objects that can be created for that class. These modules contain parameters of type module for defining the visibility amongst the classes determined by the event flow. Event flow is from sensors to controllers to actuators, as denoted by the hierarchy of receiving events in the original RSDS method. The controller is visible to all classes. Actuator modules also have an integer as a parameter, *id*, that uniquely identifies an object. If only one object is ever created, a unique identification integer is not required in SMV. This id is required in order to differentiate between the objects within the class module, especially when the objects react differently to the same transition. For example, the first instance has $id = 1$, the second instance has $id = 2$ and so on. Also, it's a way for other classes to refer to the objects of classes and for the controller to determine which events occur with which object. Figure 7.4 illustrates the objects that can be created for class *classA*. The following translation rule describes how a class is defined as an SMV module with parameters.

Rule OO-1: *For each actuator class with multiplicity p:*
```
MODULE classA(id, C)
```
where id is a parameter that uniquely identifies the instance i.e. 1,2,...,p, C is the controller module.

Rule OO-2: *For each class with multiplicity p of instances:*
```
MODULE main
 VAR
   object1 :  classA(1,C);
   ...
   objectp :  classA(p,C);
```
where object1 refers to the first instance created for class A and similarly for all p instances. The first parameter is an integer used to identify the instance. C gives the objects read-only access to the controller module.

SMV does not have the language constructs for modelling the creation and deletion of objects and thus it has to imitate it. All possible objects are predefined in the main module as described by translation rule OO-2. The boolean variable *alive* is introduced in each class module with the aim of recording the existence of objects of the class. The controller is responsible for broadcasting events to the actuator class objects that trigger transitions that result in the action of recording the existence of the objects (object creation or deletion). This should occur in the same step

(coarse-grain view), and therefore the controller and actuator transitions are synchronised in the SMV. An object is created by setting the variable *alive* to 1 (or true) as a result of such a controller event that occurs with the identification integer of the object (i.e. a controller transition). An object is destroyed when a controller event occurs for that object and the *alive* variable is set to 0 (or false). Usually the name of the event is affixed with the phrase *new* for creating an object or with the phrase *kill* for destroying an object. If all the objects of a class have been created and the controller broadcasts an event *new* to any of the existing objects, it will simply be ignored (**1:alive** means: otherwise the alive value stays the same). No extra objects will be created, if the number of existing objects is equal to the class multiplicity.

The translation rules for defining and changing the values of the *alive* variable are as follows.

Rule OO-3: *For each class:*
```
 VAR
   alive :  boolean;
```
is defined in the class module and indicates whether the instance of the class exists.

Rule OO-4: *The alive variable is initialised as follows:*
```
ASSIGN
 init(alive):= init_alive;
```
where init_alive is the initial value of the alive variable (usually 0 or false).

Rule OO-5: *An object is created by setting alive to 1 (true):*
```
ASSIGN
 next(alive):=
   case
     C.CT1 & id = C.idA : 1;
     ...
     1:alive;
   esac;
```
where id = C.idA refers to the specific instance that is created.

An actuator class usually consists of a single attribute that represents its current state, whose type is an enumerated set or integer. If the attribute type is an integer, then it must be finite and within a reasonable sized range. Otherwise, some kind of data abstraction technique should be applied. The reason for this is that the attribute is mapped to the current state in the corresponding statemachine diagram and its type represents all the other possible states in the statemachine. If this is too large the state space of the SMV model increases and can make model checking impossible. These attributes are translated into variables in the SMV modules.

Rule OO-6: *For each attribute st of enumerated type:*
```
 VAR
   st :  {st1, st2 ..., stn};
```
where {st1, st2 ..., stn} are all the possible states in the statemachine for this class.

Rule OO-7: *For each attribute att of integer type:*
```
VAR
   att :  1..k;
```
where k is a suitable (small) bound. We cannot quantify "small" as it depends on the overall size of the specification. The overall size of the model should have a maximum of 10^{20} *states.*

The initial values of the attributes, denoted by the initial states of the statemachines, are defined when the object of a class is created and not using the **init(st)** construct as expected in SMV. It will always be the case that the SMV step which evaluates **init(st)** will occur before the objects are created. Therefore, the transition that creates the object will also define the attribute's initial value.

Rule OO-8: *The initial value of attribute st is defined:*
```
ASSIGN
 next(st):=
   case
     C.CT1 & id = C.idA : init_st;
     ...
   esac;
```
where CT1 is the transition that creates the object with id=C.idA and init_st is the initial value of st.

The class methods or operations are mapped to events in the class' statemachine. In a object oriented language such as Java [Jav], all the transitions triggered by an event are collected and used to describe the operation for that event. In SMV, each transition is modelled by an expression that is evaluated to a boolean value [2] and when the evaluation is true, it indicates that a transition has happened. By modelling transitions in this way, they correspond closely to the definition of the semantics for RSDS/UML statemachines.

Therefore, transitions are described in SMV under the DEFINE clause in each module. The actuator transitions are defined in terms of the controller transition to ensure that its state or attribute changes occur in a single step. The guards of actuator transitions (if any) are defined as part of the controller transitions. This is because guards of actuator transitions refer to the states of other actuator objects that must be visible from the actuator object, and since all of the actuator objects are visible from the controller, then extra parameters do not need to be passed between the actuator objects. Each actuator transition contains the additional condition, which is part of the guard, that confirms that the variable *alive* is true, as only existing objects can react to sensor events, so if an object is not alive, it only "reacts" to controller events that create it. The additional *alive* variable does increase the state space, but we need a way of modelling the dynamic creation and deletion of objects. The effect of transitions is described in SMV using a case statement under the ASSIGN clause.

[2]The variables described under the DEFINE clause do not increase the state space as they are not considered as additional boolean variables. They are like macros.

Rule OO-9: *For each transition AT1 of an actuator class:*
```
DEFINE
 AT1 :  C.CT1 & st = s1 & id = C.idA & alive;
```
where C.CT1 is the controller transition and id refers to a particular instance (C.idA) that the event is applied to.

Rule OO-10: *The attribute values are changed by the transitions:*
```
ASSIGN
 next(st):=
   case
     AT1 :  st2;
     AT2 :  st3;
     ...
   esac;
```
where AT1... are the transitions for the class.

Translating the Controller Class

A separate SMV module is defined for the controller where its state, events and transitions are defined. It does not require an *alive* variable as it always exists. The module definition is parameterised to obtain visibility to the objects of classes that the controller is associated to as these associations are defined in the controller module and the existence of the objects must be confirmed.

Rule OO-11: *The controller is translated into a SMV module:*
```
MODULE Controller(L1,...,Lj)
```
where L1,...,Lj are modules of classes that are associated with the controller.

Rule OO-12: *In the main module the controller is represented as:*
```
VAR
 C : Controller(L1,...,Lj);
```

The controller class contains an extra variable that defines the sensor events as well as the events that create and delete class objects and add or remove associations between them. By not specifying the initial and next state, SMV chooses nondeterministically which event will occur. If a specific order in which the events occur is enforced by the physical properties of the system, then the event behaviour can be modelled as a statemachine that corresponds to a class representing the environment that is related to all the classes in the diagram. The **init(event)** and **next(event)** in SMV models the specific behaviour of events. In RSDS/UML we assume that the events occur nondeterministically.

Rule OO-13: *The current active event is translated as:*
```
VAR
  event :  {e1, e2 ..., em, none};
```
where {e1, e2 ..., em} are all the possible events that can occur. When event equals to none, no event occurs.

Variables are defined in the controller module for each class for identifying the objects of that class and used to determine which events are applied to which objects. If a class has only one object, then a variable id for it does not need to be defined. We assume that all the events apply to it and if there is no transitions defined for that event, it will be ignored.

Rule OO-14: *For indicating the current object that the event applies to:*
```
VAR
  idA : 1..3;
  idB : 1..4;
```
where idA and idB refer to the current object that the event is applied to.

The states are defined similarly to those of the actuator classes, except for the initial state that is defined using the **init(st)** construct since the controller already exists.

Rule OO-15: *The controller state is translated as:*
```
VAR
  cst :  {cst1, cst2 ..., cstn};
```
where {cst1, cst2 ..., cstn} are all the possible states in the controller statemachine.

Rule OO-16: *The controller state is initialised as follows:*
```
ASSIGN
 init(cst):= init_cst
```
where init_cst refers to the initial state in the controller statemachine.

The controller transitions are defined in terms of an event, the source state of the transition, the id of some object that the event is applied to and an actuator guard. The event is either a sensor event or an event that creates or deletes objects of classes or associations between the objects. The object id must be given to ensure that an event is applied to a particular object. The actuator guard (if any) refers to states of other actuator objects and are defined as part of controller transitions because the controller module has access to all actuator objects and actuator transitions are defined in terms of controller transitions. The sensor guard is implicitly expressed by the controller state.

Rule OO-17: *For each controller transition CT1:*
```
DEFINE
 CT1 :  event = e1 & st = s1 & idA = 1 & G;
```
where id refers to a particular object and G is the actuator guard, e.g. AA1.state = a1 or AA1.alive.

Rule OO-18: *The attribute values are changed by the transitions:*
```
ASSIGN
 next(cst):=
   case
    CT1 :  st2;
    CT2 :  st3;
    ...
    1:cst
   esac;
```
where CT1 are the transitions for the controller class.

Translating Associations

Associations between the controller and actuator objects are defined in the controller module. A variable of type array represents the relationship between the controller and a class with multiplicity greater than one. If the multiplicity of a class associated with the controller is equal to one, then a boolean variable is sufficient. Since there is ever only one instance of the controller, an array is not required in the actuator modules. If an actuator instance needs to know whether it is associated to a controller, it can refer to the controller array as the controller is visible from the actuator class. The role given on the association end is used as the name of the array. The array indices range from 1..m where m is the class multiplicity of the class associated with the controller. This range corresponds to the id values of objects of the class associated with the controller. Each position in the array holds a boolean value denoting the existence of an association, for example if the controller is associated with an object of a class with id = 3 then 1 is stored in the third position of the array (array index = 3). Figure 7.5 shows an example of a controller associated to the class Actuator and the following translation rules demonstrate how these are represented in the SMV controller module.

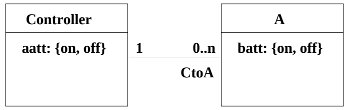


Figure 7.5: The Controller class associated to the Actuator class.

Rule OO-19: *For each association between a controller and an actuator class A whose multiplicity m is greater than one:*
```
VAR
 CtoA : array 1..m of boolean;
```
where CtoA is the role of the association.

Rule OO-20: *For each association between the controller and a class A whose multiplicity equals to one:*
```
VAR
 CtoA : boolean;
```
where CtoA is the role.

Usually, the initial values of the array are set to 0, that is the controller is not associated with any class instance as the actuator instances do not exist yet. These can be different depending on the system requirements.

Rule OO-21: *For each link with role CtoB:*
```
ASSIGN
 init(CtoB[1]) := init_1;
 init(CtoB[2]) := init_2;
  ...
 init(CtoB[m]) := init_m;
```

where init_1 is the initial value for the link (i.e. whether it is linked or not) between the controller and object 1 of class B. The total number of object for class B is m.

There are certain rules that are implied by the cardinalities on the association ends between two classes. These must be adhered to when creating or destroying objects and relating them. We represent the cardinality on the associations in an abbreviated form: A $_x \rightarrow _y$ B, where x is the cardinality found on the association end closest to class A and y is the cardinality found on the association end closest to class B. The rules are summarised as follows:

Translation rules	If A is deleted	If B is deleted	If A is created	If B is created
Rule OO-22: A $_1 \rightarrow _1$ B	Delete B & remove link	Delete A & remove link	Create B if no existing B & add link	Create A if no existing A & add link
Rule OO-23: A $_1 \rightarrow _{0..n\ or\ *}$ B	Delete B & remove link with object B	Remove link with object B	-	Create A if no existing A & add link
Rule OO-24: A $_1 \rightarrow _{1..n}$ B	Delete Bs & remove links	Delete A if this is the last B & remove link	Create B if no existing B & add link	Create A if no existing A & add link

Translation rules	If link is removed
Rule OO-25: A $_1 \rightarrow _1$ B	Delete both A and B objects.
Rule OO-26: A $_1 \rightarrow _{0..n\ or\ *}$ B	Delete B object.
Rule OO-27: A $_1 \rightarrow _{1..n}$ B	The B object is removed. If the A object is not linked to another B object, then remove A.

These rules are enforced by the controller transitions in SMV for creating or destroying an actuator object. For example, if an object of class B is removed by the controller transition CTx with event *deleteB* when the cardinality at both ends of the association is 1, then the object A is removed by the same transition as well as the link associating them. Therefore, the generated events *deleteA* and *removeAB* are not explicitly represented in SMV as we want to ensure that the effect of *deleteB* occurs in a single SMV step to correspond to a single coarse-grain step.

Associations are dynamic as they can be added or removed during run-time by the user. Events are defined in the controller module for adding and removing associations between the controller and some actuator class objects. The name of these events are written as **addCA** for adding an association, where C is the name of the controller and A is the name of the actuator class associated to the controller. The names of events for removing associations have a similar form. Associations between an object of a class A are added when the event **addCA** happens and the object to be associated to the controller is alive. If this condition is met, then the position of the array where the index is equivalent to object A's id is set to true (or 1). Conversely, if event **removeCA** happens then the position of the array where the index is equivalent to object A's id is set to false (or 0). These events and conditions are defined as part of controller transitions.

Rule OO-28: *To link or unlink the controller with an object B, controller transitions are defined with events prefixed with add and remove:*

```
DEFINE
 CT5 :  event = addCB & st = s4 & idB = 1 & G1;
 CT6 :  event = removeCB & st = s5 & idB = 1 & G2;
```

where idB refers to a particular object of B and, G1 and G2 refer to the guard.

Then, the array is updated as follows:

```
ASSIGN
 next(CtoB[1]) :=
   case
    CT5 & B1.alive :  1;
    CT6 & idB = 1 & B1.alive :  0;
    ...
    1 :  CtoB[1];
   esac;
```

where B1.alive refers to object 1 of class B alive variable to confirm its existence. Similarly for all positions in the array.

Translating Inheritance

Inheritance is a special kind of relationship ("is a") between classes as illustrated in (a) of Figure 7.6, where classes inherits the structure and behaviour of a superclass. There are two ways that we can model this relationship in SMV. One way involves copying all of the attributes and transitions of the superclass into the subclass, which enables the subclass to have direct access to the attributes and operations of the superclass. Translation rule OO-30 describes how to model inheritance in the this way. An alternative representation involves modelling inheritance as an association where the subclass has a cardinality of 0..1 as illustrated in (b) of Figure 7.6. Translation rule OO-29 describes how the SMV code is generated for (b).

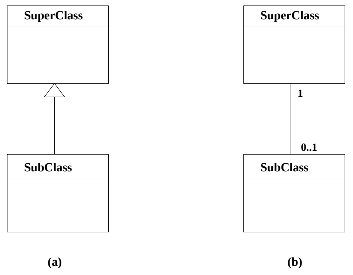


Figure 7.6: Inheritance illustrated in (a) can be expressed by associations as in (b).

Rule OO-29: *For the first modelling approach (see (b) in Figure 7.6), each subclass BC1,...,BCj of superclass AC:*

```
MODULE Associations(AC, BC1,...,BCj)
 VAR
   ACtoBC1:  array 1..k of boolean;
   ACtoBC2:  array 1..m of boolean;
   ...
```

Similarly for all subclasses of AC. The dynamic adding and removing of the objects are described using the init and next clauses as in translation rules OO-21 and OO-28.

Rule OO-30: *For the second modelling approach (see (a) in Figure 7.6), each subclass BC1,...,BCj of superclass AC:*

```
MODULE BC1
 VAR
   Attributes of AC
   Attributes of BC1
 ASSIGN
   Operations of AC
   Operations of BC1
```

Similarly for all subclasses of AC.

The advantage of defining inheritance as associations with the specific cardinality is that the SMV model is succinct. Nevertheless, it is unclear when explicit inheritance was intended. Therefore, we recommend the alternative approach of copying all the attributes and transitions of the superclass as it resembles the object oriented implementation of inheritance.

Translating the Invariants

The invariants for RSDS/UML specifications must hold for all instances of the classes. In SMV the properties to be verified are expressed in CTL and are defined under the SPEC clause. CTL does not contain any universal or existential quantifiers to reason over all instances of classes. Instead, invariants must be explicitly defined for each class instance. The class instances are defined as variables in the main module, and these are used in the expression of invariants.

Rule OO-31: *For each static invariant that defines constraints on attributes of class C with p instances and class A with q instances:*

```
SPEC
 AG(C1.catt = x -> A1.aatt = y)
SPEC
 AG(...)
SPEC
 AG(Cp.catt = x -> Aq.aatt = y))
```

where C1,...,Cp refer to the instances of C, A1,...,Aq refer to the instances of A, and x and y refer to some enumerated value of attributes catt of class C and A respectively.

Rule OO-32: *For each operational invariant that defines constraints on attributes of class C with p instances:*

```
SPEC
```

```
 AG(C1.catt = x & event = e1 -> AX(C1.catt = y))
SPEC
 AG(...)
SPEC
 AG(Cp.catt = x & event = e1 -> AX(Cp.catt = y))
```

where C1,...,Cp refer to the p instances of C and x and y refer to some enumerated value of attribute catt.

Rule OO-33: *For each temporal invariant:*

```
SPEC
 AG(P-> M(Q))
```

where P and Q define constraints on attributes of particular instances of classes and M is some temporal operator.

Invariants that describe constraints on the associations can also be verified. These type of invariants were not expressed in the original RSDS method as it was only capable of modelling systems with a static structure. Again, these invariants must hold for all instances of the classes mentioned. Translation rule 00-34 demonstrates how constraints on associations are expressed as CTL properties in SMV without the use of quantifiers. However, association constraints are not limited to this form but we want to highlight in the translation rule how the associations are expressed in the CTL properties and that CTL properties must be defined for all instances.

Rule OO-34: *For each invariant that defines constraints on associations between the controller C and class D with p instances:*

```
SPEC
 AG(C.CtoD[1] = 1 -> (C.attc = x -> D1.attd = y))
SPEC
 AG(...)
SPEC
 AG(C.CtoD[p] = 1 -> (C.attc = x -> Dp.attd = y))
```

where D1,...,Dp refer to the p objects of D and x and y refer to some enumerated value of attributes for the controller and class D respectively.

Usually, invariants apply only to “alive” instances. Therefore, in their definition a condition must be included that checks whether the instance is alive so that SMV will only check the states where alive is true. If property checks were to be performed on non-alive instances, there will not be a state explosion problem, unless there is a state explosion problem with the alive instances. This is because there are a finite number of instances that are defined for each class (maximum number of instances is specified by the class multiplicity) and at some point either all the instances are alive, and at another (e.g. initialy) all the instances are not alive.

7.3.3 The translation schemas for the actuator and controller

We have presented the translation rules for both the controller and actuator modules. The translation schema for the SMV code generated for a system is as follows.

```
MODULE main
VAR
```

```
 C : Controller(A1,A2,..,Ap);
 A1 :  Actuator(1,C);
 A2 :  Actuator(2,C);
 ...
 Ap :  Actuator(p,C);

SPEC
 AG(P)
 –where P is the property to be verified.
```

```
MODULE Controller(AA1,AA2,..,AAp)
VAR
 state :  {cst1, ...,cstn};
 event :  {e1,e2,...,em,none};
 idA : 1..p;
 CtoA : array 1..p of boolean;

DEFINE
 CT1 :  event = newA & state = cst1 & idA = 1 & AA1.alive;
 CT2 :  event = killA & state = cst2 & idA = 1 & AA1.a_state = a1;
 ...
ASSIGN
 init(state) := cst1;
 next(state) :=
  case
   CT1 :  cst2;
   CT2 :  cst4;
   ...
   1:  state;
  esac;

 init(CtoA[1]) := 0;
 next(CtoA[1]) :=
  case
   CT1 :  1;
   CT2 :  0;
   ...
   1:  CtoA[1];
  esac;
  ...
 init(CtoA[p]) := 0;
 next(CtoA[p]) :=
  case
   CT1 :  0;
   CT2 :  0;
   ...
   1:  CtoA[p];
  esac;
```

```
MODULE Actuator(id, C)
–id refers to the instance unique identifier, C to the controller
VAR
```

```
 a_state :  {a1,...,ar};
 alive :  boolean;

DEFINE
 AT1 :  C.CT1 & id = 1;
 AT2 :  C.CT2 & id = 1;
 AT3 :  C.CT3 & alive & id = C.idA;
–where A.an_state = aa2 is the guard of the transition.
 ...

ASSIGN
 next(a_state) :=
  case
   AT1 :  init_a_state;
   AT3 :  a2;
   ...
   1:  a_state;
  esac;

 init(alive) := 0
 next(alive) :=
  case
   AT1 :  1;
   AT2 :  0;
   ...
   1:  alive;
  esac;
```

7.3.4 Translating the gas burner system

Let us demonstrate how the gas burner system, whose class diagram is illustrated in Figure 7.7, is model checked with SMV. The complete SMV code generated using the translation rules is given in Appendix D.1.

We assume that the controller is derived from the amalgamation of the sensor components and that it always exists. It is responsible for creating and deleting the objects for the actuators and for defining the associations between the controller and the actuator objects. There are two actuator classes in this system: one for the valves and one for the igniter.

The structure of the gas burner system is static. Therefore we assume that the objects for the actuator components are created initially by the controller and destroyed once the system is in a safe state, that is when the flame is absent and the switch is off. Figure 7.8 describes the statemachine for the controller. The controller generates the events {*newIG, newV, shutdown*} that create and destroy the actuator objects. The links that relate the controller with the actuator objects are defined in the controller. Since there is only one igniter object, a boolean variable is used instead of an array to record the existence of its association with the controller and no *id* variable identifying igniter objects. Therefore, there is only one set of ids defined in the controller identifying the instances of the class Valve. If an event does not trigger any transitions, the values of variables in the system remain the same (i.e. skip).

```
MODULE Controller
VAR
```

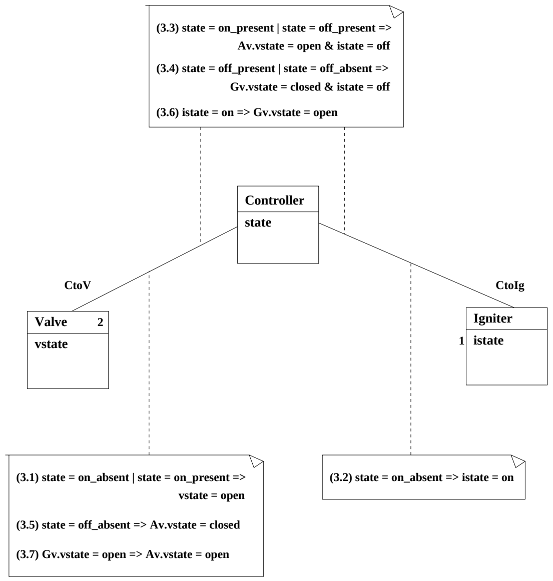

(3.3) state = on_present | state = off_present => Av.vstate = open & istate = off
(3.4) state = off_present | state = off_absent => Gv.vstate = closed & istate = off
(3.6) istate = on => Gv.vstate = open
Controller
state
CtoV
CtoIg
Valve 2
vstate
Igniter
1 istate
(3.1) state = on_absent | state = on_present => vstate = open
(3.5) state = off_absent => Av.vstate = closed
(3.7) Gv.vstate = open => Av.vstate = open
(3.2) state = on_absent => istate = on

Figure 7.7: The class diagram for the gas burner system.

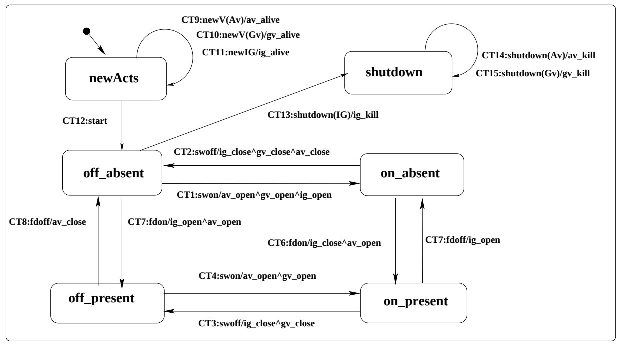


Figure 7.8: The statemachine for the controller class of the gas burner system.

```
 state :  {Off_Absent, On_Absent, Off_Present, On_Present, newActs, Shutdown};
 event :  {swon, swoff, fdon, fdoff, shutdown, newV, newIG, start, none};
 idV : 1..2;
 CtoV : array 1..2 of boolean;
 CtoIG : boolean;

DEFINE
 CT1 := event = swon & state = Off_Absent & Va.alive & Vg.alive & iG.alive;
 CT2 := event = swoff & state = On_Absent & Va.alive & Vg.alive & iG.alive;
 ...
 CT10 := event = newV & idV = 2 & state = newActs & Va.alive & Vg.alive & iG.alive;
 CT11 := event = newIG & state = newActs;
 ...
ASSIGN
 init(state) := newActs;
 next(state) :=
  case
   CT1 :  On_Absent;
   CT2 :  Off_Absent;
   ...
   1:  state;
  esac;

 init(CtoIG) := 0;
 next(CtoIG) :=
  case
   CT11 :  1;
   CT13 :  0;
   1:  CtoIG;
  esac;

 init(CtoV[1]) := 0;
 next(CtoV[1]) :=
  case
   CT9 :  1;
   CT14 :  0;
   1:  CtoV[1];
  esac;

 init(CtoV[2]) := 0;
 next(CtoV[2]) :=
  case
   CT10 :  1;
   CT15 :  0;
   1:  CtoV[2];
  esac;
```

The statemachines for the actuator component objects are illustrated in Figure 7.9. These statemachines are translated into SMV. The actuator transitions are defined in terms of controller transitions to ensure that their states change in a single SMV step. These class objects cannot react to sensor events if they do not exist. The valve objects require read-only access to each other as they both need to be alive in order to respond to the sensor events. This additional condition is part of the guard of the actuator transition.

The objects are defined in the main modules and the module parameters are given specific values, for example the controller object is passed as parameter to the actuator objects. The first

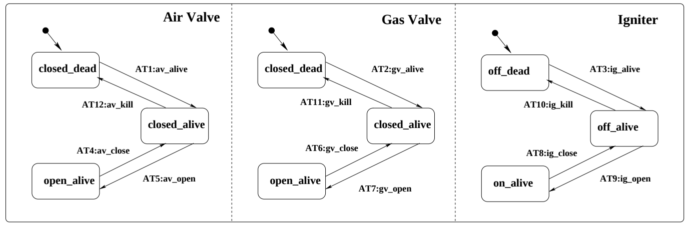


Figure 7.9: The statemachines for each actuator class of the gas burner system.

two invariants are the same as in the original coarse-grain implementation of the gas burner, that is, constraints on class attributes, with the additional conditions confirming the existence of the class instances. The last two invariants ensure that if the instances for the actuator classes exist then they must be associated to the controller. These invariants imposed on the system structure could not be formulated in the original RSDS specifications.

```
MODULE main
VAR
 C : Controller(Av,Gv,Ig);
 Av :  Valve(C,1);
 Gv :  Valve(C,2);
 Ig :  Igniter(C);

SPEC
  AG(Gv.va = open & Gv.alive -> Av.va = open & Av.alive)
SPEC
 AG(Ig.ig=on & Ig.alive & Gv.alive -> Gv.va = open)
SPEC
 AG(Ig.alive -> C.CtoIG)
SPEC
 AG(Av.alive -> C.CtoV[1])
SPEC
 AG(Gv.alive -> C.CtoV[2])
```

7.4 Comparison with RSDS translations

We have defined translations for both semantic views of RSDS specifications, namely, for the coarse-grain and the fine-grain. In this chapter, we presented a translation for the coarse-grain view of RSDS/UML specifications. We compare all of these translations based on the following criteria (summary in Table 7.3):

1. **Readability:** how easy it is for a user to understand the SMV model produced from the translation ?

2. **Expressiveness:** how well have all the elements in the original model (RSDS or RSDS/UML model) been expressed in the SMV model ?

3. **Coverage:** have all the elements in the original model been translated ?

4. **Efficiency:** how much of the resources is required for model checking the generated SMV model ?

The coarse-grain and fine-grain translations both preserve the modular structure of RSDS specifications. Therefore, a module in SMV is defined for each component in RSDS, i.e. one for each controller and one for each actuator (sensors are combined to form the controller). In the fine-grain translation the queue data structure is translated into a separate SMV module so that it is clear to the user that it is additional to the basic components. In the RSDS/UML translation, the structure of the class diagram in preserved in the SMV model, i.e. each class is translated into a separate SMV module. Also, the variables that represent the associations between the classes are placed in the SMV module for the controller. Therefore, all three translations generate models that can be easily read by the user.

There are problems with the expressiveness of the generated SMV model for all of the translations. For the coarse-grain translation, the actuator events are not explicitly represented in the SMV model. Instead they are described in terms of controller transitions that generate them. This is because we want to model one coarse-grain step as one SMV step. For the fine-grain translation, the queue elements are not dynamically generated. Instead, the snapshot of the queue for each step must be determined before translating i.e. we use this information in order to translate. The RSDS/UML translation cannot translate the dynamic instantiation of classes. Furthermore, the number of instances that can ever be created must be known and also be finite.

All the elements in the coarse-grain and fine-grain RSDS specification are translated into SMV. However, the RSDS/UML translation is not able to directly translate the dynamic creation and deletion of instances. This is because in SMV there are no constructs for defining variables at run-time.

The gas burner system was specified in both RSDS and RSDS/UML and translated into SMV model. The following table describes the resource usage for model checking (with the NuSMV tool) these models.

	RSDS/UML	RSDS coarse-grain	RSDS fine-grain
User time:	0.09 sec	0.06 sec	0.08 sec
System time:	0.03 sec	0.04 sec	0.04 sec
Virtual data size (bytes allocated):	6483K	6357K	6536K
BDD nodes allocated:	3931	733	8021
BDD cluster size:	345	59	570

Once more, the user and system time is too small to be able to use it to compare the SMV models efficiency. Therefore, we consider the BDD nodes as a reliable metric, where the lower the number of nodes the better it will be dealt with by the model checker. The coarse-grain RSDS/UML model is allocated more BDD nodes compared to the RSDS coarse-grain SMV model and less than the RSDS fine-grain SMV model. Following this pattern, we assume that the fine-grain SMV model for RSDS/UML specifications will have more BDD nodes than the fine-grain model of RSDS specifications. Therefore, the SMV models for RSDS/UML specifications are more expensive in space than RSDS specifications and this should be considered when deciding which modelling approach to adopt.

Table 7.3: Comparison of RSDS and RSDS/UML translations

Criteria	**RSDS translations Coarse-grain (CG)**	**Fine-grain (FG)**	**RSDS/UML translation Coarse-grain**
Readability	Yes - modularity is preserved	Yes - modularity is preserved & so is the event queue	Yes - class diagram structure preserved
Expressiveness	Actuator events not explicitly represented	Snapshots of the queue must be worked out before translating	Dynamic instantiation mimicked - also needs to be finite
Coverage	All elements translated	All elements translated	Cannot create instances at run-time
Efficiency	Fastest	Slowest	Greater than CG and less than FG

7.5 Proof of Correctness of the RSDS/UML Translation

The proof of correctness for the coarse-grain RSDS/UML translation is described similarly to those given for the RSDS coarse-grain and fine-grain translations i.e. we must show that the translation is sound and complete.

For soundness, we need to show that the axioms describing the RSDS/UML model elements hold from Th_{OOSys} in $\Lambda_{M(OOSys)}$ under the interpretation, that is we need to show ξ exists in Figure 7.10. Since Th_{OOSys} is the union of the class diagram theory and each of the statemachine theories, we need to show that their axioms hold in $\Lambda_{M(OOSys)}$.

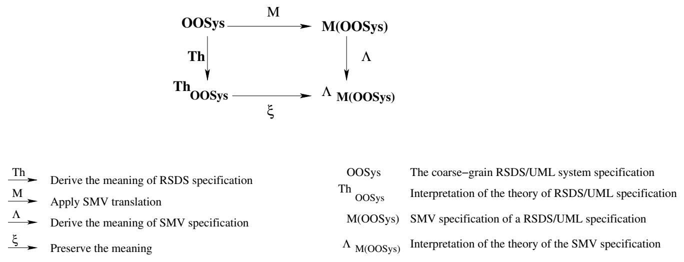

OOSys
M
M(OOSys)
Th
Λ
Th OOSys
ξ
Λ M(OOSys)
Th Derive the meaning of RSDS specification
M Apply SMV translation
Λ Derive the meaning of SMV specification
ξ Preserve the meaning
OOSys The coarse−grain RSDS/UML system specification
Th OOSys Interpretation of the theory of RSDS/UML specification
M(OOSys) SMV specification of a RSDS/UML specification
Λ M(OOSys) Interpretation of the theory of the SMV specification

Figure 7.10: A sketch of the proof of correctness.

Lemma 5. The axioms of Th_{OOSys} hold in $\Lambda_{M(OOSys)}$ under the interpretation ξ of an RSDS system theory Th_{OOSys} as an SMV interpretation of that theory in $\Lambda_{M(OOSys)}$.

Proof. The proof of the lemma is organised as follows:

- We first define the interpretation of attributes of the RSDS/UML theory Th_{OOSys} into the theory $\Lambda_{M(OOSys)}$.

- Then, we show that the axioms of the class theory Δ_{CD} hold from Th_{OOSys} in $\Lambda_{M(OOSys)}$ under the interpretation.
- Finally, we show that the component and system axioms of the statemachine theories Th_{Sys} hold from Th_{OOSys} in $\Lambda_{M(FOOSys)}$ under the interpretation.

Interpretation of attributes

The interpretation of the coarse-grain RSDS/UML system theory Th_{OOSys} into the theory $\Lambda_{M(OOSys)}$, that represents the meaning of the translation into SMV, is defined by:

1. Each instance $x \in @C$ for a class C is interpreted as a module *Cx: C(x,p1,...,pn)* where x is the unique identifier of the instance and is of type integer, and *p1,...,pn* are all the other instances whose states are accessible. According to the syntactic restrictions, there should only ever be one instance defined for the controller that has access to the states of all the actuators in the system, while all the actuators should have access to the controller instance.
2. The attribute $\overline{C} : \mathbb{F}(@C)$ for the set of all existing instances of each actuator class C is interpreted as the collection of SMV modules C_p where $\mathtt{C_p.alive} = \mathtt{1}$ (true). We assume that the controller class is always alive and therefore no alive variable is defined for it.
3. The attribute $\overline{r} : \mathbb{F}(@Cont \times A)$ for the set of all existing instances of each association r, with end points at the controller class *Cont* and actuator class A, is interpreted as an array of the form *r: array 1..x of boolean* where x refers to the multiplicity of A. The array is defined in *Cont*.
4. The notations $y \in cs(x)$ and $x \in as(y)$ that abbreviate to $(x, y) \in r$, where cs and as are the pseudo-attributes for each binary relation r between classes C and D are interpreted as array position *C.r[y]*.
5. The attribute *att* for the controller class *Cont* is interpreted as *Cont.att*, in its instance module *Cont*. This also corresponds to the current state in the controller statemachine.
6. The attribute *att* for an actuator class A is interpreted as *Ax.att*, in each instance module *Ax*. This also corresponds to the current state in the actuator statemachine.
7. The attribute m for the multiplicity of a class C is interpreted as the maximum number given to an instance as an id in the main module in SMV, i.e. *Cm: C(m,p1,...,pk);* where *p1,...,pk* are modules of other instances.
8. The sensor events $\alpha : Events$ are interpreted as $Cont.event = \alpha_{tok}$ which include the events for creating and deleting instances and associations.
9. Each operation *op* of a class C is mapped to a sensor event in the corresponding statemachine and the body of the operation is mapped to a collection of transitions for that sensor event. In SMV, each transition is defined individually, and thus we adopt this approach as well. Therefore, transitions *tr* are interpreted in two ways depending on the component:

(a) A controller transition is interpreted as a predicate $\mathbf{C.CT}_j\mathbf{=1}$ where $\mathbf{CT}_j$ is the corresponding DEFINE variable of the controller module.

(b) An actuator transition is interpreted as the disjunction $\mathbf{C.CT}_{k1}\mathbf{=1} \vee \mathbf{...} \vee \mathbf{C.CT}_{kp}\mathbf{=1}$ where $\mathbf{CT}_{kj}$ represent all the controller transitions which invoke tr in the RSDS model. Similarly for actuator events.

10. Actions for creating and deleting instances (these apply only to instances of actuators):

(a) $x.new_A$ is interpreted by *Cont.event=newA & Cont.idA=x* and

(b) $x.kill_A$ is interpreted by *Cont.event=killA & Cont.idA=x.*

where *idA* is the unique identifier of class A instances. If there is only one instance of a class then *Cont.idA=x* is not given. These actions do not apply to the controller instance as it always exists.

11. For each association r, actions for linking and unlinking tuples of instances in relationships (associations only exist between the controller and actuators):

(a) $link_r(Cont, y)$ is interpreted as *Cont.event=add_r & idA=y* where y refers to the instance of class A that must be associated and

(b) $unlink_r(Cont, y)$ is interpreted as *Cont.event=remove_r & idA=y* where y refers to the instance of class A that must no longer be associated.

Axioms that hold under the interpretation

We show how the axioms for the class diagram, object diagrams and statemachines hold under the interpretation.

Class diagram axioms:

RU1 The axioms defining the initial state of class attributes hold:

1. in the controller module, the initial state is defined as *init(att) := init_c* where *init_c* is the initial state of the controller class.
2. in the actuator module, the initial state is defined when the instance is created. The transition that creates the object sets the value of the *att* via the case statement in the *next(att)* clause.

RU2 We strengthen the axioms

$$AG(\overline{r} \subseteq \overline{D_1} \times ... \times \overline{D_k})$$

for each association r, with end points at classes D_1 to D_k, with the axioms

$$AG(\overline{r} \subseteq \overline{Cont} \times \overline{A_1} \times ... \times \overline{A_k})$$

for each association r, with end points at the controller class $Cont$ and at A_1 to A_k, since we know that control systems only contain associations between the controller class and actuator classes. Therefore, these strengthened axioms hold under the interpretation because of *Cont.r[y] = 1* $\Rightarrow$ *A1y.alive =1*, where y is the instance id of class A1, and similarly for all actuator classes. The controller is always alive.

RU3 Axioms defining the allowed multiplicity of an association r between the controller and actuator classes hold because of the definition of the association in SMV. In the case where the multiplicity is one on both ends of the association, then the association is defined as *r: boolean;*. Otherwise, if the multiplicity is p..q or m where m is some constant, then the association is defined as an array *r : array 1...q of boolean;* or *r : array 1...m of boolean;*. If the multiplicity $*$, then an array is defined with a maximum size equal to the class multiplicity.

RU4 Axioms for:

1. creating instances hold because the effect is represented by setting the alive variable for instance x to true (*x.alive = 1*).
2. deleting instances hold because the effect is represented by setting the alive variable for instance x to false (*x.alive = 0*) indicating that it does not exist.

RU5 The axioms for defining the effect of actions for:

1. linking tuples of instances in associations hold because transitions that are triggered by *C.event = add_r* and are applied to particular instances *idD=j* have the effect of setting the array position *r[j]* to true between two instances using the SMV assignment statement *next(r).*
2. unlinking tuples of instances in associations hold because transitions that are triggered by *C.event = remove_r* and applied to particular instances *idD=j* have the effect of setting the array position *r[j]* to false between two instances using the SMV assignment statement *next(r).*

RU6 The axiom for ensuring that the multiplicity of a class is adhered to holds because of the definition of the instances in the main module whereby the last one for each class has the same id (number) as its class multiplicity.

RU7 Axioms $AG(@D \subseteq @C)$ and $AG(\overline{D} \subseteq \overline{C})$ if D inherits C hold because the SMV module describing the class D contains a copy of all the attributes and methods of the SMV module for C.

RU8 Locality axioms for every class hold because if no transition or operation occurs then the default case *1:att;* of the case statement defined in the assignment statement *next(att)* denotes that the *att* value does not change.

Axioms used to define the statemachine theory:

RU12 The state-transition behaviour of the statemachine Sm, for all $x : @C$ where C is a class, is described for each transition tr of Sm with source s, target t, guard G, and trigger event α with the axioms:

1. The axiom $AG(sm(x) = s \wedge G(x) \wedge \alpha(x) \;\Rightarrow\; tr(x))$ where sm refers to the state holds because the controller transitions, which only occur if all the terms evaluate to true, are defined as

   CT1: event = e & G & st = s1 & id =x

   and the actuator and sensor transitions have the following similar form:

   AT1: C.CT1 & G & at = a1 & id =x & alive

   where *CT1* and *AT1* are the controller and actuator transitions respectively, and G is the guard and *st* refers to the state (or class attribute) and *id* to a particular instance.
2. The second axiom $AG(tr(x) \;\Rightarrow\; \alpha(x))$ holds because transitions are defined in terms of sensor events and therefore, an event must occur in order for a transition to happen.
3. The third axiom $AG(sm(x) = s \wedge tr(x) \;\Rightarrow\; AX(sm(x) = t))$ holds because of the assignment *next(st)* used to define the state changes. A case statement is used to distinguish the different effects of different transitions on the state. The LHS of the axiom is represented by the LHS of the case statements and the RHS of the axiom by the RHS of the case statement:

```
next(st):= case
  CT1 : t;
  ...
  1:st;
esac;
```

   where *CT1* is a definition of a transition that contains the LHS of the axiom.

RU13 For each pair of distinct transitions of Sm, at most one transition of Sm can occur in a step holds because each actuator transition variable is defined in terms of the controller transitions:

AT1: C.CT1 & G & at = a1 & id =x & alive

and only one of these can be true in any step. Similarly for events.

RU14 That a transition can only occur if Sm is in its source state where $s = source_{Sm}(tr)$ holds under the interpretation because the source state is explicitly given as part of the condition in the definition of a transition. If the system is not in the source state, the transition will not occur.

RU15 The locality axioms for statemachines hold as for the locality axioms of the classes.

RU16 At the system level, the global axiom stating that generated events (that trigger actuator transitions) only occur as a reaction to a controller transition holds because the controller and actuator transitions are synchronised in order to ensure that their reaction occurs in a single step. The synchronisation is implemented by defining the actuator and sensor transitions in terms of the controller transitions. Therefore the event is responded to simultaneously by the controller and by the actuators.

□

In order to show the completeness of the RSDS/UML translation, we use the following lemma.

Lemma 6. For each trace h of the SMV model $M(OOSys)$ there is a corresponding trace $r(h)$ in the RSDS/UML model $OOSys$ which satisfies the same formulae with respect to the translation ξ.

Proof. If h consists of a sequence $h_1, ..., h_n$ of n Kripke nodes (where RSDS/UML states and events are represented by variables), then $r(h)$ is defined as a sequence of n states and events as follows.

The settings of the sensors at state i of $r(h)$ are given by the value of *C.state* in h_i. The settings of other components are given by their values in h_i. The sensor event which occurs, to move from node i to node $i + 1$ is given by *C.sensor_event* in h_i. The response actuator events and transitions are given by those disjunctions of conditions used in $M(OOSys)$ to define their occurrence, which are true (i.e. their value is 1) in h_i. The settings for mimicking the dynamic instantiation of classes and associations are given by values of *alive* for each class, *CtoA* (where A is another class that the controller C is associated to) and *idA* in h_i.

These two traces have the property that:

$h \vdash_{\Lambda_{M(OOSys)}} \xi(\varphi) \;\Leftrightarrow\; r(h) \vdash_{\Gamma_{OOSys}} \varphi$ □

Hence, if h is an SMV counter-example to an RSDS/UML property ψ, i.e.

$h \vdash_{\Lambda_{M(OOSys)}} \neg\xi(\psi)$

then $r(h)$ is an RSDS/UML counter-example to ψ:

$r(h) \vdash_{\Gamma_{OOSys}} \neg(\psi)$

Let us consider a simple RSDS/UML system with a controller class C with states

$$\{s11\_s21, s12\_s21, s11\_s22, s12\_s22, s11\_s23, s12\_s23, \}$$

that is associated to an actuator $A1$ (class with only a single instance) with states $\{a1, a2, a3\}$. If

$$\varphi = \mathit{C.state} = s12\_s21 \;\&\; \mathit{C.CT1} = 1\& \; \mathit{A1.alive} = 1 \;\Rightarrow\; AX(\mathit{A1.state} = a2)$$

then a possible counter-example h produced for this system could be:

h_1 *C.state* = *s11_s21*
C.sensor_event = α
A1.state = $\{a1, a2, a3\}$
A1.alive = *0*
C.CtoA1 = 0 (dont need to use array if there is only one instance)
A1.id = *C.idA1* = *1*
C.CT1 = *1* (where $CT1$ is defined in terms of this particular state and event)
A1.AT1 = *1* (where $AT1$ is defined in terms of $CT1$ and creates instance of $A1$)

h_2 *C.state* = *s12_s21*
C.sensor_event = β
A1.state = *a1*
A1.alive = *1*
C.CtoA1 = 1
A1.id = *C.idA1* = *1*
C.CT2 = *1*
A1.AT2 = *1*

h_3 *C.state* = *s12_s22*
C.sensor_event = γ
A1.state =*a3*
A1.alive = *1*
C.CtoA1 = 1
A1.id = *C.idA1* = *1*
C.CT3 = *1*
A1.AT3 = *1*

where in h_3 *A1.state* = *a3* instead of *A1.state* = *a2*. The corresponding counter-example $r(h)$ in the RSDS/UML model $OOSys$ is:

$r(h_1)$ *C.state* = *s11_s21*
sensor_event = α (for creating an instance of A1)
The controller instance always exists
No instance of A1 exists
$tr_C = \alpha/\rho_1$
$tr_A = \rho_1$

$r(h_2)$ *C.state* = *s12_s21*
sensor_event = β (applied to A1 instance)
Instance of A1 exists and is associated to the controller instance
A1.state = *a1*
$tr_C = \beta/\rho_2$
$tr_A = \rho_2$

rh_3 *C.state* = *s12_s22*
sensor_event = γ (applied to A1 instance)
A1.state =*a3*
Instance of A1 exists and is associated to the controller instance
$tr_C = \gamma/\rho_3$
$tr_A = \rho_3$

This example illustrates the close correspondance between h and $r(h)$.

The same reasoning as in 6 also shows the completeness of the translation.

Theorem 5. *The translation is complete if we can show that: if* $\Lambda_{M(OOSys)} \vdash \xi(\varphi)$ *then* $\Gamma_{OOSys} \vdash \varphi$.

Proof. We prove the contra-positive: if $not(\Gamma_{OOSys} \vdash \varphi)$ then $not(\Lambda_{M(OOSys)} \vdash \xi\varphi)$ For every trace s of an RSDS/UML system it is simple to construct a trace h of the corresponding SMV system for which $s = r(h)$. If

$$not(\Gamma_{OOSys} \vdash \varphi)$$

there is a counter-example trace

$$s \vdash_{\Gamma_{OOSys}} \neg\varphi$$

But then

$$h \vdash_{\Lambda_{M(OOSys)}} \neg(\xi(\varphi))$$

and

$not(\Lambda_{M(OOSys)} \vdash \xi\varphi)$ which proves completeness. □

7.6 Generalising RSDS/UML

The RSDS/UML method is currently being revamped to support the development of critical systems not limited by the static structure of control systems, (i.e. the DCFD structure of sensors, controllers and actuators) so that it can be applied to more general application domains, such as critical e-commerce systems. It aims to embrace more fully the object oriented development approach of UML in order to feel the impact of the benefits on the development process. In [LAC03], a railway system has been developed using the new RSDS/UML approach and its class diagram is illustrated in Figure 7.11. This representation is more concise than the one that maps each system component to a separate class. Instead, the classes represent either physical or conceptual parts of the system, for example in the railway system the class *Location* refers to the physical track segment that a train can occupy, while the class *Route* is a purely conceptual entity. The sensor inputs and actuator outputs are modelled as attributes whose name is prefixed with a ? for sensors (input) and a ! for actuators (output). The controller is not explicitly modelled as a class in the class diagram but nevertheless is considered as being related to all of the classes and is derived automatically by the RSDS/UML tool during code synthesis. It still receives events from the environment and is responsible for deriving the reaction of the system to those events. The constraints that are attached to the classes or associations are given in an abbreviated form of OCL [UMLa] that excludes the quantifiers but its interpretation implicitly provides them.

We have not presented the details of the translation of generalised RSDS/UML specifications into SMV for the purpose of verification as the semantics for the new RSDS/UML approach have not been consolidated. However, we believe that the translation rules presented in this chapter are applicable to the new RSDS/UML approach with few alterations. We are interested in pursuing this research direction as verifying general UML specifications, both static and dynamic aspects, could benefit a wider range of users. Furthermore, we envision including additional UML views of the system in the RSDS/UML specification that could assist in the interpretation of the verification results, that is, if a counter-example is produced, a sequence diagram can illustrate the steps leading to the violation of an invariant. Moreover, sequence diagrams can be used to delineate the creation and deletion of objects and can be used for generating the translation into SMV.

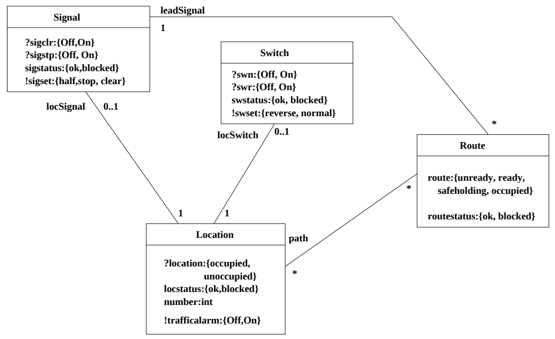


Figure 7.11: Class diagram for the railway system.

7.6.1 Additional translation rules

In this section, we give a taste of some additional translation rules that could be defined. One way of generalising the UML model is by allowing any class to be associated to any other. For example, in Figure 7.12 the class Person is associated with the class Jobs, meaning that each person has 0..2 jobs. This is a typical example in UML.

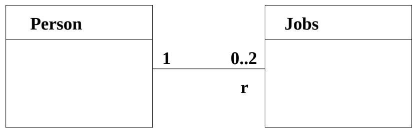


Figure 7.12: A typical example of UML class diagram.

In SMV, an association is represented by an array in the SMV module for the class whose name is given by the role. The indices of the array represent the object ids for the class that it is associated to. Therefore, each instance has an array describing which instances of the other class it is associated to, that is, the array for each instance represents the instance's view of the association.

Recall the example in Figure 7.12. If the following instances are created for these classes:

$p1,\ j1$
$p1,\ j2$
$p2,\ j3$
$p2,\ j4$

where the prefix p refers to the objects of class Person and the prefix j refers to the objects of class Jobs, then the SMV model contains the arrays with the following values:

$Person1.r\ [1,1,0,0]$
$Person2.r\ [0,0,1,1]$
$Job1.r\ [1,0]$
$Job2.r\ [1,0]$
$Job3.r\ [0,1]$
$Job4.r\ [0,1]$

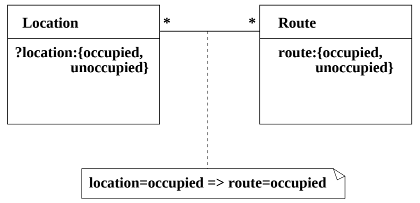
The translation rules **OO-22** to **OO-27** describe the rules implied by the cardinalities when objects are created and destroyed. Additional rules are summarised in the table for the rest of the cardinalities that were not considered.

Translation rules	If object A is deleted	If object B is deleted	If object A is created	If object B is created
Rule OO-32: A $_{0..n\ or\ *} \rightarrow _{0..n\ or\ *}$ B	Remove link with object B	Remove link with object A	-	-
Rule OO-33: A $_{1..n} \rightarrow _{1..n}$ B	Delete B if this is the last A & remove link	Delete A if this is the last B & remove link	Create B if no existing B & add link	Create A if no existing A & add link

Translation Rules	Association removed
Rule OO-34: A $_{0..n\ or\ *} \rightarrow _{0..n\ or\ *}$ B	-
Rule OO-35: A $_{1..n} \rightarrow _{1..n}$ B	If a last A object, then remove it. If a last B object, then remove it.

7.7 A Simple Railway System for Testing Efficiency

In this section, we consider a small part of the railway signalling system [LAC03] for two reasons: firstly we want to show how we represent in SMV dynamic associations between classes and secondly, we want to investigate how efficient it is to model check object oriented models and to justify our choice of some translation rules defined for RSDS/UML based on the results obtained. The complete SMV listings can be found in Appendix D.2 and D.3.



Location
?location:{occupied, unoccupied}
*
*
Route
route:{occupied, unoccupied}
location=occupied => route=occupied

Figure 7.13: Part of the railway system described in Figure 7.11.

We focus only on two entities in the railway signalling system as illustrated in Figure 7.13: Location and Route. The railway network consists of a set of locations, each which has an occupancy detector. It is divided into routes, which consists of a sequence of adjacent track locations extending from a location containing a signal. In [LAC03], Location and Route have a static association. We revise this association to make it dynamic for explanatory purposes, that is the associations between objects can be added during run-time. The association between objects is only removed when one of the objects is destroyed. We abstract most of the attributes from Figure 7.11 and simply represent a single attribute for each class as shown in Figure 7.13. This is because we are only interested in the attributes that are involved in the following invariant:

$$\mathit{location} = \mathit{occupied} \Rightarrow \mathit{route} = \mathit{occupied}$$

"If any location in a route is occupied, the route is occupied."

This invariant applies to all instances of Location and Route, that is to all railway networks. We want to verify this invariant in SMV.

We use the translation rules defined to produce the SMV model for this part of the railway system. Two separate modules are defined for the classes Location and Route. Moreover, a module is defined for the controller although it is not visualised in the class diagram. The events are defined in the controller for creating, deleting instances of Location and Route, and for defining the associations between their objects. The controller variables *locationid* and *routeid* identify which instances of Location and Route, respectively, the events are applied to. Instances of Location and Route can be associated when the event *link* happens and the Location instance is unoccupied. Once they are associated, then their attribute is set to *occupied* in order to maintain the invariant. Therefore, if the event link(L0,R1) happens, then the state of L0 must be unoccupied and all the states of route instances that are associated to L0 must be updated to occupied. The controller transition CT1 with event *locationoccupied* ensures that Route attribute is set to occupied. The associations between instances are only removed when any of the instances involved in the association are destroyed. We have identified two ways in which the associations are represented in an SMV model. In our investigation, we test which representation is more efficient. The first way is as defined by the translation rules for RSDS/UML specifications, where the arrays are defined in the controller module representing the association between the controller (with a single instance) and an actuator class. In this case, since neither of the classes associated is a controller and they both have more than a single instance, a 2-Dimensional (2D) array is used to represent the instances of both classes. For the railway system, the 2D array defined in the controller is *Location_Route* that has the meaning $\mathit{Location\_Route} \subseteq \overline{\mathit{Location}} \times \overline{\mathit{Route}}$.

The second way of representing associations consists of defining an array (1-Dimensional) in each module representing the class, where the indices of the array refer to the instances of the class that it is associated with. Therefore, each instance of the class will have its own array that indicates which instances of the other class it is associated to. For the railway system, the array *LocRoute* is defined for each instance of class Location and the array *RouteLoc* is defined for each instance of class Route.

For the translation, the class multiplicity is required that will also be used as the upper bound on the range for the cardinality on the association ends. As we want to test the efficiency of the translation, we try out a large number for the multiplicity of each class and we discuss how SMV coped with this. Moreover, we want to determine which translation approach is more efficient, that is using 2D or 1D arrays. The multiplicity that we chose for Location is 31, with instance ids ranging from 0..30, and the multiplicity for Route is 16, with instance ids ranging from 0..15. The runtime statistics are summarised in the table below.

nuSMV Resource Usage	**2D Array**	**1D Array**
User time:	200.240 sec.	out-of-memory (1680.470 sec.)
System time:	0.280 sec.	800.100 sec.
Virtual data size (bytes allocated):	32688K	-16679K
BDD nodes allocated:	404699	242067855
BDD nodes representing init set of states:	591	-
BDD nodes representing invar set:	1	-

The SMV model with the 1D array for each instance could not be model checked with NuSMV as its state space was too big. The 2D array is a more efficient representation than the 1D array and this justifies our choice, in translation rule 19, of using an array to represent in SMV an association between the controller and an actuator. Since there is only one instance of the controller, we do not require a 2D array but as RSDS/UML specifications become more general, we do require a 2D array when the instance of the classes that are associated are greater than one.

The system that we implemented was simple, but it had a large number of instances. One can argue that the number of instances for this system is not huge if we consider a real control system such as the batch production plant [ARA$^+$01] that contains over 100 instances of valves and pumps and other components that interact with each other. However, the structure of these systems is usually static so the complexity introduced by associations can be avoided. Nevertheless, model checking RSDS/UML specifications for control systems with associations can be slow for medium sized systems and impossible for large ones.

7.8 Related Work

We survey various methods designed to verify formally UML specifications and we have divided our findings into two sections: those methods that use model checking for verification that are directly comparable to our approach and those that use other formal verification techniques. We discuss these in more detail.

7.8.1 Verifying UML designs with model checking

Methods that apply model checking to UML specifications by defining translations between their notations, often encounter difficulties when translating some of the object oriented concepts into the model checking notation because of the semantic differences between the notations. The most eminent of these arises with the translation of dynamic instantiation of classes and associations. Moreover, since the translation approach is adopted for integrating UML design and model checking, it must be shown (preferably formally) that the model produced in the target language corresponds to that in the source language and that any properties proven in the target hold in the source as well. Therefore, in this section we consider how such methods overcome difficulties when translating and whether they provide a proof of correctness for their translation. We consider the two most popular model checking languages, namely SMV and SPIN, that have been presented for model checking UML specifications.

Verifying with the SMV model checker

The approach that we have adopted for reducing the semantic gap between the RSDS/UML and SMV language is to restrict the object oriented notation before translating. The main restriction concerns the class multiplicity and the cardinality on the association ends of class diagrams. A maximum number of possible objects must be known before run-time as well as the maximum number of objects that are associated with each other. Moreover, we recommend that variables of integer type be limited to a manageable size for model checking or that data abstraction techniques must be applied to extract relevant ranges of values as for altitude values in the autopilot case study. Our objective is to model the dynamic instantiation of classes and associations as verification of these aspects has not been dealt with by other researchers.

A similar approach is also adopted by [Tan01] where a translation is presented from UML (class diagrams, object diagrams and statecharts) to SMV and the object oriented notation is restricted and the dynamic instantiation of classes and associations is not modelled. A proof of correctness with respect to UML for this translation has not been given and it would be challenging to do so as the semantics of UML have not been precisely defined.

The toolset described in [SCH02b] manages to verify UML models with the ASM model checker [Win01], that is based on SMV, without requiring a proof of correctness with respect to UML. They define precise semantics for the UML models (informally described in [UMLa]) in the Abstract State Machines (ASM) notation [Bör95, Gur91, Gur95]. Details of the semantics for UML statecharts, and activity diagrams can be found in [BCR00a, BCR00b]. The toolset is capable of static validation, syntax checking and checking whether the model satisfies the UML semantics, and dynamic validation performed by model checking, checks that some significant properties are satisfied by the model, such as safety and liveness properties.

Verifying with the SPIN model checker

Several alternative approaches for overcoming the problem of reducing the semantic gap between the languages are identified in the literature concerning methods that use SPIN for model checking UML models.

The methodology described in [dMGMP02] derives system models in PROMELA, the language of SPIN, from UML statecharts. Sequence diagrams are used as the property language rather than Object Constraint Language (OCL) that is typically used for this purpose. Even though OCL is more expressive than sequence diagrams, the latter are more comprehensible to users and can be directly mapped to finite state transition models. Their objective is to partially verify large concurrent systems in the initial design stages and are therefore concerned with the abstraction of statecharts. RSDS/UML defined two semantic views of statemachines to overcome this problem. Their current work [GMMP02a, GMMP02b, GPE02] is directed towards defining and applying abstraction techniques for model checking, that is constructing the smallest model without losing the interesting properties. This is a good attempt at reducing the state space explosion problem inherent in model checkers.

Hugo [KM02, SKM01] is a tool that also uses model checking to verify systems designed with UML. It differs from other approaches because it verifies whether the interactions of objects

expressed by a collaboration diagram are preserved in a set of statemachines. Therefore, the statemachines are translated into a PROMELA model and the collaboration diagrams are translated into Büchi automata, and then SPIN verifies the model against the automata. Compared to RSDS/UML, Hugo translates a larger set of UML statecharts, such as nested states and history states, and the execution semantics are closer to the fine-grain view of RSDS/UML statemachines. Also, the statemachines translated are a variant of UML statecharts that is based on the dynamic computation of UML statechart behaviour, while the variant of UML statecharts used by RSDS/UML is based on a pre-determined calculation of possible state transitions in response to input events. Hugo's translation of statemachines is limited to translating only one instance of a class and does not support the dynamic instantiation of objects. Moreover, the properties to be verified are limited to what can be expressed in a collaboration diagram, while RSDS/UML automatically translates constraints described as a subset of OCL. In [SKM01], the authors claim that the correctness of the translation can be done by inspection. However, the PROMELA model produced looks complicated and some discrepancies can be easily missed. Hugo also generates Java classes for animation or for the direct inclusion into a Java application.

Latella *et. al.* [LMM99a] describe a translation from a subset of UML statecharts to PROMELA for the purpose of verification. This subset of UML statecharts covers state refinement and transition priority and since the dynamic semantics (execution semantics) are not formalised in UML, the authors give an operational semantics in [LMM99b] to this subset. The translation consists of two steps. First each UML statechart diagram is translated into an intermediate representation called Extended Hierarchical Automata (EHA), hence providing formal syntax to UML statecharts that is required for verification. Secondly, the EHA are translated into PROMELA. No proof of correctness is given for the translation relative to the operational semantics. A particular example of how to model check embedded systems designed with UML is described in [MB02]. This translation is limited to a single statechart and does not translate object-oriented features of UML statecharts.

vUML [LP99] is a tool that translates UML classes and statecharts into PROMELA and represents any counter-examples produced as sequence diagrams thereby hiding completely the model checking part from the users. Again, only a subset of UML is translated, for example, history states are not considered. Moreover, vUML does not allow any run-time instantiation. The verification capabilities of vUML are limited to verifying only deadlock properties.

The Bandera Toolset [HD01] does not aim to verify UML models, but focuses on the reverse process of analysing Java programs by model checking them. Java source code is given as input to the tool together with requirements specified in Bandera's temporal specification language, which are used to generate models in the language of the following model checkers: Spin, dSpin [IS99], SMV and JPF [BHPV00]. If an error is detected it is traced back, step by step, to the Java source code. It builds on compiler techniques for reducing object oriented techniques into the language of the chosen model checker. Problems with translating dynamic type substitution (polymorphism) arise.

7.8.2 Verifying UML designs with formal verification techniques

Model checking is not the only formal technique used for verifying UML models. In this section we discuss some other techniques that are commonly used.

Alloy

Methods using model checking for verification of large systems are handicapped by the state space explosion problem. The Alloy constraint analyser [JSS00] is a tool that can analyse large object models. Its input language Alcoa (successor of Nitpick [Jac96]) is based on Z but is close to UML making transcription trivial. Alcoa contains explicit quantifiers and set comprehension, thus requiring more expert knowledge from the developers than RSDS/UML. Alloy works by translating the constraints into boolean formulas, and then applying SAT solvers. Alcoa is not decidable, so it cannot provide sound and complete analysis.

Verifying UML with B

Many researches are investigating the ability of the B method to support the analysis and verification of UML models. Some of these researches adopt the same approach as we do, by defining translations [MS00, BS00c, ML02, Tre02] that map elements from UML models of a system into a elements of the B specification language (AMN). The most common modelling notation of UML considered for the translations is class diagrams with OCL constraints. The dynamic aspect of the class described by the operations, are usually translated as well, and in some translations their behaviour is obtained from UML statecharts. The B proof tools are used to analyse and verify the OCL constraints and the class behaviour can be animated in B at the specification level. No proof of correctness with respect to the UML semantics is provided for these translations. Some [MS00] argue that they provide a formal interpretation of the UML models by expressing them in the B notation. Only [BS00c] have implemented a tool for automating the translation. However, as we have already discussed B cannot verify temporal properties.

The USE Tool

The UML-based Specification Tool (USE) [RG00] provides support for validation and verification of UML models that is based on animation. It simulates UML models and has an OCL interpreter for constraint checking. There are functions for changing the system state by destroying and creating objects or associations or setting values to attributes. By translating into B, RSDS/UML provides similar capabilities as B supports animation and monitoring of invariants and preconditions. Additional verification capabilities are provided by RSDS/UML via the translation to SMV for verifying temporal properties. The counter-examples produced by SMV for properties that are not true in that model, define sequences of behaviour which assists the developer with debugging.

UMLAUT

The Unified Modelling Language All pUrposes Transformer (UMLAUT) [HJGP99, JGF98] is a framework for verifying and generating code for distributed control systems specified using UML. A system described as UML models is transformed into an executable model for simulation by

algebraic compositions of reified elementary transformations. Validation is based on Labeled Transition Systems (LTS) that are related to the basic notation used for model checking. The transformation rules are much more complex than the translation rules presented for RSDS/UML. As a consequence, the correctness of these transformations must be difficult to prove. Moreover, semantics that formalise the transformation rules have not been developed, and this reduces our confidence in the generated code produced.

The KeY System

RSDS/UML shares common objectives with the Key System [ABB$^+$03], a CASE tool that has been enhanced with extra functionality for formal specification and verification of UML models. The interesting aspect of this approach is that formal verification is based on the axiomatic semantics of a subset of JAVA, called JAVA CARD, and thus the structure of the proof follows closely to the structure of the JAVA program. The properties to be verified are expressed using dynamic logic and are automatically generated from OCL constraints. However, this new dynamic logic is not as established as the predicate logic that RSDS/UML relies on. Moreover, RSDS has an established modular design methodology that is lacking in KeY. The Key System is also comparable to Bandera as they both work with Java programs, though the Key System verifies systems more precisely than Bandera.

7.9 Summary

In this chapter, we present the RSDS/UML semantics and describe the expected form of the RSDS/UML specifications for control systems. This form has some syntactic restrictions that are not expressed in the RSDS/UML semantics. We only consider a restricted subset of UML at the moment, but our future aim is to gradually include more models and loosen the restrictions.

We have shown how to provide verification capabilities for RSDS/UML specifications by translating them into the model checking notation SMV. We have dealt with the translation issues that arose by restricting the object oriented aspects of the UML models and by constraining the model to be finite. These issues were brought about due to the restriction of model checking being applied only to finite models and the state space explosion problem, as well as the richness of the object oriented modelling notation compared to that of SMV.

Our translation represents the following RSDS/UML elements in SMV:

- Constraints that are attached to associations and described in a succinct notation are translated explicitly into SMV for each instance.
- Dynamic classes and associations are translated respecting the class multiplicity and the cardinality on the association ends. The inheritance relationship is also translated into SMV.
- The coarse-grain semantic view of the statemachine notation is translated into SMV, where each coarse-grain step corresponds to a single SMV step.

We describe the proof of correctness of the translation, an important aspect that is often omitted by researches in the related work. This omission occurs because of the lack of formal semantics of the UML notation or because they define the semantics by translating into a notation.

The state space explosion problem inherent in model checking is the main limitation of our approach. We tested the translation on a simple railway example with a large number of instances. The optimised translation approach was model checked successfully within a reasonable amount of time.

CHAPTER 8

Conclusion

This thesis focuses on finding a way to verify the temporal properties of RSDS specifications. We proposed the use of model checking for this purpose. Model checking was integrated into the RSDS method by defining translations from RSDS specifications to the input language of the model checker. The SMV model checker was chosen as it is well suited to representing SRS statemachines and can model check both CTL and LTL properties. The main contributions of this thesis are summarised as follows:

Model checking the coarse-grain view of RSDS with SMV: The coarse-grain view of an RSDS specification is model checked in SMV by defining a translation between the notations. The translation is defined by a number of translation rules for translating each element in the coarse-grain view of an RSDS specification into corresponding elements in the input language of SMV. The translation preserves the modularity of RSDS specifications and ensures that one coarse-grain step corresponds to one SMV step. The correctness of the translation is shown with a proof. In order to construct the proof, the semantics of RSDS and SMV must be defined formally and preferably in the same formalism. We have thus consolidated the RSDS semantics and presented the semantics where the controller is derived automatically from the sensor component descriptions. Also, we provide SMV with formal semantics using the same formalism as that of RSDS, based on the operational semantics defined in [McM92b]. Model checking suffers from the state space explosion problem and therefore, it is unfeasible to model check some large systems specified in RSDS. Since decomposition approaches are applied to improve the manageability of large systems specified in RSDS, we find natural ways of applying them to the SMV models for the purpose of reducing the state space.

Model checking the fine-grain view of RSDS with SMV: Similarly, we define a translation from the fine-grain view of a RSDS specification into the input language of SMV for the purpose of model checking. The fine-grain view is more complex than the coarse-grain view, as the order of actuator events being generated is important and must be preserved. The queue structure, used in the definition of the RSDS semantics to ensure the order of actuator events, is emulated in the SMV model. A proof is constructed to assert the correctness of the translation of the fine-grain view. In addition, we show how the decomposition approaches that are applied to large systems specified in RSDS, can also be applied in a natural way to the SMV models generated.

Evaluating RSDS against SCR and PVS: The RSDS method, including the integration of the SMV model checker, is evaluated by performing a detailed comparison against the SCR and PVS methods. We develop the RSDS specification for a simple autopilot system and provide the SMV model and B specification. This autopilot system was developed previously by both the SCR and PVS methods and used as a basis for comparison. We pay particular attention to the verification techniques used by all methods involved in the comparison.

Model checking the coarse-grain view of RSDS/UML with SMV: As a starting point, we apply model checking to RSDS/UML specifications by defining a translation with a set of translation rules, from the coarse-grain view of RSDS/UML to the SMV input language. We construct a proof to demonstrate the correctness of the translation. The translation for the fine-grain view of RSDS/UML has not yet been defined, because RSDS/UML is constantly being adapted to support the development of a wider range of critical applications. However, we provide some additional translation rules to give an idea of how the RSDS/UML translation can be easily enhanced as to represent future versions of RSDS/UML.

8.1 Critical Evaluation

We evaluate the solution presented in this thesis for enhancing the verification support of RSDS specifications with respect to two sets of criteria. The first set of criteria concern the general objectives of both RSDS and RSDS/UML methods, that is, of improving the applicability and usability of formal methods. We want to evaluate how well these objectives are maintained even with the integration of model checking. The second set of criteria concern the quality of the translations.

8.1.1 Maintaining RSDS and RSDS/UML objectives

The main objectives of RSDS and RSDS/UML are to improve the applicability and usability of formal methods for developing reactive systems, whereby the latter takes an object oriented approach. These are achieved by providing a graphical notation for modelling and by automating as much of the development process as possible. Verification is supported by the B method that has been integrated into both RSDS and RSDS/UML methods by translation. Some of these objectives are part of the ideal criteria in [CW96] that tools and methods must try to satisfy.

Integration of tools or methods

No single tool or method is absolute for developing software and therefore it is recommended that a combination of approaches be applied. Moreover, it is better to integrate existing tools that are familiar rather than invent new ones. An appropriate style and meaning must be found when the approaches are used jointly [CW96].

Model checking has been integrated into the RSDS method by defining a translation for each semantic view that can be automated from RSDS specifications into the input language of SMV. The translations ensures that SMV is hidden from the user as the input and output of SMV are interpreted on the RSDS specification. The meaning of the combination of RSDS and SMV is defined formally by the proofs of correctness of the translations. The proofs guarantee that the semantics of RSDS specifications are preserved by the translations.

Similarly, model checking has been integrated with RSDS/UML by defining a translation. We have also shown that the translation preserves the semantics of RSDS/UML by giving a proof of correctness. However, we have not shown how the results are interpreted onto an RSDS/UML specification. This should not be difficult to do.

Automation and error detection

Automation in tools consists of the machine performing the required operations with no user interaction. Tools that provide a high degree of automation dramatically enhance their usability and applicability. RSDS aims to automate as much of the development process of reactive systems as possible. We have remained true to this aim with the proposal of integrating model checking with RSDS by defining translations that can be automated. The user does not need to have any prior knowledge of SMV. In addition, SMV verifies the system properties automatically. The current version of RSDS, as implemented by Kevin Lano, contains the following features:

- It allows a user to draw a statemachine model of a system.
- It assists the user with the drawing of the statemachines by choosing templates (instead of drawing them from scratch) for typical components, such as for switches with two states, and valves with three states. Once a template is chosen, the default transition and state names can be changed.
- It allows users to enter the system invariants in the specific format defined for static, operational and temporal invariants.
- Consistency and completeness checks can be performed on the statemachine model.
- B machines can be automatically generated from the statemachines and invariants.
- SMV code for the coarse-grain view of the statemachines can be automatically generated.

From the work presented in this thesis, what still needs to be implemented is:

- The mapping of any counter-examples produced from the SMV model representing coarse-grain RSDS specifications.
- The translation from fine-grain RSDS specifications to SMV.
- The mapping of any counter-examples produced from the SMV model representing fine-grain RSDS specifications.

Therefore, we can only discuss the usability and applicability of using RSDS to model coarse-grain RSDS specification and model checking them with SMV. A user can develop a coarse-grain specification for a reactive system very quickly, using the templates and easily without requiring expert knowledge of the underlying methods (graphical notation). Then, the RSDS tool automatically generates the corresponding SMV model. The SMV code is then model checked using NuSMV, which returns true if a property is true in the model, and false with a counter-example otherwise. If a counter-example is produced, it is not mapped back onto the RSDS model, but this is easy to do so from the mapping defined in this thesis. Once the RSDS specification is corrected by the user, the tool can then re-generate the SMV model. This process continues until all properties are true. We assume that this process will be similar for fine-grain RSDS specifications. Therefore, from this process, the usability and applicability of using formal methods are enhanced because the user is presented with a graphical notation for describing the specification, SMV code is generated automatically and verification occurs with a "push of a button". Moreover, automation is a desired feature for safety critical systems as it reduces user interaction that could be error prone.

Error detection is achieved early on in the development process by the RSDS tool and SMV. The RSDS tool performs consistency checking of the invariants, while SMV automatically explores the state space for any violations of the invariants in the given model. Model checking aims to detect errors, which according to [CW96] is a better approach than just certifying correctness. The difference between the detecting errors and certifying correctness that the authors in [CW96] want to emphasise, is that model checking detects errors by generating counter-examples if a property is false, thus pointing out specific contradictions present in the specification. We believe that both are equally important as detecting errors assists the user with development, while certification substantially increases the confidence in the final system developed.

Although verification is well supported in the RSDS development process, the task of formulating the system invariants is still carried out by the user. This task is usually very hard as it depends on the user knowing some logic and ensuring that the requirements are complete and consistent. RSDS provides some templates for the common format of the invariants for reactive systems and the RSDS tool checks to some extent for consistency and for completeness. However, more work is required for improving the tool support in this area.

Tool support for the RSDS/UML method is currently being implemented. It has the same aims as the RSDS tool of detecting errors early on in the development process. Consistency and completeness checking has not been implemented yet. However, an initial version of the translations to B and to Java classes have been implemented.

8.1.2 Quality of the translations

We have presented three translations to SMV, where two were from RSDS specifications and the third from RSDS/UML specifications. We want to ensure that the translations have some desirable properties that contribute to their quality. We evaluate the quality of the translations based on factors discussed in [KG02] that are broadly categorised as: semantic, syntactic and efficiency. The *semantic* factors consider whether the elements in the source language correspond closely to

the elements in the target language. In some cases, the elements cannot be directly represented in the target language and an adequate representation must be agreed upon. The *syntactic* factors consider the desirable properties of the target model such as modularity and readability. Since the target notation is a model checker, the *efficiency* of the target model produced is important to ensure that the state space explosion problem is handled.

RSDS coarse-grain translation

- *Semantic factors:* All of the elements of the RSDS coarse-grain specification were represented successfully in the SMV input language. The proof of correctness guarantees this. However, the mapping between the elements is not direct. We use the information that we already know about SRS statemachines and how they are synthesised in order to translate more efficiently. For example, we do not translate the sensor components into SMV and, to ensure that a coarse-grain step corresponds to a single SMV step, the controller and actuator transitions are synchronised in the SMV model. Thus, in SMV the generation of events is not modelled explicitly. Moreover, we choose not to translate subcontroller components into SMV, because the subcontroller and actuator transitions would be both defined in terms of controller transitions in order to ensure that a coarse-grain step corresponds to a single SMV step. This means that introducing the subcontrollers in the SMV modules does not improve the manageability of large systems in SMV and simply adds to the state space.

- *Syntactic factors:* The modularity of the coarse-grain specifications is mostly preserved in the SMV model, as subcontrollers are not explicitly expressed in SMV (only in the case of the horizontal decomposition approach, but are again limited to a single level). Each system component is represented as a separate SMV module and the dependency of the modules follow closely to that described by the DCFD. Although the SMV notation does not have any complex structuring mechanisms, the simple modularity that it provides enhances readability. If the naming conventions for the states, events and transitions are preserved in the translation, we believe that it is possible for the user to understand the SMV model without having any prior knowledge of the notation. Nevertheless, we intend to hide the SMV model from the user to make RSDS seamless.

- *Efficiency:* We need to produce SMV models that are as small as possible without misrepresenting or losing any of information from the RSDS specifications. The reason for this is to ensure that the model is small enough for model checking. Any aspects that are redundant have been removed, such as the SMV modules for sensor components and subcontrollers. Moreover, the decomposition approaches have been used to verify (if possible) subsystems independently in separate SMV programs. This can be difficult for subsystems that are dependent on each other. We overcome this problem with the introduction of virtual sensors. It would be beneficial to implement larger case studies using RSDS to determine the scalability of the translation.

RSDS fine-grain translation

- *Semantic factors:* In the fine-grain translation, there is a very close relation between the semantics of the fine-grain view of RSDS specifications and the SMV model generated. In fact, the queue structure defined in the semantics is emulated as an array in SMV that models the exact order of the actuator events. This close relation means that the proof of correctness is easier to show. This queue is implemented in a static way in SMV and all of the values at all times have to be worked out automatically.
- *Syntactic factors:* The modularity of fine-grain specifications is preserved in the SMV model. Subcontrollers are explicitly expressed as modules in SMV and the structuring of these modules is analogous to that of the RSDS components.
- *Efficiency:* The state space of the fine-grain SMV model is larger than that of the coarse-grain because of the introduction of the array for defining the exact order in which the actuator events are processed in a reaction cycle. Again, the sensor components are not represented in SMV as the controller is modelled as an amalgamation of sensor components. As with the coarse-grain, the decomposition approaches can be used to divide the system into subsystems and verify them independently as separate SMV programs. Various larger case studies should be implemented for the fine-grain as well and possibly further optimisation techniques could be identified.

RSDS/UML coarse-grain translation

- *Semantic factors:* The elements in an RSDS/UML specification were not all directly represented in the SMV model. This is because RSDS/UML has a high-level specification language with object oriented constructs, while the SMV language is low-level based on transition systems. Therefore, it is not possible to represent some of the object oriented construct in SMV. For example, the dynamic instantiation of classes and associations allowed in RSDS/UML specifications cannot be directly represented in SMV as SMV models have a fixed state space that does not dynamically change during run-time. Thus, in the SMV model, we can only emulate the dynamic creation and deletion of objects and their links. Also, SMV can only model check finite systems, and therefore it needs to know the number of objects created for each class. We ensure that this additional information (such as the maximum number of objects created in the system lifetime) is available from the RSDS/UML specification by imposing certain restrictions on RSDS/UML specifications. We give a proof of correctness for translation of the restricted RSDS/UML into SMV. A fair criticism of this work is that the benefits of using object oriented notation are lost (such as dynamic instantiation) when translating into SMV.
- *Syntactic factors:* As we already mentioned, we impose syntactic restrictions on RSDS/UML specifications. Also, we already know what the static structure of a system to be modelled with RSDS/UML consists of: a class for each actuator and controller and associations linking them. This structure is preserved by the translation as each class is represented as an

SMV module. The definition of the links between the objects are defined in the controller. Encapsulation of classes is not maintained in the SMV model as some objects directly access the attributes of other objects. However, since the user will not need to interact with the SMV model, the readability and maintainability of the SMV model generated is not vital.

- *Efficiency:* The SMV model generated by the coarse-grain RSDS/UML translation is much larger than that generated by the coarse-grain RSDS translation. This is because the links between objects in a system are explicitly defined using arrays which augment the state space of the SMV model. We showed, by testing, that the way the links are defined (i.e. 1-dimensional array versus 2-dimensional array) can significantly increase the state space of the model, making model checking infeasible. Therefore, we have chosen to translate RSDS/UML elements in the most efficient way possible.

8.2 Future Work

We give some suggestions for future work.

- **Scalability:** In this thesis we have proposed some ways of managing the verification of large systems by applying the decomposition approaches in a natural way to the SMV models generated. The largest system that was developed with RSDS and translated into SMV is the production cell. This is not enough for commenting on the extent on which these approaches improve the scalability such that model checking is achievable. Therefore, we suggest that we test these approaches extensively on “real” large systems with hundreds of components.

  RSDS/UML improves the scalability of large systems in terms of modelling multiple components with similar behaviour as a single class. In SMV, the representation of classes as SMV modules does not help address the state space explosion problem for large systems because each object is represented explicitly (i.e. as an instance of the module). Only the readability and maintainability of the system is improved with the definition of modules in SMV. Therefore, we can examine ways in which to optimise the SMV model generated for RSDS/UML specifications. The ways in which the decomposition approaches are applied to the SMV models of RSDS specification, could also be applied to RSDS/UML specification. However, further work is required to investigate what these decomposition approaches mean when applied to RSDS/UML specifications and whether they can be applied to any system (i.e. not limited to reactive systems).
- **Tool Support:** There is currently a version of the RSDS tool that provides automatic translations into B, JAVA and SMV. Only the coarse-grain SMV translation has been implemented. Therefore, the translations presented in this thesis should be implemented and also the results produced by SMV should be interpreted on the RSDS specifications. In this way, the user does not need to have any prior knowledge of SMV.

  Similarly, the translation from the coarse-grain view of RSDS/UML specification must be implemented in the RSDS/UML tool and the SMV results must be interpreted. In this thesis we have shown how counter-examples are mapped to RSDS/UML statemachines and an animator can be implemented that highlights the states and transitions that correspond to the counter-example. Furthermore, we could extend RSDS/UML to include other diagrams, such as collaboration diagrams or sequence diagrams, where the former can be used to illustrate some of the properties to be verified as in Hugo [KM02] and the latter can be used to interpret counter-examples as in vUML [LP99].

- **Extending RSDS/UML:** Our work has focused on using a restricted subset of UML for specifying reactive systems and for defining a translation to SMV for it. It would be worthwhile to extend the subset of UML being used and to loosen some of the restrictions imposed as this will improve the expressivity of the modelling language and also allow for the specification of a wider range of critical systems (i.e. not only reactive systems). For example, we can enhance the expressivity of statemachines with the addition of composite states and of class diagrams with the addition of aggregation relationships and association classes.

  Furthermore, we can investigate how concepts from the UML profile for the specification of scheduling, performance and time [UMLb] can be incorporated in RSDS/UML. Control systems often have timing devices that monitor the time taken to perform operations. Timing properties must be expressed using a suitable language, such as $P \Rightarrow AF_{\tau}Q$ in PLC specifications [LCA+02e] that means that if P is true, then there is a time within τ units in the future at which Q is true. Sequence diagrams are useful in modelling timing constraints and should be incorporation in RSDS/UML and formal temporal properties can be automatically generate from these. SMV cannot verify systems with timing properties, so we can either find a way of extending the SMV tool in order to verify with timing properties (since NuSMV is open source) or we use the model checkers that focus on verifying real-time systems (such as UPPAAL, Kronos).

8.3 Closing Remark

It is common for a collection of techniques to be combined for increasing the benefits of a method or tool. Despite the state space explosion problem and the fact that it applies only to finite state systems, model checking is a desirable debugging tool. The RSDS tool has greatly benefited from this integration as errors are detected early on in the development process and temporal properties can be verified.

Glossary

Notation	Description	
AMN	Abstract Machine Notation (B notation).	13
BDD	Binary Decision Diagrams	35
BLAST	Berkely Lazy Abstraction Software	44
CELENEC EN 50128	Railway industry standard.	9
controllable transitions	They are triggered by internal events and are represented by solid lines.	55
CTL	Computational Tree Logic	25
CTL*	A logic that embodies CTL and LTL.	25
DCFD	Data Control Flow Diagrams	51
environmental assumptions	They describe assumptions that are true about the environment.	52
external events	These are events that occur in the environment detected by sensors i.e.sensor events.	55
fault-tolerant systems	The ability of systems to operate normally despite hardware or software failure.	52
FSM	Finite State Machines	13
internal events	These are events that occur within the system i.e. they are generated by the system and are therefore also known as generated events.	55
LTL	Linear-Time Temporal Logic	25
LTS	Labelled transition systems	24
OBDD	Ordered Binary Decision Diagrams	35

Notation	Description	
operational invariants	These have the form α & $P \Rightarrow AX(Q)$ where P and Q are state constraints on sensors and actuators and α is a sensor event and AX is the temporal operator. These invariants are usually generated automatically by the RSDS tool by converting the static invariants into this form. They are then used to synthesise the control algorithm.	53
outer-level controller	A controller that is dedicated regularly checking the environmental assumptions to ensure that the system is put in a safe state if these are false.	52
PLTL	Propositional Linear Time Logic	15
ProB	A model checker for B	45
PVS	Prototype Verification Sytsem	11
RSDS	Reactive System Development Support	50
RSML	Requirements State Machine Language	12
SCR	Software Cost Reduction	12
SMV	Symbolic Model Verifier	41
SRS	Structured Reactive Systems	51
static invariants	These have the form $P \Rightarrow Q$ where P and Q consist only of constraints of current states of sensors and actuators in the system. No temporal operators or event names are used.	53
STeP	The Stanford Temporal Prover	44
system constraints	They describe the behaviour of the system.	52
TCTL	Real-time Temporal Logic	46
temporal invariants	These have the form $P \Rightarrow M(Q)$ where M is some temporal operator such as AF and AG and P and Q are state constraints on sensors and actuators.	53
UML	Unified Modelling Language	12
uncontrollable transitions	They are triggered by external events and are represented by dashed lines.	55
VIS	Verification Interacting with Synthesis	41

APPENDIX A

SMV code generated for the Fault Tolerant Production Cell

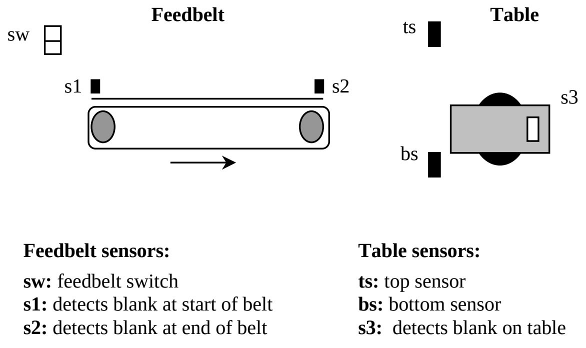


Figure A.1: The sensors for the feedbelt and table components.

A.1 The subsystem for the table components

```
MODULE main
VAR
  Cont : Controller;
  Mtablemotor : tablemotor(Cont);

SPEC
```

```
  AG((Cont.state = Off_Off_Off | Cont.state = On_Off_Off)  ->
          Mtablemotor.tablemotor = up)

SPEC
  AG(AG(Cont.event = s3on ->
           (Cont.state = Off_On_Off | Cont.state = Off_On_On |
    Cont.state = On_On_Off | Cont.state = On_On_On)) &
     AG(Mtablemotor.tablemotor = up ->
       AF(Cont.state = Off_On_Off | Cont.state = Off_On_On |
      Cont.state = On_On_Off | Cont.state = On_On_On))  ->
                   ((Cont.state = Off_Off_Off | Cont.state = Off_On_Off |
     Cont.state = On_Off_Off | Cont.state = On_On_Off)  ->
        AF(Cont.state = On_On_Off | Cont.state = Off_On_Off)))

------------------------------------

MODULE Controller
VAR
  -- state : (bs, ts, s3)
  --------------------------------
  state : { Off_Off_Off, Off_Off_On, Off_On_Off, Off_On_On, On_Off_Off,
            On_Off_On, On_On_Off, On_On_On };
  event : { bson, bsoff, tson, tsoff, s3on, s3off };

DEFINE
  CT0 := event = bson & state = Off_Off_Off;
  CT1 := event = bson & state = Off_Off_On;
  CT2 := event = bson & state = Off_On_Off;
  CT3 := event = bson & state = Off_On_On;
  CT4 := event = bsoff & state = On_Off_Off;
  CT5 := event = bsoff & state = On_Off_On;
  CT6 := event = bsoff & state = On_On_Off;
  CT7 := event = bsoff & state = On_On_On;
  CT8 := event = tson & state = Off_Off_Off;
  CT9 := event = tson & state = Off_Off_On;
  CT10 := event = tson & state = On_Off_Off;
  CT11 := event = tson & state = On_Off_On;
  CT12 := event = tsoff & state = Off_On_Off;
  CT13 := event = tsoff & state = Off_On_On;
  CT14 := event = tsoff & state = On_On_Off;
  CT15 := event = tsoff & state = On_On_On;
  CT16 := event = s3on & state = Off_Off_Off;
  CT17 := event = s3on & state = Off_On_Off;
  CT18 := event = s3on & state = On_Off_Off;
  CT19 := event = s3on & state = On_On_Off;
  CT20 := event = s3off & state = Off_Off_On;
  CT21 := event = s3off & state = Off_On_On;
  CT22 := event = s3off & state = On_Off_On;
  CT23 := event = s3off & state = On_On_On;

ASSIGN
  init(state) := Off_Off_Off;

  next(state) :=
    case
      CT0 : On_Off_Off;
      CT1 : On_Off_On;
      CT2 : On_On_Off;
```

```
      CT3 : On_On_On;
      CT4 : Off_Off_Off;
      CT5 : Off_Off_On;
      CT6 : Off_On_Off;
      CT7 : Off_On_On;
      CT8 : Off_On_Off;
      CT9 : Off_On_On;
      CT10 : On_On_Off;
      CT11 : On_On_On;
      CT12 : Off_Off_Off;
      CT13 : Off_Off_On;
      CT14 : On_Off_Off;
      CT15 : On_Off_On;
      CT16 : Off_Off_On;
      CT17 : Off_On_On;
      CT18 : On_Off_On;
      CT19 : On_On_On;
      CT20 : Off_Off_Off;
      CT21 : Off_On_Off;
      CT22 : On_Off_Off;
      CT23 : On_On_Off;
      1 : state;
    esac;

------------------------------------

MODULE tablemotor(Cont)
VAR
  tablemotor : { Off, up, down };
ASSIGN
  init(tablemotor) := up;

  next(tablemotor) :=
    case
      Cont.CT0 : up;
      Cont.CT1 : Off;
      Cont.CT2 : Off;
      Cont.CT3 : Off;
      Cont.CT4 : up;
      Cont.CT5 : down;
      Cont.CT6 : Off;
      Cont.CT7 : down;
      Cont.CT8 : Off;
      Cont.CT9 : down;
      Cont.CT10 : Off;
      Cont.CT11 : Off;
      Cont.CT12 : up;
      Cont.CT13 : down;
      Cont.CT14 : up;
      Cont.CT15 : Off;
      Cont.CT16 : down;
      Cont.CT17 : down;
      Cont.CT18 : Off;
      Cont.CT19 : Off;
      Cont.CT20 : up;
      Cont.CT21 : Off;
      Cont.CT22 : up;
      Cont.CT23 : Off;
      1 : tablemotor;
    esac;
```

A.2 The subsystem for the feedbelt components

```
MODULE main
VAR
  C : Controller;
  Mbeltmotor : beltmotor(C);

SPEC

-----------------------------------------------

MODULE Controller
VAR
  -- state : (sw, s2, Stm)
  --------------------------------
  state : { Off_Off_NotStm, Off_Off_Stm, Off_On_NotStm, Off_On_Stm,
            On_Off_NotStm, On_Off_Stm, On_On_NotStm, On_On_Stm };
  event : { swon, swoff, s2on, s2off, stmon, stmoff };

  fail-safe : boolean;


DEFINE
  CT0 := event = swon & state = Off_Off_NotStm;
  CT1 := event = swon & state = Off_Off_Stm;
  CT2 := event = swon & state = Off_On_NotStm;
  CT3 := event = swon & state = Off_On_Stm;
  CT4 := event = swoff & state = On_Off_NotStm;
  CT5 := event = swoff & state = On_Off_Stm;
  CT6 := event = swoff & state = On_On_NotStm;
  CT7 := event = swoff & state = On_On_Stm;
  CT8 := event = s2on & state = Off_Off_NotStm;
  CT9 := event = s2on & state = Off_Off_Stm;
  CT10 := event = s2on & state = On_Off_NotStm;
  CT11 := event = s2on & state = On_Off_Stm;
  CT12 := event = s2off & state = Off_On_NotStm;
  CT13 := event = s2off & state = Off_On_Stm;
  CT14 := event = s2off & state = On_On_NotStm;
  CT15 := event = s2off & state = On_On_Stm;
  CT16 := event = stmon & state = Off_Off_NotStm;
  CT17 := event = stmon & state = Off_On_NotStm;
  CT18 := event = stmon & state = On_Off_NotStm;
  CT19 := event = stmon & state = On_On_NotStm;
  CT20 := event = stmoff & state = Off_Off_Stm;
  CT21 := event = stmoff & state = Off_On_Stm;
  CT22 := event = stmoff & state = On_Off_Stm;
  CT23 := event = stmoff & state = On_On_Stm;

ASSIGN
  init(state) := Off_Off_NotStm;

  next(state) :=
    case
      CT0 : On_Off_NotStm;
```

```
      CT1 : On_Off_Stm;
      CT2 : On_On_NotStm;
      CT3 : On_On_Stm;
      CT4 : Off_Off_NotStm;
      CT5 : Off_Off_Stm;
      CT6 : Off_On_NotStm;
      CT7 : Off_On_Stm;
      CT8 : Off_On_NotStm;
      CT9 : Off_On_Stm;
      CT10 : On_On_NotStm;
      CT11 : On_On_Stm;
      CT12 : Off_Off_NotStm;
      CT13 : Off_Off_Stm;
      CT14 : On_Off_NotStm;
      CT15 : On_Off_Stm;
      CT16 : Off_Off_Stm;
      CT17 : Off_On_Stm;
      CT18 : On_Off_Stm;
      CT19 : On_On_Stm;
      CT20 : Off_Off_NotStm;
      CT21 : Off_On_NotStm;
      CT22 : On_Off_NotStm;
      CT23 : On_On_NotStm;
      1 : state;
    esac;

  init(fail-safe) := 0;

  next(fail-safe) :=
   case
      state = next(state) : 1;
        1 : fail-safe;
    esac;
---------------------------------------

MODULE beltmotor(C)
VAR
  beltmotor : { Off, On };
ASSIGN
  init(beltmotor) := Off;

  next(beltmotor) :=
    case
      C.CT0 : On;
      C.CT1 : On;
      C.CT2 : Off;
      C.CT3 : On;
      C.CT4 : Off;
      C.CT5 : Off;
      C.CT6 : Off;
      C.CT7 : Off;
      C.CT8 : Off;
      C.CT9 : Off;
      C.CT10 : Off;
      C.CT11 : On;
      C.CT12 : Off;
      C.CT13 : Off;
      C.CT14 : On;
    C.CT15 : On;
    C.CT16 : Off;
    C.CT17 : Off;
    C.CT18 : On;
    C.CT19 : On;
    C.CT20 : Off;
    C.CT21 : Off;
    C.CT22 : On;
    C.CT23 : Off;
    1 : beltmotor;
  esac;
```

APPENDIX B

SMV code generated for the Gas Burner System

B.1 Using the translation in [CAB+98]

```
MODULE main

VAR
  --Events
  event : {none, swoff, swon, fdoff, fdon};
  sw1 : boolean;
  sw2 : boolean;
  fd1 : boolean;
  fd2 : boolean;
  av_close : boolean;
  av_open : boolean;
  gv_close  : boolean;
  gv_open : boolean;
  ig_close : boolean;
  ig_open : boolean;

  --And-states
  Switch : {on, off};
  FlameDetector :{absent, present};
  Controller : {off_absent, on_absent,  off_present, on_present};
  AirValve : {open, closed};
  GasValve : {open, closed};
  Ignitor : {on, off};


DEFINE
  stable := event=none & !av_close & !av_open & !gv_close &
              !gv_close & !ig_close & !ig_open;

  in-Sys := 1;

  in-Switch := in-Sys;
  in-SwOff := in-Switch & Switch = off;
  in-SwOn := in-Switch & Switch = on;
  in-FlameDtc := in-Sys;
  in-FDAbs := in-FlameDtc & FlameDetector = absent;
  in-FDPrs := in-FlameDtc & FlameDetector = present;

  in-Cont := in-Sys;
  in-OffAbs := in-Cont & Controller = off_absent;
  in-OnAbs := in-Cont & Controller = on_absent;
  in-OffPrs := in-Cont & Controller = off_present;
  in-OnPrs := in-Cont & Controller = on_present;

  in-AirValve := in-Sys;
  in-AVopen := in-AirValve & AirValve = open;
  in-AVclosed := in-AirValve & AirValve = closed;

  in-GasValve := in-Sys;
  in-GVopen := in-GasValve & GasValve = open;
  in-GVclosed := in-GasValve & GasValve = closed;

  in-Ignitor := in-Sys;
  in-IGopen := in-Ignitor & Ignitor = open;
  in-IGclosed := in-Ignitor & Ignitor = closed;

  ST1 := in-SwOn & event = swoff;
  ST2 := in-SwOff & event = swon;
  ST3 := in-FDAbs & event = fdon;
  ST4 := in-FDPrs & event = fdoff;

  CT1 := in-OffAbs & sw1;
  CT2 := in-OnAbs & sw2;
  CT3 := in-OnPrs & sw2;
  CT4 := in-OffPrs & sw1;
  CT5 := in-OnPrs & fd2;
  CT6 := in-OnAbs & fd1;
  CT7 := in-OffAbs & fd1;
  CT8 := in-OffPrs & fd2;

  AT1 := in-AVopen & av_close;
  AT2 := in-AVclosed & av_open;
  AT3 := in-GVopen & gv_close;
  AT4 := in-GVclosed & gv_open;
  AT5 := in-IGopen & ig_close;
  AT6 := in-IGclosed & ig_open;

ASSIGN

init(Switch) := off;
next(Switch) :=
   case
     ST2 : off;
     ST1 : on;
     1  : Switch;
   esac;

init(FlameDetector) := absent;
next(FlameDetector) :=
   case
     ST3 : present;
     ST4 : absent;
```

```
     1 : FlameDetector;
   esac;

init(Controller) := off_absent;
next(Controller) :=
   case
     CT1|CT5 : on_absent;
     CT2|CT8 : off_absent;
     CT3|CT7 : off_present;
     CT4|CT6 : on_present;
     1 : Controller;
   esac;

init(AirValve) := closed;
next(AirValve) :=
   case
     AT2 : open;
     AT1 : closed;
     1 : AirValve;
   esac;

init(GasValve) := closed;
next(GasValve) :=
   case
     AT4 : open;
     AT3 : closed;
     1 : GasValve;
   esac;

init(Ignitor) := off;
next(Ignitor) :=
   case
     AT6 : on;
     AT5 : off;
     1 : Ignitor;
   esac;

--external events
next(event) :=
  case
    stable & !next(stable) : {swon, swoff, fdoff, fdon, none};
    1: none;
  esac;

--generated (internal) events

init(sw1) := 0;
next(sw1) := ST1;

init(sw2) := 0;
next(sw2) := ST2;

init(fd1) := 0;
next(fd1) := ST3;

init(fd2) := 0;
next(fd2) := ST4;
```

```
init(av_close) := 0;
next(av_close) := CT2|CT8;

init(av_open) := 0;
next(av_open) := CT1|CT7|CT4|CT6 ;

init(gv_close) := 0;
next(gv_close) := CT2|CT3 ;

init(gv_open) := 0;
next(gv_open) := CT1|CT4 ;

init(ig_close) := 0;
next(ig_close) := CT3|CT6|CT2 ;

init(ig_open) := 0;
next(ig_open) := CT7|CT1|CT5 ;

SPEC
 AG(Ignitor = on -> GasValve = open)

SPEC
 AG(GasValve = open -> AirValve = open)
```

B.2 Using the alternative translation

B.2.1 The coarse-grain view of the system

```
MODULE main
VAR
  Fd : FlameDetector;
  Sw : Switch;
  Av : AirValve(Sw,Fd);
  Gv : GasValve(Sw,Fd);
  Ig : Ignitor(Sw,Fd);

SPEC
  AG(Gv.gv = open -> Av.av = open)

SPEC
  AG(Ig.ig=on -> Gv.gv = open)

------------------------------------------------------------------------

MODULE FlameDetector

VAR
 fd: {Absent, Present};
 event : {fdon, fdoff};

DEFINE
 FDT1 := fd = Absent & event = fdon;
 FDT2 := fd = Present & event = fdon;
 FDT3 := fd = Absent & event = fdoff;
 FDT4 := fd = Present & event = fdoff;

ASSIGN
  init(fd) := Absent;
```

```
  next(fd) :=
     case
       FDT1 | FDT2 : Present;
       FDT3 | FDT4 : Absent;
       1 : fd;
     esac;

------------------------------------------------------------------------

MODULE Switch
VAR
 sw: {On, Off};
 event : {swon, swoff};

DEFINE
 SWT1 := sw = On & event = swon;
 SWT2 := sw = Off & event = swon;
 SWT3 := sw = On & event = swoff;
 SWT4 := sw = Off & event = swoff;

ASSIGN
  init(sw) := Off;
  next(sw) :=
     case
       SWT1 | SWT2 : On;
       SWT3 | SWT4 : Off;
       1 : sw;
     esac;

------------------------------------------------------------------------

MODULE AirValve(Sw, Fd)

VAR
  av : { closed, open };

DEFINE
  T1 := Sw.event = swon & Sw.sw = Off & Fd.fd = Absent;
  T2 := Sw.event = swoff & Sw.sw = On & Fd.fd = Absent;
  T4 := Sw.event = swon & Sw.sw = Off & Fd.fd = Present;
  T5 := Fd.event = fdon & Sw.sw = Off & Fd.fd = Absent;
  T6 := Fd.event = fdon & Sw.sw = On & Fd.fd = Absent;
  T8 := Fd.event = fdoff & Sw.sw = Off & Fd.fd = Present;

ASSIGN
  init(av) := closed;

  next(av) :=
    case
      T1 : open;
      T2 : closed;
      T4 : open;
      T5 : open;
      T6 : open;
      T8 : closed;
      1 : av;
    esac;
```

```
-----------------------------------------------------------------------------

MODULE GasValve(Sw, Fd)
VAR
  gv : { closed, open };

DEFINE
  GT1 := Sw.event = swon & Sw.sw = Off & Fd.fd = Absent;
  GT2 := Sw.event = swoff & Sw.sw = On & Fd.fd = Absent;
  GT3 := Sw.event = swoff & Sw.sw = On & Fd.fd = Present;
  GT4 := Sw.event = swon & Sw.sw = Off & Fd.fd = Present;

ASSIGN
  init(gv) := closed;

  next(gv) :=
    case
      GT1 : open;
      GT2 : closed;
      GT3 : closed;
      GT4 : open;
      1 : gv;
    esac;

-----------------------------------------------------------------------------

MODULE Ignitor(Sw, Fd)
VAR
  ig : { on, off };

DEFINE
  IT1 := Sw.event = swon & Sw.sw = Off & Fd.fd = Absent;
  IT2 := Sw.event = swoff & Sw.sw = On & Fd.fd = Absent;
  IT3 := Sw.event = swoff & Sw.sw = On & Fd.fd = Present;
  IT5 := Fd.event = fdon & Sw.sw = Off & Fd.fd = Absent;
  IT6 := Fd.event = fdon & Sw.sw = On & Fd.fd = Absent;
  IT7 := Fd.event = fdoff & Sw.sw = On & Fd.fd = Present;

ASSIGN
  init(ig) := off;

  next(ig) :=
    case
      IT1 : on;
      IT2 : off;
      IT3 : off;
      IT5 : off;
      IT6 : off;
      IT7 : on;
      1 : ig;
    esac;
```

B.2.2 The fine-grain view of the system

```
MODULE main
VAR
  Sw : Switch(Qu);
  Fd : FlameDetector(Qu);
```

```
  Av : AirValve(Qu);
  Gv : GasValve(Qu);
  Ig : Ignitor(Qu);
  Qu : Queue(Sw, Fd, Av, Gv, Ig);


SPEC
  AG(Gv.gv = open & Qu.is_empty = 1 -> Av.av = open)

SPEC
  AG(Ig.ig=on & Qu.is_empty = 1 -> Gv.gv = open)

SPEC
 AG(Qu.is_empty = 0 -> AF Qu.is_empty = 1)

-----------------------------------------------------------------------------

MODULE FlameDetector(q)

VAR
 fd: {Absent, Present};
 event : {fdon, fdoff};

DEFINE
 FDT1 := fd = Absent & event = fdon & q.is_empty = 1;
 FDT2 := fd = Present & event = fdon & q.is_empty = 1;
 FDT3 := fd = Absent & event = fdoff & q.is_empty = 1;
 FDT4 := fd = Present & event = fdoff & q.is_empty = 1 ;

ASSIGN
  init(fd) := Absent;
  next(fd) :=
     case
       FDT1 | FDT2 : Present;
       FDT3 | FDT4 : Absent;
       1 : fd;
     esac;

-----------------------------------------------------------------------------

MODULE Switch(q)
VAR
 sw: {On, Off};
 event : {swon, swoff};

DEFINE
 SWT1 := sw = On & event = swon  & q.is_empty = 1;
 SWT2 := sw = Off & event = swon & q.is_empty = 1;
 SWT3 := sw = On & event = swoff & q.is_empty = 1;
 SWT4 := sw = Off & event = swoff & q.is_empty = 1;

ASSIGN
  init(sw) := Off;
  next(sw) :=
     case
       SWT1 | SWT2 : On;
       SWT3 | SWT4 : Off;
       1 : sw;
     esac;

--------------------------------------------------------------------------

MODULE AirValve(q)

VAR
  av : { closed, open };

DEFINE
  AVT1 := q.is_empty = 0 & q.Q[1] = av1; -- & av = open;
  AVT2 := q.is_empty = 0 & q.Q[1] = av2; -- & av = closed;

ASSIGN
  init(av) := closed;

  next(av) :=
    case
     AVT1: closed;
AVT2: open;
1 : av;
    esac;

------------------------------------------------------------------------------

MODULE GasValve(q)
VAR
  gv : { closed, open };

DEFINE
  GVT1 := q.is_empty = 0 & q.Q[1] = gv1; -- & gv = open;
  GVT2 := q.is_empty = 0 & q.Q[1] = gv2; -- & gv = closed;

ASSIGN
  init(gv) := closed;

  next(gv) :=
    case
     GVT1: closed;
  GVT2: open;
      1 : gv;
    esac;

------------------------------------------------------------------------------

MODULE Ignitor(q)
VAR
  ig : { on, off };

DEFINE
  IGT1 := q.is_empty = 0 & q.Q[1] = ig1; --- & ig = on;
  IGT2 := q.is_empty = 0 & q.Q[1] = ig2; --- & ig = off;

ASSIGN
  init(ig) := off;

  next(ig) :=
    case
```

```
      IGT1: off;
  IGT2: on;
      1 : ig;
    esac;

------------------------------------------------------------------------------

MODULE Queue(SW, FD, AV, GV, IG)
VAR
  Q:array 1..3 of {av1, av2, gv1, gv2, ig1,ig2, null}; --- 1 = off and 2 = on
  is_empty : boolean;

DEFINE
  CT1 := SW.SWT2 & FD.fd = Absent & is_empty = 1;
  CT2 := SW.SWT3 & FD.fd = Absent & is_empty = 1 ;
  CT3 := SW.SWT3 & FD.fd = Present & is_empty = 1 ;
  CT4 := SW.SWT2 & FD.fd = Present & is_empty = 1 ;
  CT5 := FD.FDT1 & SW.sw = Off & is_empty = 1 ;
  CT6 := FD.FDT1 & SW.sw = On & is_empty = 1 ;
  CT7 := FD.FDT4 & SW.sw = On & is_empty = 1 ;
  CT8 := FD.FDT1 & SW.sw = Off & is_empty = 1 ;

ASSIGN

init(is_empty) := 1;
next(is_empty) :=
   case
      CT1 : 0;
      CT2 : 0;
      CT3 : 0;
      CT4 : 0;
      CT5 : 0;
      CT6 : 0;
      CT7 : 0;
      CT8 : 0;
      AV.AVT1 & Q[2]=null : 1;
      AV.AVT2 & Q[2]=null : 1;
      GV.GVT1 & Q[2]=null : 1;
      GV.GVT2 & Q[2]=null : 1;
      IG.IGT1 & Q[2]=null : 1;
      IG.IGT2 & Q[2]=null : 1;
      1:is_empty;
   esac;

init(Q[1]) := null;
next(Q[1]) :=
   case
      CT1 : av2;
      CT2 : ig1;
      CT3 : ig1;
      CT4 : av2;
      CT5 : ig1;
      CT6 : ig1;
      CT7 : ig2;
      CT8 : av1;
      AV.AVT1:Q[2];
      AV.AVT2:Q[2];
      GV.GVT1:Q[2];
      GV.GVT2:Q[2];
      IG.IGT1:Q[2];
      IG.IGT2:Q[2];
    1:Q[1];
esac;

init(Q[2]) := null;
next(Q[2]) :=
   case
      CT1 : gv2;
      CT2 : gv1;
      CT3 : gv1;
      CT4 : gv2;
      CT5 : av2;
      CT6 : av2;
      CT7 : null;
      CT8 : null;
      AV.AVT1:Q[3];
      AV.AVT2:Q[3];
      GV.GVT1:Q[3];
      GV.GVT2:Q[3];
      IG.IGT1:Q[3];
      IG.IGT2:Q[3];
    1:Q[2];
esac;

init(Q[3]) := null;
next(Q[3]) :=
   case
      CT1 : ig2;
      CT2 : av1;
      CT3 : null;
      CT4 : null;
      CT5 : null;
      CT6 : null;
      CT7 : null;
      CT8 : null;
      AV.AVT1:null;
      AV.AVT2:null;
      GV.GVT1:null;
      GV.GVT2:null;
      IG.IGT1:null;
      IG.IGT2:null;
    1:Q[3];
esac;
```

APPENDIX C

The SMV and B Code Generated for the Autopilot System

C.1 The SMV Code Generated for the Autopilot System

We have annotated the source with the number of the invariants from which the code was derived.

```
MODULE main
VAR
   c1 : CAScont;
   cas : CASdisplay(c1);
   c2 : ALTcont;
   mo : Mode(c2, alt, en);
   en : ALTengage(c2, mo, alt);
   fpa : FPAdisplay(c2, mo, en, alt);
   alt : ALTdisplay(c2, mo);

-- CAS properties
------------------
SPEC AG (c1.cas_event = CASdialed -> AX (cas.state = desired))
SPEC AG (AX(c1.cas_event = CASdialed) -> AF (cas.state = desired))

-- ALT properties
--------------------
SPEC AG (c2.state = away -> EF c2.state = near)
SPEC AG (en.Alt_armed = armed -> mo.mode = FPA)
SPEC AG (!( mo.mode= ATT) -> EF mo.mode = ATT)
SPEC AG (!( mo.mode= ATT) & c2.event = ATTpressed -> AF mo.mode = ATT)
SPEC AG(!(mo.mode = ATT & alt.ast = desired& fpa.fst = desired)) --from \ref{ap26}
SPEC AG(c2.event = FPAdialed -> AX(fpa.fst = desired)) --from \ref{ap6}

    -- SCR properties
-------------------
SPEC AG(mo.mode = FPA -> AX(!(mo.mode = FPA) -> fpa.fst = current)) -- from \ref{24}
SPEC AG(mo.mode = ALT -> AX(!(mo.mode = ALT) -> alt.ast = current)) -- from \ref{25}

-------------------------------------------------------
-- The CAS Subsystem
-------------------------------------------------------
```

```
MODULE CAScont
VAR
  state : {on, off};
  cas_event : {CASpressed, CASdialed, CASreached};

DEFINE
  CT1 := state = off & cas_event = CASpressed;  --from 4.1
  CT2 := state = on & cas_event = CASpressed;   --from 4.2 & 4.3
  CT3 := state = off & cas_event = CASdialed;     --from 4.4
  CT4 := state = on & cas_event = CASdialed;      --from 4.4
  CT5 := state = off & cas_event = CASreached;  --from 4.5
  CT6 := state = on & cas_event = CASreached;   --from 4.5

ASSIGN
  init(state) := off;
  next(state) :=
     case
       CT1: on;
       CT2 : off;
     1 : state;
  esac;

-------------------------------------------------------

MODULE CASdisplay(cont)
VAR
  state : {desired, current};

DEFINE
  DT1 := cont.CT2 & state = desired ;
  DT2 := (cont.CT3 | cont.CT4) & state = current;
  DT3 := (cont.CT5 | cont.CT6) & state = desired;

ASSIGN
  init(state) := current;
  next(state) :=
       case
          DT1 : current;
          DT2 : desired;
          DT3 : current;
       1 : state;
  esac;

-------------------------------------------------------
--The ALT subsystem
-------------------------------------------------------

MODULE ALTcont
VAR
   state : {near, away, at};
   event : {ATTpressed, FPApressed, ALTpressed, ALTgetsNear,
        ALTreached, FPAreached, ALTdialed, FPAdialed, ALTgetsAway};

DEFINE
    CT1 :=event =  ATTpressed & state = near;
    CT2 :=event =  ATTpressed & state = away;
    CT3 :=event =  ATTpressed & state = at;
```

```
    CT4 :=event =  FPApressed & state = near;
    CT5 :=event =  FPApressed & state = away;
    CT6 :=event =  FPApressed & state = at;

    CT7 :=event =  ALTpressed & state = near;
    CT8 :=event =  ALTpressed & state = away;
    CT9 :=event =  ALTpressed & state = at;

    CT10 :=event = ALTgetsNear & state = near;
    CT11 :=event = ALTgetsNear & state = away;
    CT12 :=event = ALTgetsNear & state = at;

    CT13 :=event = ALTreached & state = near;
    CT14 :=event = ALTreached & state = away;
    CT15 :=event = ALTreached & state = at;

    CT16 :=event = ALTdialed & state = near;
    CT17 :=event = ALTdialed & state = away;
    CT18 :=event = ALTdialed  & state = at;

    CT19 :=event = FPAdialed & state = near;
    CT20 :=event = FPAdialed & state = away;
    CT21 :=event = FPAdialed  & state = at;

    CT22 :=event = FPAreached & state = near;
    CT23 :=event = FPAreached & state = away;
    CT24 :=event = FPAreached & state = at;

    CT25 :=event = ALTgetsAway & state = near;
    CT26 :=event = ALTgetsAway & state = away;
    CT27 :=event = ALTgetsAway & state = at;

ASSIGN
  init(state):=away;
  next(state) :=
       case
     CT11 : near;
     CT12 : near;
     CT13 | CT14  : at;
        CT25 : away ;
       1 : state;
  esac;

------------------------------------------------------------------

MODULE Mode(OC, a, En)

VAR
   mode : {ATT, ALT, FPA};

DEFINE
   MT1 := !(mode = ATT) &  (OC.CT1 | OC.CT2 | OC.CT3) ; --from 4.11 & 4.12
   MT2 := !(mode = FPA) & (OC.CT4 | OC. CT5 | OC.CT6) ; --from 4.8
   MT3 := mode = FPA & (OC.CT4 | OC.CT5 | OC.CT6) ; --from 4.7
   MT4 := !(mode = ALT) & a.ast = desired & OC.CT8 ;    --from 4.15
   MT5 := !(mode = ALT) & a.ast = desired & (OC.CT7 | OC.CT9);  --from 4.24
   MT6 := mode = FPA & En.Alt_armed = armed & (OC.CT10 | OC.CT11 | OC.CT12); --from 4.16
   MT7 := mode = FPA & En.Alt_armed = armed & (OC.CT13 | OC.CT14 | OC.CT15);
   MT8 := mode = FPA & En.Alt_armed = armed & (OC.CT16 | OC.CT17 | OC.CT18);  --from 4.23
   MT9 := mode = ALT & !(a.ast = desired) &  OC.CT8 ;
   MT10 := mode = ALT & (OC.CT16 | OC.CT17 | OC.CT18); --from 4.17

ASSIGN
   init(mode) := ATT;
   next(mode) := case
            MT1 : ATT ;
            MT2 : FPA ;
            MT3 : ATT ;
            MT4 : FPA ;
            MT5 : ALT ;
            MT6 : ALT;
            MT7 : ALT ;
            MT8 : ATT;
            MT9 : FPA;
           1 : mode;
    esac;

--------------------------------------------------------------------

MODULE ALTengage(C, M, AL)

VAR
  Alt_armed : {armed, not_armed};

DEFINE
  ET1 := C.CT8 & !(M.mode = ALT) & AL.ast = desired & Alt_armed = not_armed; --from 4.15
  ET2 := (C.CT16 | C.CT17 | C.CT18) &  M.mode = ALT & Alt_armed = armed;   --from 4.22
  ET3 := (C.CT16 | C.CT17 | C.CT18) &  Alt_armed = armed;                  --from 4.23
  ET4 := (C.CT10 | C.CT11 | C.CT12) &  M.mode = FPA & Alt_armed = armed;   --from 4.16
  ET5 := !(M.mode = ATT) &  (C.CT1 | C.CT2 | C.CT3) ;   --from 4.11 & 4.12
  ET6 := M.mode = FPA & (C.CT4 | C.CT5 | C.CT6) ;       --from 4.7
  ET7 := M.mode = FPA & Alt_armed = armed & (C.CT13 | C.CT14 | C.CT15);

ASSIGN
  init(Alt_armed) := not_armed;
  next(Alt_armed) :=
     case
       ET1 : armed;
       ET2 : not_armed ;
       ET3 : not_armed ;
       ET4 : not_armed ;
       ET5 : not_armed ;
       ET6 : not_armed ;
       ET7 : not_armed ;
     1 : Alt_armed;
  esac;

--------------------------------------------------------------------

MODULE FPAdisplay(C, M, En, Al)

VAR
  fst : {desired, current};

DEFINE
   FT1 := (C.CT19 | C.CT20 | C.CT21) &  fst = desired; --from 4.6
```

```
   FT2 := (C.CT19 | C.CT20 | C.CT21) &  fst = current; --from 4.6

   FT3 := (C.CT22 | C.CT23 | C.CT24) &  fst = current; --from 4.9
   FT4 := (C.CT22 | C.CT23 | C.CT24) &  fst = desired; --from 4.9

   FT5 := (C.CT10 | C.CT11 | C.CT12) & M.mode = FPA & En.Alt_armed = armed
           & fst = desired; --from 4.16
   FT6 := (C.CT10 | C.CT11 | C.CT12) & M.mode = FPA & En.Alt_armed = armed
          & fst = current; --from 4.16

   FT7 := (C.CT1 | C.CT2 | C.CT3) & !(M.mode = ATT) & fst = desired; --from 4.12
   FT8 := (C.CT1 | C.CT2 | C.CT3) & !(M.mode = ATT) & fst = current; --from 4.12

   FT9 := (C.CT16 | C.CT17 | C.CT18) & En.Alt_armed = armed & fst = desired; --from 4.23
   FT10 := (C.CT16 | C.CT17 | C.CT18) & En.Alt_armed = armed & fst = current; --from 4.23

   FT11 := (C.CT7 | C.CT9) & !(M.mode = ALT) & Al.ast = desired & fst = desired; --from 4.24
   FT12 := (C.CT7 | C.CT9) & !(M.mode = ALT) & Al.ast = desired & fst = current; --from 4.24

   FT13 := (C.CT16 | C.CT17 | C.CT18) & En.Alt_armed = not_armed & fst = current; --from R2
   FT14 := (C.CT16 | C.CT17 | C.CT18) & En.Alt_armed = not_armed & fst = desired; --from R2

   FT15 := M.mode = FPA & (C.CT4 | C.CT5 | C.CT6) & fst = current; --from 4.7
   FT16 := M.mode = FPA & (C.CT4 | C.CT5 | C.CT6) & fst = desired; --from 4.7

   FT17 := M.mode = FPA & (C.CT13 | C.CT14 | C.CT15) & fst = current;
   FT18 := M.mode = FPA & (C.CT13 | C.CT14 | C.CT15) & fst = desired;

 ASSIGN
   init(fst) := current;
   next(fst):=
     case
       FT1 | FT2 : desired ;
       FT3 | FT4 : current ;
       FT5 | FT6 : current ;
       FT7 | FT8 : current ;
       FT9 | FT10 : current ;
       FT11 | FT12 : current ;
       FT13 | FT14 : current ;
       FT15 | FT16 : current;
       FT17 | FT18 : current;
     1: fst;
  esac;

-------------------------------------------------------------

MODULE ALTdisplay(C, M)

VAR
  ast : {desired, current};

DEFINE
  AT1 := (C.CT4 | C.CT5 | C.CT6) & !(M.mode = FPA) & ast = desired ; --from 4.8
  AT2 := (C.CT4 | C.CT5 | C.CT6) & !(M.mode = FPA) & ast = current;  --from 4.8

  AT3 := (C.CT16 | C.CT17 | C.CT18) & ast = current;  --from 4.13
  AT4 := (C.CT16 | C.CT17 | C.CT18) & ast = desired;  --from 4.13
  AT5 := (C.CT13 | C.CT14 | C.CT15) & ast = desired;  --from 4.14
  AT6 := (C.CT13 | C.CT14 | C.CT15) & ast = current;  --from 4.14

  AT7 := (C.CT1 | C.CT2 | C.CT3) & !(M.mode = ATT) & ast = desired; --from 4.12
  AT8 := (C.CT1 | C.CT2 | C.CT3) & !(M.mode = ATT) & ast = current; --from 4.12

ASSIGN
   init(ast) := current;
   next(ast):=
     case
       AT1 | AT2 : current ;
       AT3 | AT4 : desired ;
       AT5 | AT6 : current ;
       AT7 | AT8 : current ;
     1:ast;
  esac;
```

C.2 The B Code Generated for the Autopilot System

```
/*-------------------------------------------------------------------*/

MACHINE
CASModeCont

SEES
   APTypes

INCLUDES  /* actuator */
   CASdisplay

VARIABLES
   CASmode

INVARIANT
   CASmode : Switch

INITIALISATION
   CASmode := off

OPERATIONS
    CASpressed =
    IF
       CASmode = off
    THEN
       CASmode := on
    ELSE
       CASmode := off ||
       set_cur_CASdisplay
END;

/* ------------------------------*/

    CASdialed =
    IF
       CASmode = off
    THEN
       set_pre_CASdisplay
```

```
    END
END

/*-------------------------------------------------------------------*/

MACHINE
    CASdisplay

SEES
    APTypes

VARIABLES
    CASdisplay

INVARIANT
    CASdisplay : Display

INITIALISATION
    CASdisplay := current

OPERATIONS

/* The CAS dispaly is shows the given value */

    set_cur_CASdisplay =
    IF CASdisplay = desired
    THEN
       CASdisplay := current
    END;

    set_pre_CASdisplay =
    IF
       CASdisplay = current
    THEN
       CASdisplay := desired
    END
END

/*-------------------------------------------------------------------*/

MACHINE
    Altitude

SEES
    APTypes

VARIABLES
    Alt

INVARIANT
    Alt : ALT_val

INITIALISATION
    Alt := away

OPERATIONS

    AltgetsNear =
    IF
       Alt /= near
    THEN
       Alt := near
    END;

/*-------------------------------*/

    AltgetsAway =
    IF
       Alt /= away
    THEN
       Alt := away
    END;

/*------------------------------*/

    Altreached =
    IF
       Alt /= at
    THEN
       Alt := at
    END
END

/*--------------------------------------------------------------------*/

MACHINE
OtherCont

SEES
APTypes

INCLUDES   /* Actuators */
FPAdisplay,
ALTdisplay,
Altitude /* Sensor */

PROMOTES
set_Alt_away

VARIABLES
ModeState,
ALT_armed

INVARIANT
ModeState : SysState &
ALT_armed : ALTarmed &
(ALT_armed = armed => ModeState = FPAmode)


INITIALISATION
ModeState := ATTmode ||
ALT_armed := not_armed

OPERATIONS

ATT_pressed =
```

```
IF
ModeState = ALTmode or
ModeState = FPAmode
THEN
ModeState := ATTmode ||
set_cur_ALTdisplay ||
set_cur_FPAdisplay ||
ALT_armed := not_armed
END;

/* --------------------------------*/

FPA_pressed =
IF
ModeState = ALTmode or
ModeState = ATTmode
THEN
ModeState := FPAmode ||
set_cur_ALTdisplay
ELSE
ModeState := ATTmode ||
set_cur_ALTdisplay ||
set_cur_FPAdisplay
END;

/*---------------------------------*/

ALT_pressed =
IF
ModeState /= ALTmode &
ALT_display = pre_selected
THEN
IF
Alt /= away
THEN
ModeState := ALTmode ||
ALT_armed := not_armed ||
set_cur_FPAdisplay
ELSE
ModeState := FPAmode ||
ALT_armed := armed
END
END;

/*--------------------------------*/

ALT_gets_near =
IF

ModeState = FPAmode &
ALT_armed = armed
THEN
set_Alt_near ||
ModeState := ALTmode ||
set_cur_FPAdisplay ||
ALT_armed := not_armed
END;
```

```
/*---------------------------------*/


ALT_reached =
IF
ALT_armed = armed
THEN
set_Alt_at ||
set_cur_ALTdisplay ||
ModeState := ALTmode ||
set_cur_FPAdisplay ||
ALT_armed := not_armed
ELSE
set_Alt_at ||

set_cur_ALTdisplay

END;

/*-----------------------------*/


FPA_reached =
BEGIN
set_cur_FPAdisplay
END;

/*-----------------------------*/

ALT_input_dial =
IF
ModeState = ALTmode or
ALT_armed = armed
THEN
ModeState := ATTmode ||
set_pre_ALTdisplay ||
set_cur_FPAdisplay ||
ALT_armed := not_armed
ELSE IF
ModeState = FPAmode or
ModeState = ATTmode
     THEN
set_pre_ALTdisplay
     END
END;

/*-----------------------------*/

FPA_input_dial =
IF
ModeState /= FPAmode
THEN
set_pre_FPAdisplay
END

END
```

```
/*-----------------------------------------------------------------------*/

MACHINE
    ALTdisplay

SEES
    APTypes

VARIABLES
    ALTdisplay

INVARIANT
    ALTdisplay : Display

INITIALISATION
    ALTdisplay := current

OPERATIONS

/* The if state is not needed because we want to change the
value no matter what was there before */

    set_cur_ALTdisplay =
    IF ALTdisplay = desired
    THEN
       ALTdisplay := current
    END;

    /*------------------------------------*/

    set_pre_ALTdisplay =
    IF ALTdisplay = current
    THEN
       ALTdisplay := desired
    END
END

/*-----------------------------------------------------------------------*/

MACHINE
    FPAdisplay

SEES
    APTypes

VARIABLES
    FPAdisplay

INVARIANT
    FPAdisplay : Display

INITIALISATION
    FPAdisplay := current

OPERATIONS

    /* The FPA display shows the given value */
    set_cur_FPAdisplay =
    IF
       FPAdisplay = desired
    THEN
       FPAdisplay := current
    END;

    set_pre_FPAdisplay =
    IF
       FPAdisplay = current
    THEN
       FPAdisplay := desired
    END
END

/*----------------------------------------------------------------------*/

MACHINE
    APTypes

SETS
    SysState = {ALTmode, FPAmode, ATTmode};
    ALTarmed = {armed, not_armed};

    Switch = {on, off};
    Display = {desired, current}; /* actual desired = desired value */

    ALT_val = {away, near, at}

    /* away -------------- the desired value > 1200 feet away
       near - the desired value <= 1200 feet away
       at - the desired value = the actual altitude */
END

/*----------------------------------------------------------------------*/

MACHINE
    OuterCont

SEES
    APTypes

INCLUDES /* Controllers */
    OtherCont,
    CASModeCont


VARIABLES /* These are the old values that we set and compare
           against in the cycle operation.    */
    ALTsw,
    ATTsw,
    CASsw,
    FPAsw,
    desiredALT,/* Target values */
actualALT,
    Fpa

INVARIANT
```

```
    ALTsw : Switch &
    ATTsw : Switch &
    CASsw : Switch &
    FPAsw : Switch &
    desiredALT : NAT &
    actualALT : NAT &
    Fpa : NAT

INITIALISATION
    ALTsw := off ||
    ATTsw := off ||
    CASsw := off ||
    FPAsw := off ||
    desiredALT := 0 ||
    actualALT := 0 ||
    Fpa := 0

OPERATIONS
    cycle (new_ALT, new_ATT, new_CAS, new_FPA, input_alt, actual_alt,
          dialed_alt, input_fpa, actual_fpa, dialed_fpa, input_cas)=
    PRE
      new_ALT : Switch &
      new_FPA : Switch &
      new_ATT : Switch &
      new_CAS : Switch &
      input_alt : BOOL &
      input_fpa : BOOL &
      input_cas : BOOL &
      actual_alt : NAT &
      dialed_alt : NAT &  /* target value*/
      actual_fpa : NAT &
      dialed_fpa : NAT
    THEN
      skip
    END

END

/*----------------------------------------------------------------------*/

IMPLEMENTATION
    OuterCont_1

REFINES
    OuterCont

SEES
    APTypes,
    Bool_TYPE

IMPORTS
    /* These are the old values that we set and compare
       against in the cycle operation.    */

ALTsw1_Vvar(Switch),
ATTsw1_Vvar(Switch),
CASsw1_Vvar(Switch),
FPAsw1_Vvar(Switch),
desiredALT1_Nvar(3000),/* Target values */
actualALT1_Nvar(3000),
Fpa1_Nvar(3000)

INITIALISATION
ALTsw1_STO_VAR(off) ;
ATTsw1_STO_VAR(off);
CASsw1_STO_VAR(off);
FPAsw1_STO_VAR(off);
desiredALT1_STO_NVAR(0);
actualALT1_STO_NVAR(0);
Fpa1_STO_NVAR(0)

OPERATIONS
cycle(new_ALT, new_ATT, new_CAS, new_FPA, input_alt, actual_alt,
  dialed_alt, input_fpa, actual_fpa, dialed_fpa, input_cas) =
    VAR
      alt_sw1, att_sw1,
      cas_sw1, fpa_sw1,
      desired, actual,
      fpa1

    IN
      alt_sw1 <-- ALTsw1_VAL_VAR;
      att_sw1 <-- ATTsw1_VAL_VAR;
      cas_sw1 <-- CASsw1_VAL_VAR;
      fpa_sw1 <-- FPAsw1_VAL_VAR;
      desired <-- desiredALT1_VAL_NVAR;
      actual <-- actualALT1_VAL_NVAR;
      fpa1 <-- Fpa1_VAL_NVAR;

      IF
        (new_ALT = on & alt_sw1 = off)
      THEN
        ALTpressed
      END ;


      IF
        (new_ATT = on & att_sw1 = off)
      THEN
        ATTpressed
      END ;


      IF
        (new_CAS = on & cas_sw1 = off)
      THEN
        CASpressed
      END ;


      IF
        (new_FPA = on & fpa_sw1 = off)
      THEN
        FPApressed
      END ;
```

```
      /* Check whether a new CAS has been dialed in */
      IF
        input_cas = TRUE
      THEN
        CASdialed
      END ;

      /* Check whether a new FPA has been dialed in */
      IF
        input_fpa = TRUE
      THEN
        FPAdialed ;
        Fpa1_STO_NVAR(dialed_fpa);
        fpa1 <-- Fpa1_VAL_NVAR
      END ;

      /* Checks whether the actual FPA has reached the target FPA */
      IF
        actual_fpa = fpa1
      THEN
        FPAreached
      END ;

      /* Check whether a new altitude has been dialed in */
      IF
        input_alt = TRUE
      THEN
        ALTdialed ;
        desiredALT1_STO_NVAR(dialed_alt);
        desired <-- desiredALT1_VAL_NVAR
      END;

      /* Update the actualALT so that we can ensure that the
operations maintain the invariants */
      IF
         actual_alt /= actual
      THEN
         actualALT1_STO_NVAR(actual_alt);
         actual <-- actualALT1_VAL_NVAR
      END ;

      /* Checks the realtionship between the actual and the
         target desiredALT  */
     IF
        desiredALT <= 1200 + actual
     THEN
        ALTgetsNear
     ELSE IF
           desired = actual
        THEN
           ALTreached
        ELSE
           AltgetsAway
        END
     END
    END ;
    /* Update the sensor values (switches) */
    ALTsw1_STO_VAR(new_ALT)  ;
    ATTsw1_STO_VAR(new_ATT) ;
    CASsw1_STO_VAR(new_CAS) ;
    FPAsw1_STO_VAR(new_FPA)

END
END
```

APPENDIX D

SMV code generated for RSDS/UML specifications

D.1 SMV code for the gas burner system

The alive variable has been added to the property descriptions as we are only interested in the value of actuator instances that are alive.

```
MODULE main
VAR
  C : Controller(Av, Gv, Ig);
  Av : Valve(C,1);
  Gv : Valve(C,2);
  Ig : Igniter(C);

SPEC
  AG((Gv.alive = 1 & Av.alive = 1) -> (Gv.va = open  -> Av.va = open ))

SPEC
  AG((Ig.alive & Gv.alive) -> (Ig.ig=on -> Gv.va = open))

SPEC
  AG(Ig.alive -> C.CtoIG)

SPEC
  AG(Av.alive -> C.CtoV[1])

SPEC
  AG(Gv.alive -> C.CtoV[2])

--------------------------------------------------------------------------

MODULE Controller(Va, Vg, iG)
VAR
  state : { Off_Absent, On_Absent, Off_Present, On_Present, newActs, Shutdown };
  event : { swon, swoff, fdon, fdoff, shutdown, newV, newIG, start, none};
  idV : 1..2;

  CtoV : array 1..2 of boolean;
  CtoIG : boolean; --only one instance of each

DEFINE
  CT1 := event = swon & state = Off_Absent & Va.alive & Vg.alive & iG.alive;
  CT2 := event = swoff & state = On_Absent & Va.alive & Vg.alive & iG.alive;
  CT3 := event = swoff & state = On_Present & Va.alive & Vg.alive & iG.alive;
  CT4 := event = swon & state = Off_Present & Va.alive & Vg.alive & iG.alive;
  CT5 := event = fdon & state = Off_Absent & Va.alive & Vg.alive & iG.alive;
  CT6 := event = fdon & state = On_Absent & Va.alive & Vg.alive & iG.alive;
  CT7 := event = fdoff & state = On_Present & Va.alive & Vg.alive & iG.alive;
  CT8 := event = fdoff & state = Off_Present & Va.alive & Vg.alive & iG.alive;
  CT9 := event = newV & idV = 1 & state = newActs & Va.alive & Vg.alive & iG.alive;
  CT10 := event = newV & idV = 2 & state = newActs & Va.alive & Vg.alive & iG.alive;
  CT11 := event = newIG & state = newActs;
  CT12 := event = start & state = newActs;
  CT13 := event = shutdown & state = Off_Absent;
  CT14 := event = shutdown & idV= 1 & state = Shutdown;
  CT15 := event = shutdown & idV= 2 & state = Shutdown;

ASSIGN
  init(state) := newActs;
  next(state) :=
    case
      CT1 : On_Absent;
      CT2 : Off_Absent;
      CT3 : Off_Present;
      CT4 : On_Present;
      CT5 : Off_Present;
      CT6 : On_Present;
      CT7 : On_Absent;
      CT8 : Off_Absent;
      CT9 : newActs;
      CT10 : newActs;
  CT11 : newActs;
  CT12 : Off_Absent;
  CT13 : Shutdown;
      1 : state;
    esac;

   init(CtoIG) := 0;
   next(CtoIG) :=
      case
   CT11 : 1;
   CT13 : 0;
   1: CtoIG;
   esac;

   init(CtoV[1]) := 0;
   next(CtoV[1]) :=
      case
      CT9  : 1;
  CT14 : 0;
  1 : CtoV[1];
    esac;

   init(CtoV[2]) := 0;
   next(CtoV[2]) :=
      case
```

```
      CT10 : 1;
      CT15 : 0;
  1 : CtoV[2];
    esac;



----------------------------------------------------------------------------

MODULE Valve(C,id)

VAR
  va : { closed, open };
  alive : boolean;

DEFINE
  AT1 := C.CT1 & id = 1;
  AT2 := C.CT2 & id = 1;
  AT3 := C.CT3 & id = 2;
  AT4 := C.CT4 & id = 1;
  AT5 := C.CT5 & id = 1;
  AT6 := C.CT6 & id = 1;
  AT7 := C.CT8 & id = 1;
  AT8 := C.CT9 & id = 1;
  AT9 := C.CT10 & id = 2;
  AT10 := C.CT1 & id = 2;
  AT11 := C.CT2 & id = 2;
  AT12 := C.CT4 & id = 2;
  AT13 := C.CT10 & id = 2;


ASSIGN
  init(va) := closed;
  next(va) :=
    case
  -- for air-valve --
   -------------------
      AT1: open;
      AT2: closed;
      AT4: open;
      AT5: open;
      AT6: open;
      AT7: closed;
      AT8: closed;


  -- for gas valve --
  -------------------
      AT10: open;
      AT11: closed;
      AT3: closed;
      AT12: open;
      AT13: closed;

      1 : va;
    esac;

  init(alive) := 0;
  next(alive) :=
    case
    C.CT9 & id = C.idV : 1;
  C.CT10 & id = C.idV : 1;
  C.CT14 & id = C.idV : 0;
  C.CT15 & id = C.idV : 0;
      1:alive;
    esac;


-----------------------------------------------------------------------------

MODULE Igniter(C)
VAR
  ig : { on, off };
  alive : boolean;

DEFINE
  AT1 := C.CT1;
  AT2 := C.CT2;
  AT3 := C.CT3;
  AT4 := C.CT5;
  AT5 := C.CT6;
  AT6 := C.CT7;
  AT7 := C.CT9;
  AT8 := C.CT10;
  AT9 := C.CT11;
ASSIGN
  init(ig) := off;
  next(ig) :=
    case
      C.CT1 : on;
      C.CT2 : off;
      C.CT3 : off;
      C.CT5 : off;
      C.CT6 : off;
      C.CT7 : on;
      C.CT9 : off;
      C.CT10 : off;
  C.CT11 : off;
      1 : ig;
    esac;

  init(alive) := 0;
  next(alive) :=
    case
    C.CT11 : 1;
  C.CT13 : 0;
      1:alive;
    esac;
```

D.2 SMV code for the Railway System (2D Array)

We have removed some of the similar sections of SMV code for improving its representation.

```
MODULE main
VAR
```

```
  L0: Location(C,0);
  L1: Location(C,1);
  L2: Location(C,2);

  ...

  L27: Location(C,27);
  L28: Location(C,28);
  L29: Location(C,29);
  L30: Location(C,30);

  R0: Route(C,0);
  R1: Route(C,1);
  R2: Route(C,2);
  R3: Route(C,3);
  R4: Route(C,4);
  R5: Route(C,5);
  R6: Route(C,6);
  R7: Route(C,7);
  R8: Route(C,8);
  R9: Route(C,9);
  R10: Route(C,10);
  R11: Route(C,11);
  R12: Route(C,12);
  R13: Route(C,13);
  R14: Route(C,14);
  R15: Route(C,15);

  C: Controller(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16,
  L17,L18,L19,L20,L21,L22,L23,L24,L25,L26,L27,L28,L29,L30,R0,R1,R2,R3,R4,
  R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15);

SPEC
  AG(C.Location_Route[0][0] & L0.location = occupied -> R0.route = occupied)
SPEC
  AG(C.Location_Route[0][1] & L0.location = occupied -> R1.route = occupied)
SPEC
  AG(C.Location_Route[0][2] & L0.location = occupied -> R2.route = occupied)
SPEC
  AG(C.Location_Route[0][3] & L0.location = occupied -> R3.route = occupied)
SPEC
  AG(C.Location_Route[0][4] & L0.location = occupied -> R4.route = occupied)
SPEC
  AG(C.Location_Route[0][5] & L0.location = occupied -> R5.route = occupied)
SPEC
  AG(C.Location_Route[0][6] & L0.location = occupied -> R6.route = occupied)
SPEC
s  AG(C.Location_Route[0][7] & L0.location = occupied -> R7.route = occupied)
SPEC
  AG(C.Location_Route[0][8] & L0.location = occupied -> R8.route = occupied)
SPEC
  AG(C.Location_Route[0][9] & L0.location = occupied -> R9.route = occupied)
SPEC
  AG(C.Location_Route[0][10] & L0.location = occupied -> R10.route = occupied)
SPEC
  AG(C.Location_Route[0][11] & L0.location = occupied -> R11.route = occupied)
SPEC
  AG(C.Location_Route[0][12] & L0.location = occupied -> R12.route = occupied)
```

```
SPEC
  AG(C.Location_Route[0][13] & L0.location = occupied -> R13.route = occupied)
SPEC
  AG(C.Location_Route[0][14] & L0.location = occupied -> R14.route = occupied)
SPEC
  AG(C.Location_Route[0][15] & L0.location = occupied -> R15.route = occupied)

  ...


SPEC
  AG(C.Location_Route[30][0] & L30.location = occupied -> R0.route = occupied)
SPEC
  AG(C.Location_Route[30][1] & L30.location = occupied -> R1.route = occupied)
SPEC
  AG(C.Location_Route[30][2] & L30.location = occupied -> R2.route = occupied)
SPEC
  AG(C.Location_Route[30][3] & L30.location = occupied -> R3.route = occupied)
SPEC
  AG(C.Location_Route[30][4] & L30.location = occupied -> R4.route = occupied)
SPEC
  AG(C.Location_Route[30][5] & L30.location = occupied -> R5.route = occupied)
SPEC
  AG(C.Location_Route[30][6] & L30.location = occupied -> R6.route = occupied)
SPEC
  AG(C.Location_Route[30][7] & L30.location = occupied -> R7.route = occupied)
SPEC
  AG(C.Location_Route[30][8] & L30.location = occupied -> R8.route = occupied)
SPEC
  AG(C.Location_Route[30][9] & L30.location = occupied -> R9.route = occupied)
SPEC
  AG(C.Location_Route[30][10] & L30.location = occupied -> R10.route = occupied)
SPEC
  AG(C.Location_Route[30][11] & L30.location = occupied -> R11.route = occupied)
SPEC
  AG(C.Location_Route[30][12] & L30.location = occupied -> R12.route = occupied)
SPEC
  AG(C.Location_Route[30][13] & L30.location = occupied -> R13.route = occupied)
SPEC
  AG(C.Location_Route[30][14] & L30.location = occupied -> R14.route = occupied)
SPEC
  AG(C.Location_Route[30][15] & L30.location = occupied -> R15.route = occupied)


--------------------------------------------------------------

MODULE Controller(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16,L17,
L18,L19,L20,L21,L22,L23,L24,L25,L26,L27,L28,L29,L30,R0,R1,R2,R3,R4,R5,R6,R7,R8,
R9,R10,R11,R12,R13,R14,R15)

VAR
  event: {locationoccupied,locationunoccupied,newL, newR, killL, killR, link};
  locationid: 0..30;
  routeid: 0..15;
  Location_Route: array 0..30 of array 0..15 of boolean;

DEFINE
  CT1 := event = locationoccupied;
```

```
  CT2 := event = locationunoccupied;

ASSIGN
  init(Location_Route[0][0]) := 0;
  init(Location_Route[0][1]) := 0;
  init(Location_Route[0][2]) := 0;
  init(Location_Route[0][3]) := 0;
  init(Location_Route[0][4]) := 0;
  init(Location_Route[0][5]) := 0;
  init(Location_Route[0][6]) := 0;
  init(Location_Route[0][7]) := 0;
  init(Location_Route[0][8]) := 0;
  init(Location_Route[0][9]) := 0;
  init(Location_Route[0][10]) := 0;
  init(Location_Route[0][11]) := 0;
  init(Location_Route[0][12]) := 0;
  init(Location_Route[0][13]) := 0;
  init(Location_Route[0][14]) := 0;
  init(Location_Route[0][15]) := 0;

  ...

  init(Location_Route[30][0]) := 0;
  init(Location_Route[30][1]) := 0;
  init(Location_Route[30][2]) := 0;
  init(Location_Route[30][3]) := 0;
  init(Location_Route[30][4]) := 0;
  init(Location_Route[30][5]) := 0;
  init(Location_Route[30][6]) := 0;
  init(Location_Route[30][7]) := 0;
  init(Location_Route[30][8]) := 0;
  init(Location_Route[30][9]) := 0;
  init(Location_Route[30][10]) := 0;
  init(Location_Route[30][11]) := 0;
  init(Location_Route[30][12]) := 0;
  init(Location_Route[30][13]) := 0;
  init(Location_Route[30][14]) := 0;
  init(Location_Route[30][15]) := 0;


  next(Location_Route[0][0]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 0 & R0.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 0): 0;
      1: Location_Route[0][0];
    esac;

   next(Location_Route[0][1]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 1 & R1.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 1): 0;
      1: Location_Route[0][1];
    esac;

   next(Location_Route[0][2]) :=
    case
```

```
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 2 & R2.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 2): 0;
      1: Location_Route[0][2];
    esac;

next(Location_Route[0][3]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 3 & R3.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 3): 0;
      1: Location_Route[0][3];
    esac;

next(Location_Route[0][4]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 4 & R4.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 4): 0;
      1: Location_Route[0][4];
    esac;

next(Location_Route[0][5]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 5 & R5.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 5): 0;
      1: Location_Route[0][5];
    esac;

next(Location_Route[0][6]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 6 & R6.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 6): 0;
      1: Location_Route[0][6];
    esac;

next(Location_Route[0][7]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 7 & R7.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 7): 0;
      1: Location_Route[0][7];
    esac;

    next(Location_Route[0][8]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 8 & R8.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 8): 0;
      1: Location_Route[0][8];
    esac;

next(Location_Route[0][9]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 9 & R9.alive : 1;
```

```
      (event = killL & locationid = 0) | (event = killR & routeid = 9): 0;
      1: Location_Route[0][9];
    esac;

next(Location_Route[0][10]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 10 & R10.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 10): 0;
      1: Location_Route[0][10];
    esac;

next(Location_Route[0][11]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 11 & R11.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 11): 0;
      1: Location_Route[0][11];
    esac;

next(Location_Route[0][12]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 12 & R12.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 12): 0;
      1: Location_Route[0][12];
    esac;

next(Location_Route[0][13]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 13 & R13.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 13): 0;
      1: Location_Route[0][13];
    esac;

next(Location_Route[0][14]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 14 & R14.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 14): 0;
      1: Location_Route[0][14];
    esac;

next(Location_Route[0][15]) :=
    case
      event = link & locationid = 0 & L0.location = unoccupied & L0.alive &
      routeid = 15 & R15.alive : 1;
      (event = killL & locationid = 0) | (event = killR & routeid = 15): 0;
      1: Location_Route[0][15];
    esac;

...

next(Location_Route[30][0]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 0 & R0.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 0): 0;
      1: Location_Route[30][0];
    esac;

next(Location_Route[30][1]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 1 & R1.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 1): 0;
      1: Location_Route[30][1];
    esac;

next(Location_Route[30][2]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 2 & R2.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 2): 0;
      1: Location_Route[30][2];
    esac;

next(Location_Route[30][3]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 3 & R3.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 3): 0;
      1: Location_Route[30][3];
    esac;

next(Location_Route[30][4]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 4 & R4.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 4): 0;
      1: Location_Route[30][4];
    esac;

next(Location_Route[30][5]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 5 & R5.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 5): 0;
      1: Location_Route[30][5];
    esac;

next(Location_Route[30][6]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 6 & R6.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 6): 0;
      1: Location_Route[30][6];
    esac;

next(Location_Route[30][7]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 7 & R7.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 7): 0;
      1: Location_Route[30][7];
```

```
    esac;

next(Location_Route[30][8]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 8 & R8.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 8): 0;
      1: Location_Route[30][8];
    esac;

next(Location_Route[30][9]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 9 & R9.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 9): 0;
      1: Location_Route[30][9];
    esac;

next(Location_Route[30][10]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 10 & R10.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 10): 0;
      1: Location_Route[30][10];
    esac;

next(Location_Route[30][11]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 11 & R11.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 11): 0;
      1: Location_Route[30][11];
    esac;

next(Location_Route[30][12]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 12 & R12.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 12): 0;
      1: Location_Route[30][12];
    esac;

next(Location_Route[30][13]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 13 & R13.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 13): 0;
      1: Location_Route[30][13];
    esac;

next(Location_Route[30][14]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 14 & R14.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 14): 0;
      1: Location_Route[30][14];
    esac;
```

```
next(Location_Route[30][15]) :=
    case
      event = link & locationid = 30 & L30.location = unoccupied & L30.alive &
      routeid = 15 & R15.alive : 1;
      (event = killL & locationid = 30) | (event = killR & routeid = 15): 0;
      1: Location_Route[30][15];
    esac;

---------------------------------

MODULE Location(C,id)
VAR
  location: {occupied,unoccupied};
  alive : boolean;

ASSIGN
  init(location) := unoccupied;

  next(location) :=
    case
      C.CT1 & C.locationid = id & C.routeid = 0 & C.Location_Route[id][0]:
      occupied;
      C.CT1 & C.locationid = id & C.routeid = 1 & C.Location_Route[id][1]:
      occupied;
      C.CT1 & C.locationid = id & C.routeid = 2 & C.Location_Route[id][2]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 3 & C.Location_Route[id][3]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 4 & C.Location_Route[id][4]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 5 & C.Location_Route[id][5]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 6 & C.Location_Route[id][6]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 7 & C.Location_Route[id][7]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 8 & C.Location_Route[id][8]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 9 & C.Location_Route[id][9]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 10 & C.Location_Route[id][10]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 11 & C.Location_Route[id][11]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 12 & C.Location_Route[id][12]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 13 & C.Location_Route[id][13]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 14 & C.Location_Route[id][14]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 15 & C.Location_Route[id][15]:
      occupied;

  C.CT2 & C.locationid = id & C.routeid = 0 & C.Location_Route[id][0]:
      unoccupied;
      C.CT2 & C.locationid = id & C.routeid = 1 & C.Location_Route[id][1]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 2 & C.Location_Route[id][2]:
```

```
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 3 & C.Location_Route[id][3]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 4 & C.Location_Route[id][4]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 5 & C.Location_Route[id][5]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 6 & C.Location_Route[id][6]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 7 & C.Location_Route[id][7]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 8 & C.Location_Route[id][8]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 9 & C.Location_Route[id][9]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 10 & C.Location_Route[id][10]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 11 & C.Location_Route[id][11]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 12 & C.Location_Route[id][12]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 13 & C.Location_Route[id][13]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 14 & C.Location_Route[id][14]:
      unoccupied;
   C.CT2 & C.locationid = id & C.routeid = 15 & C.Location_Route[id][15]:
      unoccupied;

     1: location;
    esac;

  init(alive) := 0;

  next(alive) :=
           case
                C.event = newL & C.locationid = id : 1;
                C.event = killL & C.locationid = id : 0;
               1:alive;
          esac;

---------------------------------

MODULE Route(C,id)
VAR
  route: {occupied, unoccupied};
  alive : boolean;

ASSIGN
  init(route) := unoccupied;

-- If CT1 occurs then and there is a connection then we know that
-- L0.lcoation = occupied and we need to update all the routes state to
-- occupied if they are connected to L0
  next(route) :=
    case
      C.CT1 & C.locationid = 0 & C.Location_Route[0][id]: occupied;
      C.CT1 & C.locationid = 1 & C.Location_Route[1][id]: occupied;
      C.CT1 & C.locationid = 2 &C.Location_Route[2][id]: occupied;
  C.CT1 & C.locationid = 3 &C.Location_Route[3][id]: occupied;
  C.CT1 & C.locationid = 4 &C.Location_Route[4][id]: occupied;
  C.CT1 & C.locationid = 5 &C.Location_Route[5][id]: occupied;
  C.CT1 & C.locationid = 6 & C.Location_Route[6][id]: occupied;
      C.CT1 & C.locationid = 7 & C.Location_Route[7][id]: occupied;
      C.CT1 & C.locationid = 8 &C.Location_Route[8][id]: occupied;
  C.CT1 & C.locationid = 9 &C.Location_Route[9][id]: occupied;
  C.CT1 & C.locationid = 10 &C.Location_Route[10][id]: occupied;
  C.CT1 & C.locationid = 11 &C.Location_Route[11][id]: occupied;
  C.CT1 & C.locationid = 12 & C.Location_Route[12][id]: occupied;
      C.CT1 & C.locationid = 13 & C.Location_Route[13][id]: occupied;
      C.CT1 & C.locationid = 14 &C.Location_Route[14][id]: occupied;
  C.CT1 & C.locationid = 15 &C.Location_Route[15][id]: occupied;
  C.CT1 & C.locationid = 16 &C.Location_Route[16][id]: occupied;
  C.CT1 & C.locationid = 17 &C.Location_Route[17][id]: occupied;
  C.CT1 & C.locationid = 18 & C.Location_Route[18][id]: occupied;
      C.CT1 & C.locationid = 19 & C.Location_Route[19][id]: occupied;
      C.CT1 & C.locationid = 20 &C.Location_Route[20][id]: occupied;
  C.CT1 & C.locationid = 21 &C.Location_Route[21][id]: occupied;
  C.CT1 & C.locationid = 22 &C.Location_Route[22][id]: occupied;
  C.CT1 & C.locationid = 23 &C.Location_Route[23][id]: occupied;
  C.CT1 & C.locationid = 24 & C.Location_Route[24][id]: occupied;
      C.CT1 & C.locationid = 25 & C.Location_Route[25][id]: occupied;
      C.CT1 & C.locationid = 26 &C.Location_Route[26][id]: occupied;
  C.CT1 & C.locationid = 27 &C.Location_Route[27][id]: occupied;
  C.CT1 & C.locationid = 28 &C.Location_Route[28][id]: occupied;
  C.CT1 & C.locationid = 29 &C.Location_Route[29][id]: occupied;
  C.CT1 & C.locationid = 30 & C.Location_Route[30][id]: occupied;

      1: route;
    esac;

   init(alive) := 0;
   next(alive) :=
           case
                C.event = newR & C.routeid =id : 1;
                C.event = killR & C.routeid =id : 0;
               1:alive;
          esac;
```

D.3 SMV code for the Railway System (1D Array)

Again, we have removed some of the similar sections of SMV code for improving its representation.

```
MODULE main
VAR
  L0: Location(C,0);
  L1: Location(C,1);
  L2: Location(C,2);
  L3: Location(C,3);
  ...

  L26: Location(C,26);
  L27: Location(C,27);
  L28: Location(C,28);
  L29: Location(C,29);
```

```
  L30: Location(C,30);

  R0: Route(C,0);
  R1: Route(C,1);
  R2: Route(C,2);
  R3: Route(C,3);
  R4: Route(C,4);
  R5: Route(C,5);
  R6: Route(C,6);
  R7: Route(C,7);
  R8: Route(C,8);
  R9: Route(C,9);
  R10: Route(C,10);
  R11: Route(C,11);
  R12: Route(C,12);
  R13: Route(C,13);
  R14: Route(C,14);
  R15: Route(C,15);

  C: Controller(R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,L0,L1,L2,L3,L4,
L5,L6,L7,L8,L9, L10,L11,L12,L13,L14,L15,L16,L17,L18,L19,L20,L21,L22,L23,L24,L25,L26,L27,
L28,L29,L30);

SPEC
  AG(L0.LocRoute[0] & R0.RouteLoc[0] & L0.location = occupied -> R0.route = occupied)
SPEC
  AG(L0.LocRoute[1] & R1.RouteLoc[0] & L0.location = occupied -> R1.route = occupied)
SPEC
  AG(L0.LocRoute[2] & R2.RouteLoc[0] & L0.location = occupied -> R2.route = occupied)
SPEC
  AG(L0.LocRoute[3] & R3.RouteLoc[0] & L0.location = occupied -> R3.route = occupied)
SPEC
  AG(L0.LocRoute[4] & R4.RouteLoc[0] & L0.location = occupied -> R4.route = occupied)
SPEC
  AG(L0.LocRoute[5] & R5.RouteLoc[0] & L0.location = occupied -> R5.route = occupied)
SPEC
  AG(L0.LocRoute[6] & R6.RouteLoc[0] & L0.location = occupied -> R6.route = occupied)
SPEC
  AG(L0.LocRoute[7] & R7.RouteLoc[0] & L0.location = occupied -> R7.route = occupied)
SPEC
  AG(L0.LocRoute[8] & R8.RouteLoc[0] & L0.location = occupied -> R8.route = occupied)
SPEC
  AG(L0.LocRoute[9] & R9.RouteLoc[0] & L0.location = occupied -> R9.route = occupied)
SPEC
  AG(L0.LocRoute[10] & R10.RouteLoc[0] & L0.location = occupied -> R10.route = occupied)
SPEC
  AG(L0.LocRoute[11] & R11.RouteLoc[0] & L0.location = occupied -> R11.route = occupied)
SPEC
  AG(L0.LocRoute[12] & R12.RouteLoc[0] & L0.location = occupied -> R12.route = occupied)
SPEC
  AG(L0.LocRoute[13] & R13.RouteLoc[0] & L0.location = occupied -> R13.route = occupied)
SPEC
  AG(L0.LocRoute[14] & R14.RouteLoc[0] & L0.location = occupied -> R14.route = occupied)
SPEC
  AG(L0.LocRoute[15] & R15.RouteLoc[0] & L0.location = occupied -> R15.route = occupied)

 ...
```

```
---------------------------------

MODULE Controller(R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,L0,L1,L2,L3,L4,
L5,L6,L7,L8,L9, L10,L11,L12,L13,L14,L15,L16,L17,L18,L19,L20,L21,L22,L23,L24,L25,L26,L27,
L28,L29,L30)

VAR
  event: {locationoccupied, locationunoccupied, newL, newR, killL, killR, link};
  locationid: 0..30;
  routeid: 0..15;

DEFINE
  CT1 := event = locationoccupied;
  CT2 := event = locationunoccupied;

  CT3 := event = link & L0.location = unoccupied & L0.alive & R0.alive;
  CT4 := event = link & L0.location = unoccupied & L0.alive & R1.alive;
  CT5 := event = link & L0.location = unoccupied & L0.alive & R2.alive;
  CT6 := event = link & L0.location = unoccupied & L0.alive & R3.alive;
  CT7 := event = link & L0.location = unoccupied & L0.alive & R4.alive;
  CT8 := event = link & L0.location = unoccupied & L0.alive & R5.alive;
  CT9 := event = link & L0.location = unoccupied & L0.alive & R6.alive;
  CT10 := event = link & L0.location = unoccupied & L0.alive & R7.alive;
  CT11 := event = link & L0.location = unoccupied & L0.alive & R8.alive;
  CT12 := event = link & L0.location = unoccupied & L0.alive & R9.alive;
  CT13 := event = link & L0.location = unoccupied & L0.alive & R10.alive;
  CT14 := event = link & L0.location = unoccupied & L0.alive & R11.alive;
  CT15 := event = link & L0.location = unoccupied & L0.alive & R12.alive;
  CT16 := event = link & L0.location = unoccupied & L0.alive & R13.alive;
  CT17 := event = link & L0.location = unoccupied & L0.alive & R14.alive;
  CT18 := event = link & L0.location = unoccupied & L0.alive & R15.alive;

  ...

  CT484 := event = link & L30.location = unoccupied & L30.alive & R0.alive;
  CT485 := event = link & L30.location = unoccupied & L30.alive & R1.alive;
  CT486 := event = link & L30.location = unoccupied & L30.alive & R2.alive;
  CT487 := event = link & L30.location = unoccupied & L30.alive & R3.alive;
  CT488 := event = link & L30.location = unoccupied & L30.alive & R4.alive;
  CT489 := event = link & L30.location = unoccupied & L30.alive & R5.alive;
  CT490 := event = link & L30.location = unoccupied & L30.alive & R6.alive;
  CT491 := event = link & L30.location = unoccupied & L30.alive & R7.alive;
  CT492 := event = link & L30.location = unoccupied & L30.alive & R8.alive;
  CT493 := event = link & L30.location = unoccupied & L30.alive & R9.alive;
  CT494 := event = link & L30.location = unoccupied & L30.alive & R10.alive;
  CT495 := event = link & L30.location = unoccupied & L30.alive & R11.alive;
  CT496 := event = link & L30.location = unoccupied & L30.alive & R12.alive;
  CT497 := event = link & L30.location = unoccupied & L30.alive & R13.alive;
  CT498 := event = link & L30.location = unoccupied & L30.alive & R14.alive;
  CT499 := event = link & L30.location = unoccupied & L30.alive & R15.alive;

---------------------------------

MODULE Location(C,id)
VAR
  location: {occupied,unoccupied};
  alive : boolean;
  LocRoute : array 0..15 of boolean;
```

```
ASSIGN
  init(LocRoute[0]) := 0;
  init(LocRoute[1]) := 0;
  init(LocRoute[2]) := 0;
  init(LocRoute[3]) := 0;
  init(LocRoute[4]) := 0;
  init(LocRoute[5]) := 0;
  init(LocRoute[6]) := 0;
  init(LocRoute[7]) := 0;
  init(LocRoute[8]) := 0;
  init(LocRoute[9]) := 0;
  init(LocRoute[10]) := 0;
  init(LocRoute[11]) := 0;
  init(LocRoute[12]) := 0;
  init(LocRoute[13]) := 0;
  init(LocRoute[14]) := 0;
  init(LocRoute[15]) := 0;

  next(LocRoute[0]) :=
    case
      C.CT3 & C.locationid = id & id = 0 : 1;
  C.CT19 & C.locationid = id & id = 1 : 1;
  C.CT35 & C.locationid = id & id = 2 : 1;
  C.CT51 & C.locationid = id & id = 3 : 1;
  C.CT67 & C.locationid = id & id = 4 : 1;
  C.CT83 & C.locationid = id & id = 5 : 1;
  C.CT100 & C.locationid = id & id = 6 : 1;
  C.CT116 & C.locationid = id & id = 7 : 1;
  C.CT132 & C.locationid = id & id = 8 : 1;
  C.CT148 & C.locationid = id & id = 9 : 1;
  C.CT164 & C.locationid = id & id = 10 : 1;
  C.CT180 & C.locationid = id & id = 11 : 1;
  C.CT196 & C.locationid = id & id = 12 : 1;
  C.CT212 & C.locationid = id & id = 13 : 1;
  C.CT228 & C.locationid = id & id = 14 : 1;
  C.CT244 & C.locationid = id & id = 15 : 1;
  C.CT260 & C.locationid = id & id = 16 : 1;
      C.CT276 & C.locationid = id & id = 17 : 1;
  C.CT292 & C.locationid = id & id = 18 : 1;
  C.CT308 & C.locationid = id & id = 19 : 1;
  C.CT324 & C.locationid = id & id = 20 : 1;
  C.CT340 & C.locationid = id & id = 21 : 1;
  C.CT356 & C.locationid = id & id = 22 : 1;
  C.CT372 & C.locationid = id & id = 23 : 1;
  C.CT388 & C.locationid = id & id = 24 : 1;
  C.CT404 & C.locationid = id & id = 25 : 1;
  C.CT420 & C.locationid = id & id = 26 : 1;
  C.CT436 & C.locationid = id & id = 27 : 1;
  C.CT452 & C.locationid = id & id = 28 : 1;
  C.CT468 & C.locationid = id & id = 29 : 1;
  C.CT484 & C.locationid = id & id = 30 : 1;
      (C.event = killL & C.locationid = id) | (C.event = killR & C.routeid = 0): 0;
      1: LocRoute[0];
    esac;

    ...
```

```
next(LocRoute[15]) :=
    case
  C.CT18 & C.locationid = id & id = 0 : 1;
  C.CT34 & C.locationid = id & id = 1 : 1;
  C.CT50 & C.locationid = id & id = 2 : 1;
  C.CT66 & C.locationid = id & id = 3 : 1;
  C.CT82 & C.locationid = id & id = 4 : 1;
  C.CT98 & C.locationid = id & id = 5 : 1;
  C.CT115 & C.locationid = id & id = 6 : 1;
  C.CT131 & C.locationid = id & id = 7 : 1;
  C.CT147 & C.locationid = id & id = 8 : 1;
  C.CT162 & C.locationid = id & id = 9 : 1;
  C.CT179 & C.locationid = id & id = 10 : 1;
  C.CT195 & C.locationid = id & id = 11 : 1;
  C.CT211 & C.locationid = id & id = 12 : 1;
  C.CT227 & C.locationid = id & id = 13 : 1;
  C.CT243 & C.locationid = id & id = 14 : 1;
  C.CT259 & C.locationid = id & id = 15 : 1;
  C.CT275 & C.locationid = id & id = 16 : 1;
      C.CT291 & C.locationid = id & id = 17 : 1;
  C.CT307 & C.locationid = id & id = 18 : 1;
  C.CT323 & C.locationid = id & id = 19 : 1;
  C.CT339 & C.locationid = id & id = 20 : 1;
  C.CT353 & C.locationid = id & id = 21 : 1;
  C.CT371 & C.locationid = id & id = 22 : 1;
  C.CT387 & C.locationid = id & id = 23 : 1;
  C.CT403 & C.locationid = id & id = 24 : 1;
  C.CT419 & C.locationid = id & id = 25 : 1;
  C.CT435 & C.locationid = id & id = 26 : 1;
  C.CT451 & C.locationid = id & id = 27 : 1;
  C.CT467 & C.locationid = id & id = 28 : 1;
  C.CT483 & C.locationid = id & id = 29 : 1;
  C.CT499 & C.locationid = id & id = 30 : 1;
      (C.event = killL & C.locationid = id) | (C.event = killR & C.routeid = 15): 0;
      1: LocRoute[15];
    esac;


  init(location) := unoccupied;
  next(location) :=
    case
      C.CT1 & C.locationid = id & C.routeid = 0 & LocRoute[0]:
      occupied;
      C.CT1 & C.locationid = id & C.routeid = 1 & LocRoute[1]:
      occupied;
      C.CT1 & C.locationid = id & C.routeid = 2 & LocRoute[2]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 3 & LocRoute[3]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 4 & LocRoute[4]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 5 & LocRoute[5]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 6 & LocRoute[6]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 7 & LocRoute[7]:
      occupied;
  C.CT1 & C.locationid = id & C.routeid = 8 & LocRoute[8]:
```

```
    occupied;
C.CT1 & C.locationid = id & C.routeid = 9 & LocRoute[9]:
    occupied;
C.CT1 & C.locationid = id & C.routeid = 10 & LocRoute[10]:
    occupied;
C.CT1 & C.locationid = id & C.routeid = 11 & LocRoute[11]:
    occupied;
C.CT1 & C.locationid = id & C.routeid = 12 & LocRoute[12]:
    occupied;
C.CT1 & C.locationid = id & C.routeid = 13 & LocRoute[13]:
    occupied;
C.CT1 & C.locationid = id & C.routeid = 14 & LocRoute[14]:
    occupied;
C.CT1 & C.locationid = id & C.routeid = 15 & LocRoute[15]:
    occupied;

C.CT2 & C.locationid = id & C.routeid = 0 & LocRoute[0]:
    unoccupied;
    C.CT2 & C.locationid = id & C.routeid = 1 & LocRoute[1]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 2 & LocRoute[2]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 3 & LocRoute[3]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 4 & LocRoute[4]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 5 & LocRoute[5]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 6 & LocRoute[6]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 7 & LocRoute[7]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 8 & LocRoute[8]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 9 & LocRoute[9]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 10 & LocRoute[10]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 11 & LocRoute[11]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 12 & LocRoute[12]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 13 & LocRoute[13]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 14 & LocRoute[14]:
    unoccupied;
 C.CT2 & C.locationid = id & C.routeid = 15 & LocRoute[15]:
    unoccupied;
   1: location;
  esac;

init(alive) := 0;
next(alive) :=
         case
              C.event = newL & C.locationid = id : 1;
              C.event = killL & C.locationid = id : 0;
             1:alive;
        esac;
```

```
----------------------------------

MODULE Route(C,id)
VAR
  route: {occupied, unoccupied};
  alive : boolean;
  RouteLoc : array 0..30 of boolean;

ASSIGN
 init(RouteLoc[0]) := 0;
  init(RouteLoc[1]) := 0;
  init(RouteLoc[2]) := 0;
  init(RouteLoc[3]) := 0;
  init(RouteLoc[4]) := 0;
  init(RouteLoc[5]) := 0;
  init(RouteLoc[6]) := 0;
  init(RouteLoc[7]) := 0;
  init(RouteLoc[8]) := 0;
  init(RouteLoc[9]) := 0;
  init(RouteLoc[10]) := 0;
  init(RouteLoc[11]) := 0;
  init(RouteLoc[12]) := 0;
  init(RouteLoc[13]) := 0;
  init(RouteLoc[14]) := 0;
  init(RouteLoc[15]) := 0;
  init(RouteLoc[16]) := 0;
  init(RouteLoc[17]) := 0;
  init(RouteLoc[18]) := 0;
  init(RouteLoc[19]) := 0;
  init(RouteLoc[20]) := 0;
  init(RouteLoc[21]) := 0;
  init(RouteLoc[22]) := 0;
  init(RouteLoc[23]) := 0;
  init(RouteLoc[24]) := 0;
  init(RouteLoc[25]) := 0;
  init(RouteLoc[26]) := 0;
  init(RouteLoc[27]) := 0;
  init(RouteLoc[28]) := 0;
  init(RouteLoc[29]) := 0;
  init(RouteLoc[30]) := 0;


  next(RouteLoc[0]) :=
   case
C.CT3 & C.routeid = id & id = 0 : 1;
    C.CT4 & C.routeid = id & id = 0 : 1;
    C.CT5 & C.routeid = id & id = 0 : 1;
    C.CT6 & C.routeid = id & id = 0 : 1;
    C.CT7 & C.routeid = id & id = 0 : 1;
    C.CT8 & C.routeid = id & id = 0 : 1;
    C.CT9 & C.routeid = id & id = 0 : 1;
    C.CT10 & C.routeid = id & id = 0 : 1;
    C.CT11 & C.routeid = id & id = 0 : 1;
    C.CT12 & C.routeid = id & id = 0 : 1;
    C.CT13 & C.routeid = id & id = 0 : 1;
    C.CT14 & C.routeid = id & id = 0 : 1;
    C.CT15 & C.routeid = id & id = 0 : 1;
```

```
    C.CT16 & C.routeid = id & id = 0 : 1;
    C.CT17 & C.routeid = id & id = 0 : 1;
    C.CT18 & C.routeid = id & id = 0 : 1;
    (C.event = killL & C.locationid = 0) | (C.event = killR & C.routeid = 0): 0;
    1: RouteLoc[0];
   esac;

   ...

  next(RouteLoc[30]) :=
   case
    C.CT484 & C.routeid = id & id = 0 : 1;
    C.CT485 & C.routeid = id & id = 0 : 1;
    C.CT486 & C.routeid = id & id = 0 : 1;
    C.CT487 & C.routeid = id & id = 0 : 1;
    C.CT488 & C.routeid = id & id = 0 : 1;
    C.CT489 & C.routeid = id & id = 0 : 1;
    C.CT490 & C.routeid = id & id = 0 : 1;
    C.CT491 & C.routeid = id & id = 0 : 1;
    C.CT492 & C.routeid = id & id = 0 : 1;
    C.CT493 & C.routeid = id & id = 0 : 1;
    C.CT494 & C.routeid = id & id = 0 : 1;
    C.CT495 & C.routeid = id & id = 0 : 1;
    C.CT496 & C.routeid = id & id = 0 : 1;
    C.CT497 & C.routeid = id & id = 0 : 1;
    C.CT498 & C.routeid = id & id = 0 : 1;
    (C.event = killL & C.locationid = 30) | (C.event = killR & C.routeid = id): 0;
    1: RouteLoc[30];
   esac;


  init(route) := unoccupied;

-- If CT1 occurs then and there is a connection then we know that
-- L0.lcoation = occupied and we need to update all the routes state to
-- occupied if they are connected to L0
  next(route) :=
    case
      C.CT1 & C.locationid = 0 & RouteLoc[0]: occupied;
      C.CT1 & C.locationid = 1 & RouteLoc[1]: occupied;
      C.CT1 & C.locationid = 2 & RouteLoc[2]: occupied;
  C.CT1 & C.locationid = 3 & RouteLoc[3]: occupied;
  C.CT1 & C.locationid = 4 & RouteLoc[4]: occupied;
  C.CT1 & C.locationid = 5 & RouteLoc[5]: occupied;
  C.CT1 & C.locationid = 6 & RouteLoc[6]: occupied;
      C.CT1 & C.locationid = 7 & RouteLoc[7]: occupied;
      C.CT1 & C.locationid = 8 & RouteLoc[8]: occupied;
  C.CT1 & C.locationid = 9 & RouteLoc[9]: occupied;
  C.CT1 & C.locationid = 10 & RouteLoc[10]: occupied;
  C.CT1 & C.locationid = 11 & RouteLoc[11]: occupied;
  C.CT1 & C.locationid = 12 & RouteLoc[12]: occupied;
      C.CT1 & C.locationid = 13 & RouteLoc[13]: occupied;
      C.CT1 & C.locationid = 14 & RouteLoc[14]: occupied;
  C.CT1 & C.locationid = 15 & RouteLoc[15]: occupied;
  C.CT1 & C.locationid = 16 & RouteLoc[16]: occupied;
  C.CT1 & C.locationid = 17 & RouteLoc[17]: occupied;
  C.CT1 & C.locationid = 18 & RouteLoc[18]: occupied;
      C.CT1 & C.locationid = 19 & RouteLoc[19]: occupied;
    C.CT1 & C.locationid = 20 & RouteLoc[20]: occupied;
C.CT1 & C.locationid = 21 & RouteLoc[21]: occupied;
C.CT1 & C.locationid = 22 & RouteLoc[22]: occupied;
C.CT1 & C.locationid = 23 & RouteLoc[23]: occupied;
C.CT1 & C.locationid = 24 & RouteLoc[24]: occupied;
    C.CT1 & C.locationid = 25 & RouteLoc[25]: occupied;
    C.CT1 & C.locationid = 26 & RouteLoc[26]: occupied;
C.CT1 & C.locationid = 27 & RouteLoc[27]: occupied;
C.CT1 & C.locationid = 28 & RouteLoc[28]: occupied;
C.CT1 & C.locationid = 29 & RouteLoc[29]: occupied;
C.CT1 & C.locationid = 30 & RouteLoc[30]: occupied;
    1: route;
  esac;

 init(alive) := 0;
 next(alive) :=
         case
              C.event = newR & C.routeid =id : 1;
              C.event = killR & C.routeid =id : 0;
             1:alive;
        esac;
```

Bibliography

[501] CELENEC EN 50128. Railway applications - software for railway control and protection systems.

[ABB+03] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, February 2003.

[Abr96] J. Abrial. *The B Boook: Assigning programs to meanings.* Cambridge University Press, 1996.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[AGM00] Rajeev Alur, Radu Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 280–295. Springer-Verlag, 2000.

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[AHM+98] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.

[AJKV97] Rajeev Alur, Lalita Jategaonkar Jagadeesan, Joseph J. Kott, and James E. Von Olnhausen. Model-checking of real-time systems: a telecommunications application: experience report. In *Proceedings of the 19th international conference on Software engineering*, pages 514–524. ACM Press, 1997.

[AK86] K R Apt and D C Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[AK95] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 220–231, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[AKY99] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 169–178. Springer-Verlag, 1999.

[AL01] K. Androutsopoulos and K. Lano. Reactive system specification and verification in RSDS. In *Proceedings of the workshop on automated verification of critical systems (AVOCS'01)*, Oxford, UK, 2001.

[AM98] J. Abrial and L. Mussat. Introducing dynamic constraints in B. In *Second Conference on the B Method, LNCS 1393, Nantes, France*, pages 83–128. Springer Verlag, 1998.

[And96] Charles Andr. Representation and analysis of reactive behaviors: A synchronous approach. In *Invited talk at CESA'96, IEEE-SMC,*, July 1996.

[And02] Charles Andr. SyncCharts: A visual representation of reactive behaviors. Technical report, University Nice Sophia Antipolis, 2002.

[ARA$^+$01] Sanchez A., G. E. Rotstein, N. Alsop, J. P. Bromberg, C. Gollain, S. Sorensen, S. Macchietto, and C. Jakeman. Improving the development of event-driven control systems in the batch processing industry. A case study. *Submitted Elsevier Science, ISA Transactions*, 2001.

[Ate] Atelier B. $www.dmi.usherb.ca/documentations - logiciels/atelierb/fr/atb - 01.htm/$.

[AY01] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[Bae04] J.C.M. Baeten. A brief history of process algebra. Technical Report Rapport CSR 04-02, Vakgroep Informatica, Technische Universiteit Eindhoven, 2004.

[BB99] M. Büchi and R. Back. Compositional symmetric sharing in B. In *Proceedings of FM*. Springer-Verlag LNCS, 1999.

[BBC$^+$96] Nikolaj Bjrner, Anca Browne, Eddie Chang, Michael Coln, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Toms E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *International Conference on Computer Aided Verification, Lecture Notes in Computer Science*, volume 1102, pages 415–418. Springer-Verlag, 1996.

[BBF$^+$01] Beatrice Bérard, Michel Bidoit, Alain Finkel, Francois Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Verlag, August 2001.

[BCC98] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. *Lecture Notes in Computer Science*, 1536:81–102, 1998.

[BCJ02] F. Bellegarde, S. Chouali, and J. Julliand. Verification of dynamic constraints for B event systems under fairness assumptions. In *Proceeding of ZB 2002: Formal Specification and Development in Z and B*. Springer, August 2002.

[BCM90] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *In Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[BCR00a] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In Teodor Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.

[BCR00b] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 223–241. Springer-Verlag, 2000.

[BCY02] L. Brim, J. Crhova, and K. Yorav. Using assumptions to distribute ctl model checking, 2002.

[BDJK00] F. Bellegarder, C. Darlot, J. Julliand, and O. Kouchnarenko. Reformulate dynamic properties during B refinement and forget variants and loop invariants. In *Proceeding of ZB 2000: Formal Specification and Development in Z and B*, August 2000.

[BDK$^+$02] Jrgen Bohn, Werner Damm, Jochen Klose, Adam Moik, and Hartmut Wittke. Modeling and validating train system applications using Statemate and Live Sequence Charts. In H. Ehrig, B. J. Krmer, and A. Ertas, editors, *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002), Society for Design and Process Science*, Pasadena, California, June 2002.

[BDL$^+$01] Gerd Behrmann, Alexandre David, Kim G. Larsen, Oliver Mller, Paul Pettersson, and Wang Yi. UPPAAL - present and future. In *Proc. of 40th IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[Ber98] Grard Berry. The foundations of esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte*. MIT Press, 1998.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(3):34–41, 1995.

[BH96] Ramesh Bharadwaj and Connie Heitmeyer. Applying the SCR requirements specification method to practical systems: A case study. In *Presented at the 21st Software Engineering Workshop, NASA GSFC, Greenbelt MD, USA, Dec 4-5*, 1996.

[BH97a] Ramesh Bharadwaj and Connie Heitmeyer. Applying the SCR requirements method to a simple autopilot. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop*, September 1997.

[BH97b] Ramesh Bharadwaj and Constance L. Heitmeyer. Verifying eSCR requirements specifications using state exploration. In *Proceedings of First ACM SIGPLAN Workshop on Automatic Analysis of Software, Paris, France*, January 1997.

[BH99] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. In *Proceedings of Automated Software Engineering, 6, 37-68*, 1999.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - a second generation of a Java model checker. In *Workshop on Advances in Verification*, July 2000.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. In *Information and control*, 1984.

[Bör95] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.

[BPR96] D. Bert, M.-L. Potet, and Y. Rouzaud. A study on components and assembly primitives in B. In *Proceedings of 1st Conference on the B method*, pages 47–62, 1996.

[BR91] Juan Bicarregui and Brian Ritchie. Reasoning about VDM developments using the VDM support tool in mural. In S. Prehn and W. J. Toetenel, editors, *VDM '91 – Formal Software Development Methods*, pages 371–388. Springer-Verlag, October 1991.

[BR01] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, 2001.

[BR04] Purandar Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. Technical report, ACM-class: D.2.4 Software/Program Verification, 2004.

[Bry85] Randal E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd ACM/IEEE conference on Design automation*, 1985.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic manipulation with ordered binary decision diagrams. In *ACM Computing Surveys 24*, pages 293–318, 1992.

[BS00a] Ramesh Bharadwaj and Steve Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proceedings of the International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Lecture Notes in Computer Science, Berlin, Germany, 2000. Springer-Verlag.

[BS00b] Michael Butler and Colin Snook. Tool-supported use of UML for constructing B specifications. In *Technical Report*, 2000.

[BS00c] Michael Butler and Colin Snook. Verifying dynamic properties of UML models by translation to the B language and toolkit. In *In Proceedings UML 2000 Workshop, Dynamic Behaviour in UML Models: Semantic Questions*, 2000.

[BTo] BToolkit B-Core(UK)Ltd. $http://www.b-core.com/$.

[But96] Ricky W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. In *NASA Technical Memorandum 110255*, May 1996.

[CAB$^+$98] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CDH$^+$00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *In Logic of Programs: Workshop Lecture Notes in Computer Science*, volume 131, Yorktown Heights, New York, 1981. Springer-Verlag.

[CGB86] E M Clarke, O Grumberg, and M C Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM Press, 1986.

[CGH$^+$93] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and

R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[CGH97] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In *Formal Methods in System Design*, volume 10(1), pages 57–71, February 1997.

[CGL93] E. M. Clarke, O. Grumberg, and D. E. Long. Verification Tools for Finite State Concurrent Systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency-Reflections and Perspectives*, volume 803, pages 124–175, Noordwijkerhout, Netherlands, 1993. Springer-Verlag.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, 1995.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[CH00] E. M. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical report, Carnegie Mellon University, 2000. Technical report.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science Pacific Grove*, pages 353–62, 1989.

[CM02] M. Calder and A. Miller. Five ways to use symmetry and induction in the verification of networks of processes using model-checking. In *Proceedings of Automated Verification of Critical Systems (AVoCS)*, 2002.

[Com93] International Electrotechnical Commission. IEC 61131: Programmable controllers - part 3: Programming languages, 1993.

[Com99] International Electrotechnical Commission. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety related systems, 1999.

[COR$^+$95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM Press, 1987.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CVWY92] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. In *ACM Computing Surveys,28(4):626-643*, 1996.

[D. 96] D. L. Dill. The murphi verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 390–393, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[dAAG$^+$00] L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang. MOCHA: Exploiting modularity in model checking, August 2000. $http://www-cad.eecs.berkeley.edu/mocha$.

[Day98] Nancy Ann Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, Department of Computer Science, Univercity of British Columbia, 1998.

[DBMM00] T. Dimitrakos, J. Bicarregui, B. Matthews, and T. Maibaum. Compositional structuring in the B-method: a logical viewpoint of the static context. In *ZB2000, International Conference of B and Z Users*, York, UK, August-September 2000.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[dMGMP02] Mara del Mar Gallardo, Pedro Merino, and Ernesto Pimentelis. Debugging UML designs with model checking. In *In Journal of Object Technology, vol. 1, no. 2,*, pages 101–117, July-August 2002. $http://www.jot.fm/issues/issue\_2002\_07/article1$.

[DN87] D. Bjørner, C.B. Jones, M. Mac an Airchinnigh and E. J. Neuhold, editors. *VDM '87 VDM – A Formal Method at Work*, volume 252 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1987.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *In Proceedings of Hybrid Systems III, Verification and Control,Lecture Notes in Computer Science 1066*. Springer-Verlag, 1996.

[Dov] DOVE manual. *http : //www.dsto.defence.gov.au/esrl/itd/dove/*.

[DT96] Jonathan Draper and Helen Treharne. The refinement of embedded software with the B-method. In *Northern Formal Methods Workshops in Computer Science*, Bradford, 1996. Springer-Verlag.

[EH86] E. Allen Emerson and Joseph Y. Halpern. “Sometimes” and “not never” revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[Eme90] E. Allen Emerson. Temporal and modal logic. pages 995–1072, 1990.

[ES96] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[FM91] J. Fiadeiro and T. Maibaum. Describing, structuring and implementing objects. In J. De Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object Oriented Languages, LNCS 489*, pages 274–310. Springer-Verlag, 1991.

[FM92] Jose Luiz Fiadeiro and T. S. E. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[FSMS92] J.L. Fiadeiro, C. Sernadas, T. Maibaum, and A. Sernadas. Describing and structuring objects for conceptual schema development. In P.Loucopoulos and R.Ziccari, editors, *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development*, pages 117–138. John Wiley, 1992.

[GH99] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT FSE*, pages 146–162, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters : Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[GHP92] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. *Formal Methods in System Design: An International Journal*, 7(3):227–241, November 1992.

[Gia99] Dimitra Giannakopoulou. *Model checking for concurrrent software architectures*. PhD thesis, Imperial College, London, 1999.

[GLM99] S. Gnesi, Diego Latella, and Mieke Massink. Model checking UML statechart diagrams using JACK. In Raymond Paul and Catherine Meadows, editors, *Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 1999.

[GMMP02a] M. M. Gallardo, J. Martnez, P. Merino, and E. Pimentel. A tool for abstraction in model checking. In *7th International Workshop on Formal Methods for Industrial Critical Systems,Electronic Notes in Theoretical Computer Science*, volume 66.2, July 2002.

[GMMP02b] M.M Gallardo, J. Martinez, P. Merino, and E. Pimentel. aSPIN: Extending SPIN with abstraction. In *9th International SPIN Workshop,LNCS 2318*, pages 254–258, Grenoble, France, 2002.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. of Computer Aided Verification*. Spinger, 1990.

[God97a] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[God97b] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *In Proc. 9 th International Conference on Computer Aided Verication (CAV), LNCS 1254*. Springer-Verlag, 1997.

[GPE02] M. M. Gallardo, P.Merino, and E.Pimentel. Refinement of LTL formulas for abstract model checking. In *Proceedings of the 9th International Static Analysis Symposium SAS ’02, Lecture notes in Computer Science*, volume 2477, September 2002.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GR] J. Groote and M. Reniers. Algebraic process verification.

[Gro00] The VDM Tool Group. VDM++ Method Guidelines. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk /pub/vdmtools/doc/guidelines_letter.pdf.

[Gur91] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Hal93] N. Halbwachs. A tutorial of lustre. 1993.

[HB00] C. Heitmeyer and R. Bharadwaj. Applying the SCR requirements method to the light control case study. In *Journal of Universal Computer Science (JUCS)*, August 2000.

[HBGL95] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, Gaithersburg, MD, June 25-29, pp. 109-122.*, 1995.

[HD01] John Hatcliff and Matthew Dwyer. Using the Bandera Tool set to model-check properties of concurrent java software. In *Proceedings of CONCUR 2001 (invited tutorial paper)*, June 2001.

[Hei02] C. Heitmeyer. Encyclopedia of software engineering. In *Two Volumes, John J. Marciniak, editor*. ISBN: 0-471-02895-9, January 2002.

[HHK96] R.H. Hardin, Z. Harel, and R. P. Kurshan. COSPAN. In *In 8th International Conference on Computer Aided Verification CAV'96, LNCS 1102*, pages 421–427. Springer-Verlag, 1996.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[HJGP99] Wai Ming Ho, Jean-Marc Jquel, Alain Le Guennec, and Francois Pennaneac'h. UMLAUT: An extendible UML transformation framework. In *Automated Software Engineering*, pages 275–278, 1999.

[HJL96] Connie Heitmeyer, Ralph Jeffords, and Bruce Labaw. Automated consistency checking of requirements specifications. In *ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 3*, pages 231–261, July 1996.

[HJM$^+$02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science 2404*, pages 526–538. Springer-Verlag, 2002.

[HLN$^+$90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.

[HLSC01] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic test generation from statecharts using model checking, 2001.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. In *ACM Transactions on Software Engineering and Methodology*, pages 5(4):293–333, October 1996.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol95] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, 1995. Chapman & Hall.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HP85] D. Hare1 and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, New York, Tokyo, 1985. Springer-Verlag.

[HP96] G.J. Holzmann and D. Peled. The state of SPIN. In *In CAV'96: 8th International Conference on Computer Aided Verification,*, volume 1102, pages 385–389. LNCS, 1996.

[HR99] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), December 1999. LNCS 1742, Springer Verlag.

[HR00] Michael R A Huth and H. Mark D Ryan. *Logic in computer science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[HTZ96] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 208–219, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[IS99] R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In *In Proc. of the 6th SPIN Workshop, LNCS 1680*, pages 261 – 276, September 1999.

[Jac96] Daniel Jackson. Nitpick: A checkable specification language. In *In Proceedings of the Workshop on Formal Methods in Software Practice*, San Diego, January 1996.

[Jav] The source for Java technology. *http : //java.sun.com/*.

[JGF98] Jean-Marc Jézéquel, Alain Le Guennec, and cois Pennaneac'h. Fran˙Validating distributed software modelled with UML. In *In Proc. Int. Workshop UML98*, Mulhouse, France, June 1998.

[JJFM92] Claude Jard, T. Jeron, J. C. Fernandez, and L. Mounier. On-the-fly verification of finite transition systems. Technical Report RR-1861, INRIA, 1992.

[JMM99] J. Julliand, P.A. Masson, and H. Mountassir. Modular verification of dynamic properties for reactive systems. In *International Workshop on Integrated Formal Methods, York, Great Britain*, pages 89–108. Springer, 1999.

[JS93] Jeffrey J. Joyce and Carl-Johan H. Seger. Linking bdd-based symbolic evaluation to interactive theorem-proving. In *Proceedings of the 30th international on Design automation conference*, pages 469–474. ACM Press, 1993.

[JSS00] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: The Alloy Constraint Analyzer. In *In proceedings of International Conference of Software Engineering*, June 2000.

[KG02] Shmuel Katz and Orna Grumberg. A framework for translating models and specifications. In *Proceedings of Third International Conference of Integrated Formal Methods IFM*, Turku, Finland, 2002.

[Kli96] Rob Kling. Systems safety, normal accidents, and social vulnerability. pages 746–763, 1996.

[KM02] Alexander Knapp and Stephan Merz. Model checking and code generation for uml state machines and collaborations. In G. Schellhorn and W. Reif, editors, *FM-TOOLS 2002: 5th Workshop on Tools for System Design and Verification*, Report 2002-11, Reisensburg, Germany, July 2002. Institut für Informatik, Universität Augsburg.

[KV98] Orna Kupferman and Moshe Y. Vardi. Modular model checking. *Lecture Notes in Computer Science*, 1536:381–401, 1998.

[Kwo00] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In *Proceedings of UML 2000*, York, 2000.

[LA99] K. Lano and K. Androutsopoulos. Reactive system refinement of distributed systems in B. In *Proceedings of 1st International Conference on Integrated Formal Methods*, York, 1999. Springer-Verlag.

[LAC00] K. Lano, K. Androutsopoulos, and D. Clark. Structuring and design of reactive systems using RSDS and B. In *Proceedings of FASE, ETAPS 2000*, Berlin, Germany, 2000.

[LAC03] K. Lano, K. Androutsopoulos, and D. Clark. Formal specification and verification of railway systems using UML. In *FORMS Symposium on Formal Methods for Railway Operation and Control Systems*, 15-16 May, Budapest, Hungary, 2003.

[LAK00] K. Lano, K. Androutsopoulos, and P. Kan. Structuring reactive systems in B AMN. In *Third IEEE International Conference on Formal Methods (ICFEM)*, pages 25–34, York, UK, September 2000.

[Lan05] Kevin Lano. *Design for Change: Advanced System Design with Java, UML and MDA*. Elsevier, March 2005.

[LB03] Michael Leuschel and Michael Butler. ProB: A model-checker for B. In *In Proceedings FME*, Pisa, Italy, September 2003.

[LBA99] K. Lano, J. Bicarregui, and A.Sanchez. Invariant-based synthesis and composition of control algorithms using B. In *FM'99 workshop on The B-Method at FM'99 World Congress On Formal Methods In The Development Of Computing Systems*, Toulouse, France, September, 1999.

[LC99] G. Lüttgen and V. Carreño. Analyzing mode confusion via model checking, 1999.

[LCA01] K. Lano, D. Clark, and K. Androutsopoulos. Semantic foundations of RSDS. Technical report, Kings College, 2001.

[LCA02a] K. Lano, D. Clark, and K. Androutsopoulos. Formalising inter-model consistency of the UML. In *UML 2002, Workshop on Consistency Problems in UML-Based Software Development*, Dresden, Germany, 2002.

[LCA02b] K. Lano, D. Clark, and K. Androutsopoulos. From implicit specifications to explicit designs in reactive system development. In *The Third International Conference, Integrated Formal Methods (IFM) 2002*, Turku, Finland, 2002.

[LCA02c] K. Lano, D. Clark, and K. Androutsopoulos. RSDS: A subset of UML with precise semantics. In *The fourth workshop on rigorous object-oriented methods*, King's College London,UK, 2002.

[LCA02d] K. Lano, D. Clark, and K. Androutsopoulos. Safety and security analysis of object-oriented models. In *SAFECOMP 2002*, pages 82–93, Italy, 2002.

[LCA$^+$02e] K. Lano, D. Clark, K. Androutsopoulos, P. Kan, and A. Sanchez. Formal synthesis of PLC-based control systems. In *Internal report, Department of Computer Science, King's College London*, 2002.

[LCA04] K. Lano, D. Clark, and K. Androutsopoulos. UML to B: Formal verification of object-oriented models. In *IFM*, Kent, England, April 2004.

[LCAK00] K. Lano, D. Clark, K. Androutsopoulos, and P. Kan. Invariant-based synthesis of fault-tolerant systems. In Mathai Joseph, editor, *"Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000)*, pages 46–57, Pune, India, 2000. Springer-Verlag.

[Lec02] Thierry Lecomte. Event driven B: methodology, language, tool support and experiments. In *Workshop on Refinement of Critical Systems*, January 2002.

[Led91] Y. Ledru. Developing reactive systems in a vdm framework. In *IWSSD '91: Proceedings of the 6th international workshop on Software specification and design*, pages 130–139. IEEE Computer Society Press, 1991.

[Lev97] N. Leveson. Analyzing software specifications for mode confusion potential, 1997.

[LFA02] Kevin Lano, Jose Luiz Fiadeiro, and Luis Filipe De Andrade. *Software Design Using Java 2*. Palgrave Macmillan; ISBN: 1403902305, 2002.

[LFK02] Harry Li, Kathi Fisler, and Shriram Krishnamurthi. The influence of software module systems on modular verification. In *9th International SPIN Workshop on Model Checking of Software*, April 2002.

[LG98] Karen Laster and Orna Grumberg. Modular model checking of software. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 20–35. Springer-Verlag, 1998.

[LH96] K. Lano and H. Haughton. *Specification in B: An introduction using the B Toolkit.* Imperial College Press, 1996.

[LH02] Karsten Loer and Michael D. Harrison. Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In *Proceedings of 17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 223–226, Edinburgh, UK, 2002.

[LHHR94] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *Software Engineering*, 20(9):684–707, 1994.

[LHR99] Nancy G. Leveson, Mats Per Erik Heimdahl, and Jon Damon Reese. Designing specification languages for process control systems: Lessons learned and steps to the future. In *ESEC / SIGSOFT FSE*, pages 127–145, 1999.

[LK98] K. Lano and P. Kan. Reactive system development in B. In *1st YUFORIC Workshop*, Brisbane, Australia, 1998.

[LMM99a] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[LMM99b] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999.* Kluwer, 1999.

[Lot96] A. Lotzbeyer. Task description of a fault-tolerant production cell, 1996.

[LP99] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. Technical Report TUCS-TR-272, Abo Akademi University, 18, 1999.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LS95] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148:303–324, 1995.

[LS97] F.William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: A first introduction to categories.* Cambridge University Press, 1997.

[Mar91] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, oct 1991.

[MB02] I. Majzik and B. Beny. Verification of UML statechart models of embedded systems. In *In B. Straube, E.J. Marinissen, Z. Kotasek, O. Novak, J. Hlavicka, R. Ruzicka (editors): Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 70–77, April 2002.

[MC03] A. Miller and M. Calder. An application of abstraction and induction techniques to degenerating systems of processes. In *Proceedings of the International Workshop on Model-Checking for Dependable Software Intensive Systems (MCDSIS)*. IEEE Computer Society Press, 2003.

[McM92a] K. L. McMillan. *The SMV system. Carnegie Mellon University*, 1992.

[McM92b] Kenneth L. McMillan. *Symbolic Model Checking : An approach to the state space explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publ., 1993.

[McM95] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design: An International Journal*, 6(1):45–65, January 1995.

[McM98a] K. L. McMillan. *Getting Started with SMV: User's Manual. Cadence Berkeley Laboratories*, 1998.

[McM98b] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In Alan Hu, editor, *Computer Aided Verification (CAV98)*, Vancouver, Canada, June 1998.

[ME03] Madanlal Musuvathi and Dawson R. Engler. Some lessons from using static analysis and software model checking for bug finding. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[Mil80] R. Milner. A calculus on communicating systems. In *Lecture Notes in Computer Science*, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[ML02] R. Marcano and N. Levy. Transformation rules of OCL constraints into B formal expressions. In *Workshop on critical systems development with UML. 5th International Conference on the Unified Modeling Language.*, October 2002.

[MOSS99] Markus Muller-Olm, David Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In G. File and A. Cortesi, editors, *In Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, volume 1694, pages 331–354. Springer LNCS 1694, September 1999.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems:Specification.* Springer Verlag, 1992.

[MR01] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

[MS91] K.L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *In Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51, Tokyo, Japan, 1991.

[MS00] Eric Meyer and Thomas Santen. Behavioral conformance verification in an integrated approach using UML and B. In *In IFM 2000: 2nd International Workshop on Integrated Formal Methods*, pages 358–379, 2000.

[MtSg96] Zohar Manna and the STeP group. Step: The stanford temporal prover (educational release), user's manual. Technical Report STAN-CS-TR-95-1562, Stanford University, 1996. Technical report.

[MV86] P. Wolper M. Vardi. An automata-theoretic approach to automatic program verification. In *In Proc. of the IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[Nag] The Nagoya Crash. $http://www.flightsafety.org.au/articles/c0003.php$.

[NuS] The NuSMV User Manual. $http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html$.

[oD97] Ministry of Defence. Defence standard 00-55: Requirements for safety related software in defence equipment, 1997.

[Ove81] W. T. Overman. *Verification of concurrent systems: function and timing*. PhD thesis, University of California, 1981.

[PDH99] Corina S. Pasareanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN*, pages 168–183, 1999.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *In Proceedings of the 6th International Conference on Computer Aided Verification,*, volume LNCS 818. Springer-Verlag, 1994.

[Pep] Pep tool. $http://theoretica.informatik.uni-oldenburg.de/\ pep/$.

[Pet] Petri nets, by armin zimmermann. $http://pdv.cs.tu-berlin.de/\ azi/petri.html$.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computing Scientists*. The MIT Press, 1991.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 31– 2 1977. IEEE Computer Society Press.

[Pnu86] A. Pnueli. Applications of temporal logic in the specification and verification of reactive systems: a survey of current trends, lecture notes in computer science 224: Current trends in concurrency, 1986.

[PPSM03] M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti. Practical model checking of ltl with past. In *1st Int. Workshop on Automated Technology for Verification and Analysis*, National Taiwan University, December 2003.

[PR91] H. Plunnecke and W. Reisig. Bibliography of petri nets 1990. In G. Rozenberg, editor, *Advances in Petri Nets 1991, LNCS*, volume 524, page 317, Germany, 1991. Springer-Verlag.

[Pro] Prod 3.3.10. $http://www.tcs.hut.fi/Software/prod/$.

[PS91] A. Pneuli and M. Shalev. What is in a step: On the semantics of statecharts. In *Theoretical Aspects of Computer Software, International Conference TACS'91, volume 526 of Lecture Notes in Computer Science*, pages 244–264, Japan, September 1991. Springer-Verlag.

[PSS98] Amir Pnueli, M. Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, 1998.

[PT98] H. Peng and S. Tahar. A survey on compositional verification, 1998.

[QS82] J. Quelle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium proceedings, LNCS 137*, pages 337–351. Springer-Verlag, 1982.

[Ram00] S. Ramesh. Refinement and efficient verification of synchronous programs. In *In proc. IFAC Distributed Computing and Control Conference*. Pergamon Press, 2000.

[RG00] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.

[RGA$^+$96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[Rod] The rodin (rigorous open development environment for complex systems) project. $http://rodin.cs.ncl.ac.uk$.

[Ros98] Roscoe. Proving security protocols with model checkers by data independence techniques. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[Ros03] Bill Roscoe. Compiling statemate statecharts into csp and verifying them using fdr. In *Extended Abstract*, January 2003.

[RSG99] J.N. Reed, J.E. Sinclair, and F. Guigand. Deductive reasoning versus model checking: Two formal approaches for system development. In *Proceedings of 1st International Conference on Integrated Formal Methods*, York, 1999. Springer-Verlag.

[San96] A. Sanchez. *Formal Specification and Synthesis of Procedural Controllers for Process Systems (Lecture Notes in Control and Information Sciences, 212).* Springer Verlag; ISBN 3540760210, 1996.

[Sch01] Steve Schneider. *The B-method: an introduction.* Palgrave, 2001.

[Sch02a] P. Schnoebelen. The complexity of temporal logic model checking, 2002.

[SCH02b] Wuwei Shen, Kevin Compton, and James K. Huggins. A toolset for supporting UML static and dynamic model checking. In *In Proceedings, IEEE Computer Society, 26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment*, pages 147–152, Oxford, England, August 2002.

[Sha96] N. Shankar. PVS: combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, nov 1996. Springer-Verlag.

[Sha98] Natarajan Shankar. Lazy compositional verification. *Lecture Notes in Computer Science*, 1536:541–564, 1998.

[SKM01] Timm Schfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[Smi00] Graeme Smith. *The Object-Z Specification Language.* Advances in Formal Methods Series, Kluwer Academic Publishers, ISBN 0-7923-8684-1, 2000.

[Sno02] Colin Snook. Combining uml and b. In *In Proceedings of Forum on specification and design languages*, Marseille, France, 2002.

[SRI] SRI International Computer Science Laboratory. $http://pvs.csl.sri.com/$.

[SSE03] Claus Schrter, Stefan Schwoon, and Javier Esparza. The Model-Checking Kit. In *Applications and Theory of Petri Nets 2003,Lecture Notes in Computer Science*, volume 2679, pages 463–472. Springer, June 2003.

[Sta] I-Logix' Statemate MAGNUM. $http://www.ilogix.com/products/magnum/index.cfm$.

[Sto96] Neil Storey. *Safety-critical Computer Systems.* Addison-Wesley; ISBN: 0201427877, 1996.

[SW91] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. In *2nd international joint conference on Theory and practice of software development*, pages 161–177. Elsevier Science Publishers B. V., 1991.

[SZ01] E. Sekerinski and R. Zurob. iState: A statechart translator. In *UML 2001 - The Unified Modeling Language, Lecture Notes in Computer Science 2185,*, pages 376 – 390, Toronto, Canada, October 2001. Springer-Verlag.

[Tan01] Meyer C. Tanuan. Automated analysis of unified modeling language (UML) specifications. Master's thesis, Department of Computer Science, University of Waterloo, Ontaria, Canada, 2001.

[Tre02] Helen Treharne. Supplementing a UML development process with B. In *In proceedings of FME*, July 2002.

[UMLa] The unified modeling language (UML), Version 1.5, Specification provided by the OMG. $http://www.omg.org/technology/documents/formal/uml.htm$.

[UMLb] UML profile for schedulability, performance, and time), Version 1.0, Specification provided by the OMG, 2003. $http://http://www.omg.org/cgi-bin/doc?formal/2003-09-01$.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Workshop on Computer Aided Verification*, June 1990.

[Var98] Moshe Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In *Logic in Computer Science*, pages 394–405, 1998.

[VBW94] M. Y. Vardi, O. Bernholtz, and P. Wolper. An automata-theoretic approach to branching-time model checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 142–155, Standford, California, USA, 1994. Springer-Verlag.

[Ver] Verilog dot com. $http://www.verilog.com/$.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 1994.

[Wie03] R. J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate and the UML.* Morgan Kaufmann; ISBN 1-55860-755-2, 2003.

[Win01] Kirsten Winter. *Model Checking Abstract State Machines.* PhD thesis, Technischen Universität, Berlin, 2001.

[Wol95] P. Wolper. An introduction to model checking. In *In Proc. of the Software Quality Week (SQW'95)*, San Francisco, May 1995.

[WVF95] Jeannette M. Wing and Mandana Vaziri-Farahani. Model Checking Software Systems: A Case Study. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, 1995.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997.