Invariant-based Synthesis of Fault-tolerant Systems

K. Lano[1], David Clark[2], K. Androutsopoulos[1], and P. Kan[1]

[1] Department of Computer Science, King's College London, Strand, London WC2R 2LS
[2] Department of Computing, Imperial College, London SW7 2BZ

Abstract. Statecharts are a very widely used formalism for reactive system development, however there are problems in using them as a fully formal specification notation because of the conflicting variants of statechart semantics which exist. In this paper a modular subset of statechart notation is defined which has a simple semantics, and permits compositional development and verification. Techniques for decomposing specifications in this notation, design strategies for incorporating fault tolerance, and translation to the B formal language, are also described, and illustrated with extracts from a case study of a fault tolerant system.

1 Introduction

Finite state machines (FSMs) are highly recommended as a design method for safety-related systems of SIL levels 2 and above in the IEC 61508 standard [4]. Statecharts are based on state machines but add extra capabilities of modularization and expressiveness: grouping of states into superstates (OR composition) and grouping of OR states into concurrent collections (AND composition). However, in terms of their semantics, statecharts are much less transparent and more difficult to analyse than FSMs, a situation which is compounded by the conflicting variants of statechart semantics which exist. By taking advantage of the characteristic structures of reactive systems, a subset of classical statecharts may be selected as a modular specification notation for reactive systems, including real-time and fault-tolerant systems.

Section 2 defines SRS. Section 3 describes the overall development process and the translation from SRS to B AMN [9]. Section 4 illustrates the process on the case study.

2 Structured Reactive System (SRS) Statechart Notation

SRS is a modular subset of statecharts in which strong scoping on the parts of the statechart affected by a given event is imposed. An SRS statechart S is an AND composition of a set of *modules*: OR states which do not have AND states as direct or indirect substates. Modules are organised into a (client-server) hierarchy: a module transition may only send events to modules lower in the hierarchy (the *receivers* of the module). Transitions in a SRS module M have labels of form $t : e[G]/e_1 \frown \ldots \frown e_n$ where t is an (optional) transition name, e the name of the event triggering t, G is a logical *guard condition*, and the e_i are the events generated by t. G is optional and defaults to *true*. The e_i are also optional. The e_i are events of modules in $receivers(M)$, and only the states of modules in $receivers(M)$ can be referred to in G.

Appendix A gives the formal definition of SRS and module systems. Typically modules in a module system represent sensors, controllers, subcontrollers, and actuators.

Figure 1 shows the typical arrangement of modules for a reactive system (LHS), and the associated hierarchy of modules under the *receivers* relation (RHS). *Subcontroller* 1 and *Subcontroller* 2 are the receivers of *Controller*, etc, *Actuator* 3 has transitions for $g1$ and $g2$. Each module is a separate OR state within an overall AND state representing the complete system.

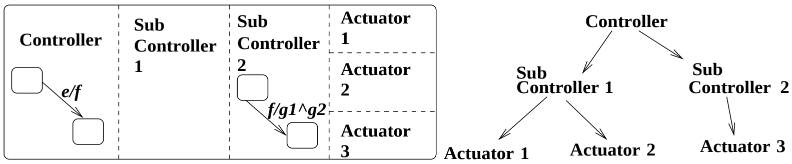

Fig. 1. Typical subsystem structure

3 Development Method

In our method, RSDS, reactive control systems are initially represented using data and control flow (DCFD) notation. Usually the behaviour of sensors, controllers and actuators will be specified using SRS modules in the notation of the appendix. Initially only sensors and actuators will have explicit state machine/module representations, and the specification of the mediating controller(s) will be implicit in a system invariant P in terms of sensor and actuator states. The form of P can be used to guide the decomposition of the controllers and to synthesise control actions, as described in [6]. Invariants of SRS modules are expressed in the following language. Let S consist of modules $M_1, \ldots, M_n$ where some of these modules describe the behaviour of sensors, others of controllers and others of actuators. For each module M a variable m_state is defined, of type $States_{flatten(M)}$, and formulae based on equalities $m\_state = s_1$ are used to express the current state of module M.

Except for the most trivial systems, it is necessary to modularise the specification of the control algorithm, in order to obtain analysable descriptions. There are several ways in which such a decomposition can be achieved:

1. *Hierarchical* composition of controllers: events e are dealt with first by an overseer controller S which handles certain interactions between components,

and e (or derived events) are then sent to subordinate controllers responsible for managing the individual behaviour of subcomponents.

This design can also be used to separate responsibility for dealing with certain aspects of a control problem (such as fault detection) from the calculation of control responses. For example in the steam boiler system of [1], detection and responses to inputs that indicate failure of components are handled in separate controllers to those which handle non-failed signals, using a “chain of responsibility” design pattern. A similar approach is used in the case study of Section 4.

2. *Horizontal* composition of controllers: events are copied to two separate control algorithms S_1 and S_2, which compute their reactions independently.
3. *Decomposition by control mode/phase:* A separate controller is specified for the control reactions to be carried out in each *mode* or *phase* of the system.

The first two are based on the *physical* decomposition of the actual system, whilst the third is based on *temporal decomposition*.

Invariants of controllers are decomposed into invariants of subcontrollers when the controller is structurally decomposed. A control algorithm is then synthesised for each of the controllers, based on their invariants [6].

Controller specifications expressed in a SRS module with invariants produce B machines whose set of operations correspond to the events which the controller responds to.

If the SRS description has a tree structured *receivers* relation, then the SRS structuring can be mapped directly to a corresponding structure in B: if module D is in $receivers(C)$, then the machine C' for C INCLUDES the machine D'.

4 Case Study: Fault Tolerant Production Cell

This system [8] is one of a series of industrial case studies based on automated manufacturing requirements. It involves the processing of metal pieces which enter the system on a feed belt, are conveyed by robot arms to a press, processed in the press, and then conveyed by the robot to a deposit belt. The layout of this version differs from the simple production cell [7] in two ways: (i) there is no crane to make the whole process cyclical; (ii) there are two presses rather than one. Otherwise, the movements of the robot arms etc. are identical.

The specification makes no requirement of alternating the presses. Should one fail, then the other takes on all the blanks coming into the system, and once the faulty press is working properly, it is brought back online again.

The system is expected to conform to the three state safety model shown in Figure 2, with transitions between the normal operating and recoverable failure states, and transitions from these to the unrecoverable failure state. In the following section we prove this conformance by constructing abstraction morphisms from the full state space of the system to the 3 state model.

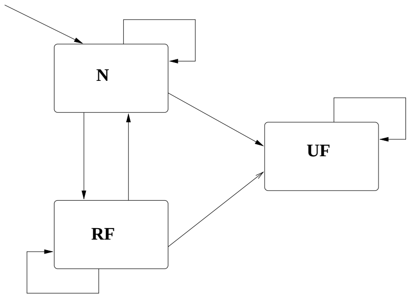

Fig. 2. Three State Failure Model for Production Cell

4.1 Statecharts and Abstraction

For each component of the cell, a series of abstractions and abstraction morphisms can be constructed. At the lowest level are the tuples of states of individual sensors and actuators within the component. For a press, for example, the basic state is a tuple $(ok, lows, mids, ups, bs, ms, pts, stm, pval)$ where ok is a boolean indicating if the press is failed or not, $lows$ is the lower position sensor, etc, bs is the blank in press sensor, stm is a boolean indicating that the arm is clear of the press, pts is the press timer state ($idle$, $active$ or $expired$), $pval$ is the timer value (pul, plm, pmu reflecting the deadlines for movement from the upper to the lower position, etc) and ms is the press motor state. For a single press there are therefore $4 * 3^7$ basic component states (8748). By constructing a suitable abstraction morphism to a simpler component model, we can create a clearer and more easily verified specification, both at the requirements and SRS level and in the B code.

In this case the high-level state ($pstate$) of a press is one of: (i) waiting to receive ($w2r$): $ok = true$, ($bs = no$ or $stm = false$), $mids = yes$, $lows = no$, $ups = no$, $ms = off$, $pts = idle$; (ii) processing: $ok = yes$, $bs = yes$, $lows = no$, $ms = up$, $pts = active$, $stm = true$, $pval = pmu$, $ups = no$; (iii) completing: $ok = yes$, $bs = yes$, $ms = down$, $lows = no$, $stm = true$, $pts = active$, $pval = pul$; (iv) waiting to deliver ($w2d$): $ok = yes$, ($bs = yes$ or $stm = false$), $lows = yes$, $mids = no$, $ups = no$, $ms = off$, $pts = idle$; (v): returning: $ok = yes$, $bs = no$, $mids = no$, $ups = no$, $ms = up$, $stm = true$, $pts = active$, $pval = plm$; (vi): failure: any state where $ok = false$. $ms = off$ will also be true in this state.

States where $bs = 0$ or $lows = 0$ or $mids = 0$ or $ups = 0$, where any two of $lows$, $mids$ and ups are yes, or where $stm = false$ with $ms \neq off$, or $pts = expired$ with $ms \neq off$ are excluded from (i) to (v): any event which leads to any of these conditions triggers a transition to *failure*. There are transitions to *failure* from any of the other states (which can be grouped into a superstate *normal*). Operational invariants of the system can then be simply defined using the abstract state $pstate$, for example

$$pstate = w2r \wedge \bigcirc stm = true \wedge bs = yes \;\Rightarrow\; \bigcirc pstate = processing$$

The module defining the control algorithm derived from these invariants is shown in Figure 3. States are expressed in the form $(lows, mids, ups, bs, ms, pts, stm, pval)$, superstates indicated by letters C, R, etc.

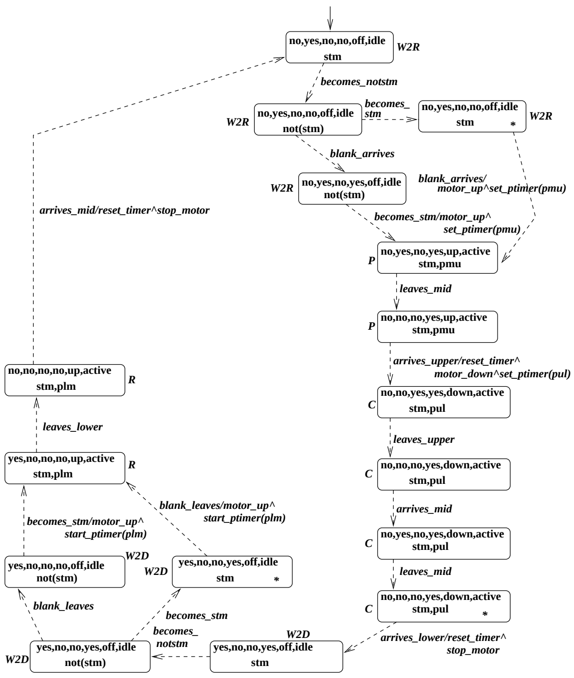

Fig. 3. Normal State Controller for Press

A further abstraction to a two state (normal/failure) model can then be made simply on the basis of the *ok* state value.

Each of the components of the production cell can likewise be abstracted to a basic safety/operational model consisting of two states, a normal and a failed state, and a transition from normal to failed and self-transitions on both states. The flattened product of these component models therefore has 256 states in the form of tuples: (feed belt, elevating rotating table, robot, robot arm 1, robot arm 2, press 1, press 2, deposit belt).

An abstraction morphism from this state machine to the three state global safety model for the cell is defined by:

1. All components *normal*: $\longmapsto$ N
2. One component failed, either press 1 or press 2 $\longmapsto$ RF
3. Both presses failed, or some other component failed $\longmapsto$ UF

Transitions are mapped correspondingly. This abstraction is an adequate refinement. A complete reachability analysis concerning failure modes can therefore be carried out on this model more effectively than on the fully flattened state machine model of the system (adequacy of $\sigma : C \rightarrow A$ implies that t is reachable from s in a concrete model C iff $\sigma(t)$ is reachable from $\sigma(s)$ in the abstraction A).

4.2 Controller Decomposition

Because of the relative independence of the control invariants for the separate sub-components of the cell, a hierarchical control structure can be used (Figure 4). Interaction between components is managed by supervisor controllers which send commands to the controllers of the individual components involved in a particular interaction. Detection of failures is handled by the FDC (Failure Detection Controller).

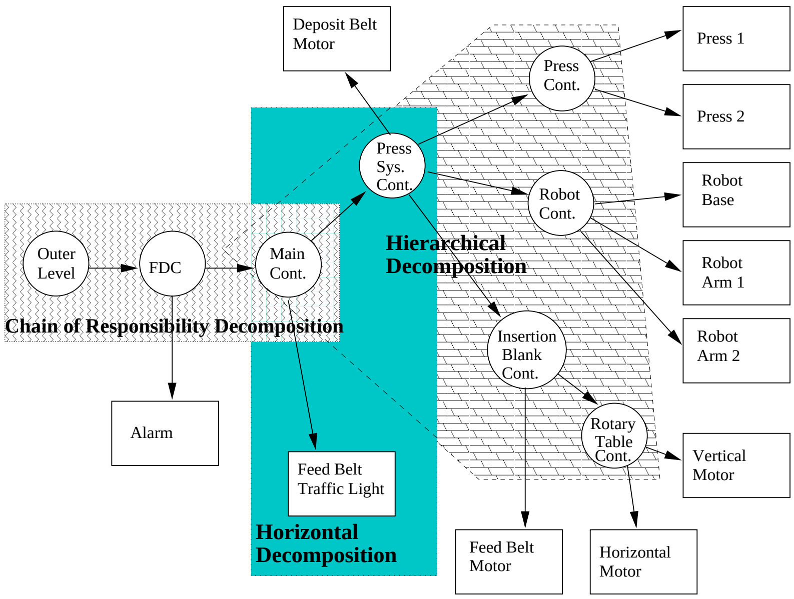

Fig. 4. Decomposition of Controllers

Failure detection is managed using the hierarchical *chain of responsibility* pattern described in [1] for fault-tolerance: at each cycle, the vector of sensor readings are checked for fault conditions, and failures notified to relevant components (if an unrecoverable failure occurs, a *shutdown* message is sent to the main controller, which then propagates it to individual controllers). Otherwise, the main controller is only sent events which do not represent failures.

4.3 Fault Detection

Checking for faults at the initial stage of sensor reading delivery to the system is essential if properties relating to individual polling cycles must be detected. For example, it could be regarded as a sign of sensor failure if both a *becomes_stm* (*stm* goes from *false* to *true*) event and a *blank_arrives* (*bs* goes from *no* to *yes*) event happen for a particular press within the same cycle (normally the arm delivering a blank would only start moving away once the blank had been confirmed to have arrived, or a timeout expired). This could not be detected at inner control levels as these know nothing of polling cycles.

Fault conditions are described by sets of invariants or control rules to be implemented by the FDC. Examples of these for a press are: (i) the arm moving into the vicinity of the press while the press is moving:

$$stm = false \wedge ms \neq off \;\Rightarrow\; ok = false$$

Or (ii) the blank falling off during a movement:

$$pstate = completing \wedge bs = yes \wedge \bigcirc bs = no \;\Rightarrow\; \bigcirc\, ok = false$$

For the feed belt we have rules: (i) If feed belt motor is on and S1 (blank sensor at start of belt) detects presence of blank, but state of S1 does not change in next cycle, then the motor has failed. (ii) If S2 (blank sensor at end of belt) changes from blank present to absent, and motor is off, then the motor has failed or blank has fallen off belt.

The system has no way of deducing the failure of the feed belt sensors if these both fail at the same time. Both the belt motor and traffic lights depend on the signal from S1. If S1 fails at *no* during an idle period of the belt, then the belt motor will not be triggered, and the traffic lights will remain green. In this situation, blanks can begin to pile up on top of each other at S1, which may eventually cause damage to the system. Only at this time will some sensor not included in the system specification as it stands detect the failure. For this reason, we assume that sensors give a 0 reading (distinct from *yes* and *no*) to indicate their failure.

Fault detection rules are implemented in the FDC, which is directly called by the outer controller. The outer controller has the form:

```
MACHINE OuterLevel
INCLUDES FailureDetectionController
VARIABLES
  fbok, fbtok, ...
INITIALISATION
  fbok := TRUE || fbtok := TRUE || ...
OPERATIONS
  cycle(curr_S1, curr_S2, ..., curr_S20, clockTick) =
    PRE curr_S1 : SENSOR_STATES ∧ ... ∧ curr_S20 : SENSOR_STATES ∧
        clockTick : TIME
    THEN
      /* call FDC : */
```

$fbok \longleftarrow check\_feedbelt\_sensors(curr\_S1, curr\_S2) \quad ||$
$fbtok \longleftarrow check\_feedbelt\_rTable(curr\_S1, curr\_S2, curr\_S3, curr\_S6) \quad ||$
$\vdots$
END
END

The checking of failure conditions is split into checks for individual components (eg. the first *check* operation) and checks concerning the interaction of several components (the second operation). The *FailureDetectionController* has the form:

```
MACHINE FailureDetectionController
INCLUDES MainController
⋮
OPERATIONS
  res ⟵ check_feedbelt_sensors(cS1, cS2) =
      PRE cS1, cS2 : SENSOR_STATES
      THEN
        IF /* rule (i) for feedbelt */
         (a1_fbelt_motor_switch = on ∧ s1_blankFB_start = yes ∧
          cS1 = yes) or
           /* rule (ii) for feedbelt */
         (a1_fbelt_motor_switch = off ∧ s1_blankFB_start = yes ∧
          cS1 = no) or ...
        THEN
          res := FALSE
        ELSE
          res := TRUE
        END
      END;

  ⋮
END
```

5 Conclusion

This paper has defined improvements to statechart notation for use in safety related reactive systems, and demonstrated the use of this notation and associated refinement rules on a case study of a fault-tolerant production cell control system. Tools have been developed to assist in the construction of DCFD and statechart models of a reactive system, and in the construction of abstraction mappings between refined and abstract statechart models [2]. The benefits of the approach include a systematic approach to reactive system development, using mainly graphical rather than explicitly formal notations, without sacrificing rigor.

The approach taken is an advance on previous work, since it incorporates a concept of refinement and safety into software tools for statecharts, which

Statemate and RSML do not. In addition, it provides a structured translation from statecharts into the B notation, instead of the translation to flat B modules given in [10].

A related approach to reactive system specification in B is that of [11]. Structuring is introduced as part of refinement in this approach, however, which may make it less simple to use for non-software engineers. In our approach the structure of specifications and implementations is identical as both are derived from the SRS specifications.

References

1. M. Ali, *B Specification of Steam Boiler*, MSc thesis, Dept. of Computing, Imperial College, 1998.
2. K. Androutsopoulos. *The Reactive System Design Tool*, ROOS Project report, Department of Computing, Imperial College, 1999.
3. I. Hayes, *A Survey of Data Refinement and Full Abstraction in VDM and Z*, Dept. of Computer Science, University of Queensland, 1991.
4. International Electrotechnical Commission, *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, 1999.
5. K. Lano, J. Bicarregui, and A. Evans. *Structured Axiomatic Semantics for UML Models*, ROOM 2000 Proceedings, to appear in Electronic Workshops in Computer Science, Springer-Verlag, 2000.
6. K. Lano, K. Androutsopoulos, D. Clark, *Structuring and Design of Reactive Systems using RSDS and B*, FASE 2000, to appear in LNCS, Springer-Verlag, 2000.
7. C. Lewerentz, T. Lindner (eds.), *Formal Development of Reactive Systems*, LNCS Vol. 891, Springer-Verlag, 1995.
8. A. Lötzbeyer, R Mühlfeld, *Task Description of a Flexible Production Cell with Real Time Properties*, FZI, Karlsruhe, 1996.
9. F. Mejia, *Formalising Existing Safety-Critical Software*, FMERail Workshop No. 2, London, UK, 1998. `http://www.ifad.dk/Projects/fmerail.htm`.
10. E. Sekerinski. Graphical Design of Reactive Systems. *2nd International Conference on B*, Lecture Notes in Computer Science, Springer Verlag, pages 182–197, 1998.
11. H. Treharne, S. Schneider, *Using a Process Algebra to Control B Operations*, IFM '99, Springer-Verlag, 1999.

A Formal Definition of SRS Statechart Notation

State machines Restricted statecharts are defined in terms of state machines. Formally, a *state machine* A consists of sets $States_A$ of states, $Events_A$ of events and $Trans_A$ of transitions. Each transition t of A has a source state $source_A(t) \in States_A$, a target state $target_A(t) \in States_A$, and an event $event_A(t) \in Events_A$ which denotes the event which may trigger the transition.

There is a designated initial state $init_A$. It is denoted as the target of a virtual transition arrow which has no source.

States will be classified as *safe* or *hazard* states (accidents are events which may occur from hazard states). Disjoint sets $Safe_A$, Haz_A denote the respective sets of safe and hazard states of A.

State machine abstraction morphisms and refinements Stepwise refinement of reactive systems involves the successive elaboration of controller responses and details of the equipment under control. A refinement C of a system A must be able to respond to all requests that A can, in a compatible way. This is formalised by the concept of an *abstraction morphism*.

A *state machine abstraction morphism* $\sigma : C \to A$ is a function from $States_C \cup Events_C \cup Trans_C$ to $States_A \cup Events_A \cup Trans_A$ which maps each state s of C to a state $\sigma(s)$ of A, each event α of C to an event $\sigma(\alpha)$ of A, and each transition t of C to a transition $\sigma(t)$ of A, such that:

1. If the event of t in C is α, then the event of $\sigma(t)$ in A is $\sigma(\alpha)$: $event_A(\sigma(t)) = \sigma(event_C(t))$.
2. $source_A(\sigma(t)) = \sigma(source_C(t))$ and $target_A(\sigma(t)) = \sigma(target_C(t))$.
3. $\sigma(init_C) = init_A$.

Such a morphism preserves the behaviour of a state machine in the sense that all transitions $t : s \to s'$ in C for a given event α must correspond to some abstract transition $\sigma(t) : \sigma(s) \to \sigma(s')$ in A for the same event. That is, no essentially new behaviour can be introduced for α.

C is a *refinement* of A via σ if σ is a state machine abstraction morphism which is an isomorphism on the set of events of C and A: no new events are introduced and no abstract events are deleted.

A refinement σ is *adequate* if every state s of A has some corresponding state of C:

$$\forall s : States_A \cdot \exists s' : States_C \cdot \sigma(s') = s$$

and if every event α which has an abstract transition from some s also has a corresponding concrete transition from each corresponding state:

$$\begin{array}{l} \forall s : States_A;\ t : Trans_A \cdot \\ \quad event_A(t) = \alpha \wedge source_A(t) = s \ \Rightarrow \\ \qquad (\forall s' : States_C \cdot \sigma(s') = s \ \Rightarrow \\ \qquad\quad \exists t' : Trans_C \cdot \sigma(event_C(t')) = \alpha \ \wedge \ \sigma(t') = t \ \wedge \ source_C(t') = s') \end{array}$$

This ensures that C can react to α in every situation that A can. By definition of refinement, the effect of this reaction is also consistent with that given in A. The adequacy condition is similar to the "preservation of precondition" requirement of Z refinement [3].

An abstraction morphism $\sigma : C \to A$ is *safety preserving* if $s \in Haz_C$ implies $\sigma(s) \in Haz_A$.

Such a σ is referred to as a *safety abstraction*. Likewise a safety refinement is defined as a refinement which additionally preserves safety.

OR and AND composition Statecharts are defined as for state machines, with the additions of the two constructions of nesting of statecharts within states (OR composition) and concurrent (AND) composition. In the SRS formalism, these constructs can be eliminated successively to reduce a statechart to a state machine (a statechart with only basic states) [5].

An OR state s of a statechart A has an enclosed statechart $smach_A(s)$ and otherwise has the same properties as a state machine 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 $Events_A$. The graphical notation for an OR state s is a state enclosing $smach_A(s)$.

An AND state s is a parallel (AND) composition $A \mid B$ of two OR states. The states of $A \mid B$ are effectively pairs (a, b) of states a of A and b of B. The graphical notation for $A \mid B$ consists of A and B separated by a dashed line. $States_{A|B} = States_A \cup States_B$ and similarly for $Trans_{A|B}$ and $Events_{A|B}$.

Modules and subsystems In SRS statecharts all systems are described in terms of *modules*: an OR state containing only basic and OR states. A system description S is specified by the AND composition $M_1 \mid \ldots \mid M_m$ of all the modules contained in it, $modules(S) = \{M_1, \ldots, M_m\}$. Such an S is termed a *module system*.

Each module M in S has a set $receivers_S(M)$ of modules in S, which are the only modules of S which it can test or send events to: transitions t of M may refer to the state of modules $M' \in receivers_S(M)$ via a guard condition $condition_M(t)$ built from logical combinations of formulae $in\ x$ or $not(in\ x)$ where x is a state of M', representing that M' is in state x or not, respectively. t may have a generation sequence $generations_M(t) : \mathrm{seq}(\bigcup_{M' \in receivers_S(M)} Events_{M'})$.

The collection of all generated events of a statechart M is $Gen_M = \bigcup_{t:Trans_M} ran(generations_M(t))$.

$Gen_M \cap Events_M = \varnothing$ for a module M in S, and $Gen_M \subseteq \bigcup_{M' \in receivers_S(M)} Events_{M'}$.

$receivers_S$ 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 $receivers_S^*[\{M\}]$ is termed the *subsystem* S' defined by M. M is then 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 [9].

The set of *external events* Ext_S of a module system S is the set $\bigcup_{M \in modules(S)} Events_M - \bigcup_{M \in modules(S)} Gen_M$. Normally these are the events of the sensor modules and the controller at the top of the controller hierarchy.

For module systems, the external events of the system must not change in a refinement. That is $Ext_S = Ext_{S'}$ where S' is the refinement of S.

Definition If s, s' are states of a statechart C, the notation $s \sqsubseteq_C s'$ denotes that $s = s'$ or s' is an OR state, $smach_C(s') = M$ and $s \in States_M$.

Definition For any state s of a statechart C, s_{init} denotes the initial state of s: $s_{init} = s$ if s is basic, $s_{init} = (s')_{init}$ if s is an OR state with initial state s', and $s_{init} = (s_{1\,init}, \ldots, s_{n\,init})$ if s is an AND state $s = s_1 \mid \ldots \mid s_n$. s_{init} is therefore always a basic state or tuple of basic states.

Definition If M contains only basic states, then $States_{flatten(M)} = States_M$. Otherwise $States_{flatten(M)} = \bigcup_{s:States_M} States^{M,s}$, where, if s is a state of a statechart M, then, if s is basic: $States^{M,s} = \{s\}$. If s is an OR state, $E = smach_M(s)$: $States^{M,s} = \bigcup_{s':States_E} States^{E,s'}$. If s is an AND state, $s = s1 \mid s2$: $States^{M,s} = States^{M,s1} \times States^{M,s2}$.

Definition For a tree-structured module system S, *flatten*(S), the flattened version of S, can be constructed by systematic removal of OR and AND states, with appropriate re-targeting of transitions [2, 5]. It is a state machine.

Definition A (SRS) statechart abstraction morphism σ from a concrete statechart C to an abstract statechart A maps states of C to those of A, events of C to events of A and transitions of C to those of A such that:

1. Source, target and trigger of transitions are preserved:

$$\begin{aligned}&source_A(\sigma(t)) = \sigma(source_C(t))\\&target_A(\sigma(t)) = \sigma(target_C(t))\\&event_A(\sigma(t)) = \sigma(event_C(t))\end{aligned}$$

2. If s is an OR state of C then $\sigma(s)$ is an OR state or basic state of A, and initial states are preserved:

$$\sigma(s_{init}) \;=\; (\sigma(s))_{init}$$

3. σ preserves $\sqsubseteq$: $\quad s \sqsubseteq_C s' \quad \Rightarrow \quad \sigma(s) \sqsubseteq_A \sigma(s')$. This means that states within particular modules M of C remain in the abstraction $\sigma(M)$ of M in A.
4. The module structure of C is preserved, ie: $receivers_A(\sigma(M)) \;=\; \sigma[receivers_C(M)]$.
5. Conditions and generations are preserved:

$$\begin{aligned}&condition_A(\sigma(t)) \;=\; \sigma(condition_C(t))\\&generations_A(\sigma(t)) \;\ll\; \sigma \circ generations_C(t)\end{aligned}$$

where $s_1 \ll s_2$ denotes that s_1 is a (possibly non-contiguous) subsequence of s_2.

Further conditions are needed in order that *flatten*(σ) can be defined:

1. σ is the union of separate morphisms $\sigma_i : MC_i \to MA_i$ between corresponding modules of C and A, $\sigma(MC_i) = MA_i$, with these morphisms being compatible on events:

$$\begin{aligned}&\alpha \in Events_{MC_i} \wedge \beta \in Events_{MC_j} \;\Rightarrow\; (\sigma_i(\alpha) = \sigma_j(\beta) \;\Rightarrow\; \alpha = \beta)\\&\alpha \in Events_{MC_i} \cap Events_{MC_j} \;\Rightarrow\; \sigma_i(\alpha) = \sigma_j(\alpha)\end{aligned}$$

where $i \neq j$.
2. Each of the σ_i are adequate, and every module of A must correspond to some module of C.
3. $x \models E$ in C (x satisfies E, where x is a state of C and E a transition condition) implies that $\sigma(x) \models \sigma(E)$, for each such condition.
This is ensured in particular for *positive* E, ie, for E containing no occurrences of $\neg$ or $\Rightarrow$.

Given these constraints, if S and S' are tree-structured module systems, connected by $\sigma : S' \to S$, then there is an abstraction morphism $\tau \;=\; \mathit{flatten}(\sigma) : \mathit{flatten}(S') \to \mathit{flatten}(S)$ derived from σ. This is proved by structural induction on S'.

Similarly there are results showing that τ is also adequate under these constraints, provided that $\sigma(s_1) \sqsubseteq_A \sigma(s_2)$ implies $s_1 \sqsubseteq_C s_2$ for states of C, the receivers structure of C and A are identical except for new, complete, leaf modules in C, and there are no generations on initial transitions of C or A. In addition it can be shown that if σ is safety-preserving, so is τ.