05 0(1.2

Structuring and Design of Reactive Systems using RSDS and B

K. Lano[1], K. Androutsopoulos[1], and D. Clark[2]

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

Abstract. With the advent of comprehensive safety standards for software intensive safety related systems, such as IEC 61508 and its specialisations for particular industry sectors (medical, machinery, process, etc), there is a need to establish combinations of techniques which can be used by industry to demonstrate conformance to these standards for particular developments. In this paper we describe one such combination of techniques, involving statecharts and B, which is aimed at reactive control system development.
We define strategies for controller decomposition which allow safety invariants to be distributed into subcontroller requirements, and define techniques for the automatic synthesis of controllers from invariants. A case study of a train control system is used to illustrate the ideas.

1 Introduction

A control algorithm for a discrete event system describes the reactions (control signals to actuators) issued in response to each input event (from sensors) which may be sent to the controller. Typically this algorithm is represented as a finite state machine (FSM) or as a statechart. In current practice, control algorithms are usually developed by hand, thus introducing possibilities for perhaps very expensive or life-threatening design faults.

We aim to improve this practice by providing techniques for the systematic derivation of executable controllers from requirements statements. These techniques integrate control engineering techniques such as *procedural controller synthesis* [12] and the B formal method and toolset [1].

Section 2 summarises the method used and the structuring approaches. Section 3 identifies the semantic problems of existing statechart based notations and defines our restricted version, Structured Reactive System Notation (SRS), of statecharts designed to eliminate these problems. Section 4 describes the synthesis process for controllers from invariants, and how controllers represented in our restricted statechart notation are translated into B modules in a structured fashion. Section 5 illustrates the process on the case study.

2 Development Method

We represent a reactive control system using standard DCFD notation, except that input event flows are indicated by dashed lines and output command flows by solid lines (see for example, Figure 1). This corresponds to the convention used on finite state machines. There may be a set of invariants (such as safety and operational behavior properties) associated with each process (representing a controller). Usually the behaviour of sensors, controllers and actuators will be

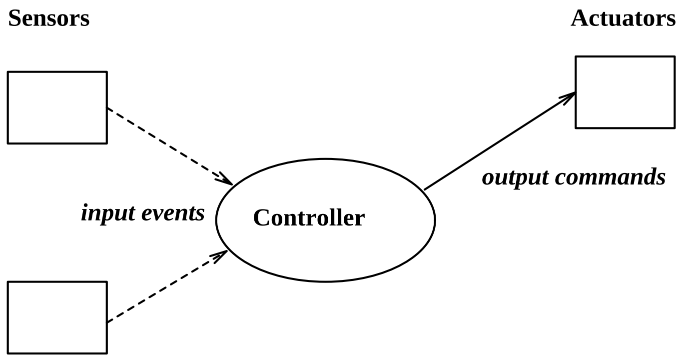

Fig. 1. Structure of Reactive Control Systems

specified by SRS modules in the statechart notation of Section 3. B specifications or implementations may be used interchangeably with SRS modules in these descriptions.

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 is appropriate if some control aspects can be managed at an aggregate level separately from control aspects which can be managed at an individual component level. It can also be used to separate responsibility for dealing with certain subsidiary aspects of a control problem (such as fault detection) from the calculation of control responses.
   Subordinate controllers S_1 and S_2 should control disjoint sets of actuators, or be independent on shared actuators in the sense that for any two command sequences a_1 and a_2 issued by S_1 and S_2 respectively to a shared actuator A, any permutation of $a_1 \frown a_2$ has the same state-transformation effect on A as $a_1 \frown a_2$. The timing of the responses of S_1 and S_2 relative to each other must also not be critical.

2. *Horizontal* composition of controllers: events are copied to two separate control algorithms S_1 and S_2, which compute their reactions independently. As with hierarchical composition, S_1 and S_2 should control disjoint sets of actuators, or be independent on shared actuators.
3. *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 [5].
4. *Annealing:* creating a separate subcontroller to encapsulate repeated control sequences as single operations (eg: opening a number of valves and pumps to open one flow path between vessels in a chemical plant).
5. *Composition of standard controllers:* recognising common control patterns (priority, and-combination, etc) and chaining together suitable versions of these standard controllers to achieve a more complex control function.

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

Various strategies may be used in selection of a suitable decomposition approach, as described in Section 5. The decomposition is usually performed on the DCFD and statechart descriptions of the system, instead of on the B representation. 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. This is usually done as part of a translation to B machines and implementations, as described in Section 4.

The B notation supports further verification techniques, such as internal consistency and refinement checks, and animation. If the statechart descriptions were translated directly to executable code, these additional verification steps could not be directly supported (a static analysis tool would need to be applied).

3 Restricted Statecharts

3.1 Problems with Statechart Semantics

The semantics of statecharts has proved to be fraught with problems and complications. The original operational semantics given by Harel and others in [3] was changed by Pnueli and Shalev in [11] since the semantics given in the first paper was highly operational and lacked global consistency.

Pnueli and Shalev's paper explored the extent to which the operational semantics, amended to provide global consistency, could be viewed as a declarative semantics – that is, a non-compositional functional semantics expressed in terms of fixed points. Correspondence with a declarative semantics proves to be fairly fragile as it can be destroyed by allowing disjunctions of events (and their absences, i.e. disjunctions of literals) as triggers for transitions. Nevertheless their paper set some sort of standard for elegance in the treatment of the semantics of statecharts.

The semantics in [11] was in turn criticised by Leveson and others in [10] as being counter-intuitive. Because generated events are considered in the same step as that in which they are generated and because transitions to add to the step are chosen one at a time, non-deterministically, it is possible to choose a transition triggered by a generated event rather than a conflicting transition triggered by an input event.

To overcome this problem [10] introduces an ordering in the way in which events are considered. In constructing a step as a series of micro steps they did not consider generated events in the micro step in which they were generated but only in the following one – once all the input events had been exhausted as triggers. This produced a more intuitive semantics but had the side effect of introducing potential cycles and hence steps whose calculation did not terminate.

Leveson and her colleagues have recently decided on eliminating generated events entirely from the syntax of RSML [9]. This not only removes the possibility of non-termination and makes the semantics of statecharts more intuitive but also simplifies the semantics considerably, albeit at the price of loss of expressiveness.

3.2 Structured Reactive System (SRS) Statechart Notation

To eliminate the above semantic problems of statecharts, we restrict statechart notation in the following ways:

1. Negations of events are not allowed as triggers to events, nor are logical combinations of events allowed.
2. Generated events are detected in subsequent steps to that in which they are generated, not in the same step.
3. A strict hierarchy of event generation and reception is enforced, so preventing cycles of sending and receiving.

Thus a transition in our restricted version has the form $t : e[G]/e_1 \frown \ldots \frown e_n$ where t is an (optional) transition label, e the name of the event triggering t (or e is a timeout specification $[min, max]$), 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. None of the e_i can lead directly or indirectly to a generation of e.

3.3 Modules and Subsystems

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

Each module M in a system description S has a set $receivers_S(M)$ of modules in S, which are the only modules of S to which it can send events:

$$generations_M(t) \in \mathrm{seq}(Events_{R_1} \cup \ldots \cup Events_{R_k})$$

for all $t : Trans_M$, where $receivers_S(M) = \{R_1, \ldots, R_k\}$, and

$$\alpha \in \operatorname{ran}(generations_M(t)) \;\wedge$$
$$\alpha \in Events_{M'} \;\Rightarrow\; M' \in receivers_S(M)$$

for any $M' \in modules(S)$.

$receivers_S$ is acyclic – $M \notin receivers_S^*(M)$ where $receivers_S^*$ denotes 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'.

Typically modules in a reactive system description represent sensors, controllers, subcontrollers, and actuators.

Figure 2 shows the typical arrangement of subsystem modules for a reactive system, and the associated hierarchy of modules under the *receivers* relation. *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 subsystem.

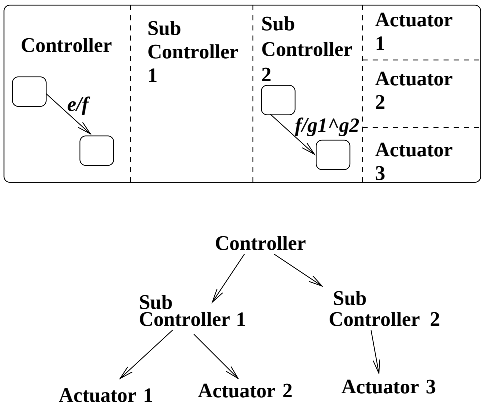

Fig. 2. Typical subsystem structure

4 Synthesis of Control Algorithms

The invariants for controller behaviour, both operational and safety, can be used to synthesise the required control algorithm, ie, the reaction to individual events.

We assume that a typical operational invariant has the form:

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

where *sstate* is some sensor state, G a guard involving other sensor or actuator states, and *astate* is an actuator state. This represents an obligation that the actuator must be on if an associated switch or trigger sensor is in state $s1$ and the guard is satisfied. (The consequent may also be a disjunction of such equalities, more generally).

Then the reaction to any event that sets *sstate* to $s1$ while G is true must set *astate* to be *on*. Also, any event which results in G becoming true whilst $sstate = s1$ must also have a reaction which sets $astate = on$.

In detail, the algorithm for synthesising an abstract B specification from the invariants is as follows:

- For each event e (that affects the state of a particular sensor, for example setting *sstate* to $s1$):
  1. Identify all invariants which may be invalidated by this state change – ie, invariants of the form

     $$\begin{array}{l} sstate = s1 \wedge G1 \;\Rightarrow\; astate = a1 \\ sstate = s1 \wedge G2 \;\Rightarrow\; astate = a2 \end{array}$$

     or any other invariants involving *sstate*.
  2. Identify actuator changes needed to maintain these invariants – gather together in a single conditional clause all cases of changes to a particular actuator:
     ```
     IF G1
     THEN astate := a1
     ELSE
       IF G2
       THEN astate := a2
       ...
     ```
     All possible cases should be defined in the invariants – for missing cases additional invariants will have to be provided by the developer. A simple completeness check is that for each sensor S, and reachable state x of S, there is some invariant containing $sstate = x$ in its antecedent.
     These then give rise to a hierarchically organised set of conditional clauses, with the most general conditions in the outer conditional tests and more specific subcases in the inner tests.
     Optionally, assignments to actuator states can be replaced by invocations of corresponding operations of a machine which encapsulates the actuator state. The controller then *INCLUDES* all machines representing actuators which it controls.
  3. Compose changes to different actuators by $||$.
  4. Restructure the operation definition if necessary to move all occurrences of $||$ inside conditional branches, so that no $||$ combinator occurs outside an *IF* statement.

Controller specifications expressed in an SRS module with invariants therefore produce B machines whose set of operations correspond to the events which the controller responds to, defined by the above synthesis procedure. The invariants translate directly to invariants of the B machine. The B machine needs to declare a variable *sstate* to record the current state of each sensor whose events are detected by the controller. The operation e above then needs the additional assignment $sstate := s1$. Alternatively sensor states can be encapsulated in separate machines, making it easier to share read access to these sensors in a number of controllers.

If actuators are defined in separate machines, these machines need to be included in the controller.

In implementations, a fixed order of control actions is determined, and expressed in B by the use of the ; construct in place of || in controller operation definitions.

5 Case Study: Train Control System

The aim of this system is to safely control a train to ensure that the train only moves when the doors are locked. Its sensors are:

- Door: $dstate \in \{locked, closed, opening, closing, open\}$
- Motion sensor: $motstate \in \{stationary, moving\}$
- Switch (to start train): $swstate \in \{on, off\}$
- Door button (to open/close doors): $dbstate \in \{on, off\}$

Its actuators are:

- Motor: $mstate \in \{on, off\}$
- Brake: $bstate \in \{on, off\}$
- Door.

The safety invariants are:

1. $motstate = moving \;\Rightarrow\; dstate \in \{closing, closed, locked\}$
2. $dstate \neq locked \;\Rightarrow\; mstate = off \wedge bstate = on$
3. $bstate = off \;\equiv\; mstate = on$ (the motor is on iff the brake is off).

The operational invariants are:

1. $swstate = on \wedge dstate = locked \;\Rightarrow\; mstate = on$
2. $dbstate = off \;\Rightarrow\; dstate \in \{closing, closed, locked\}$
3. $swstate = off \;\Rightarrow\; mstate = off$
4. $dbstate = on \wedge swstate = off \wedge motstate = stationary \;\Rightarrow\; dstate \in \{open, opening\}$
5. $swstate = on \;\Rightarrow\; dstate \in \{closing, closed, locked\}$ (the train switch has priority over the door button).

Figure 3 shows the overall context diagram of this control system.

5.1 Structuring Alternatives

We can decompose control requirements in a number of ways: temporal mode or phase decomposition, hierarchical decomposition, annealing, or by recognition of standard controllers. In the following sections we detail how these are applied in practice and illustrate them using the train control system.

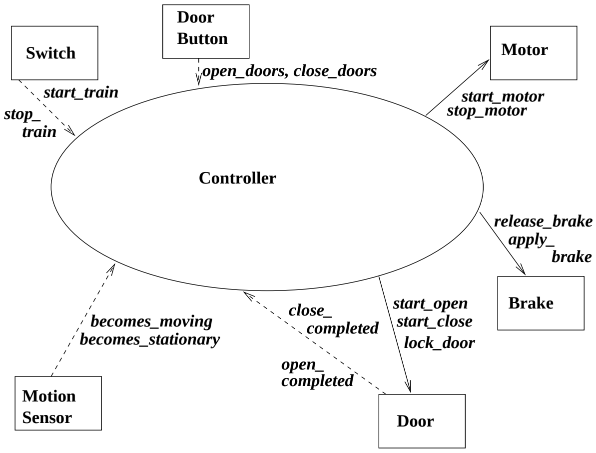

Fig. 3. DCFD of Train Control System

Temporal Mode/Phase Decomposition In order to decompose on the basis of temporal mode, we select a variable (or variables) representing sensor states, whose set of possible values are to be taken as the set of modes. In making this selection, it is useful to choose the variables which occur most frequently on the LHS of the safety and operational invariants: such variables will be involved in more control rules than others, and hence by fixing their values in particular modes, more control rules will be simplified than if less frequently occurring variables were chosen.

In the train control system, the variable *swstate* occurs most frequently on the LHS of axioms (4 occurrences), and is therefore the prime candidate for constructing modes. The two possible states *on* and *off* of this variable give rise to two modes *SwOn*, *SwOff*. The control axioms for these modes are derived from those for the overall controller by substituting in the fixed values of *swstate* in these two modes and simplifying. For *SwOn* the axioms therefore become:

- Safety:
  1. $dstate \neq locked \;\Rightarrow\; mstate = off \land bstate = on$
  2. $bstate = off \;\equiv\; mstate = on$

  Operational invariants:
  1. $dstate = locked \;\Rightarrow\; mstate = on$
  2. $dstate \in \{closing, closed, locked\}$

Notice that only a record of *dstate* needs to be kept in the controller for *SwOn* (in order to express these invariants), other sensor states need not be recorded and hence *SwOn* does not need operations for any of their events. Similarly for *SwOff*.

Operational axioms 1 and 5 of the train system define the actions the coordinator controller should take in switching from the *SwOff* to the *SwOn* phase controllers, and operational axioms 3 and 4 the actions to be taken in switching from *SwOn* to *SwOff*.

The structural decomposition resulting from this choice is shown in Figure 4. The controllers for *SwOn*, *SwOff* and the coordinator controller can now be derived from their respective sets of axioms. Because there are many shared

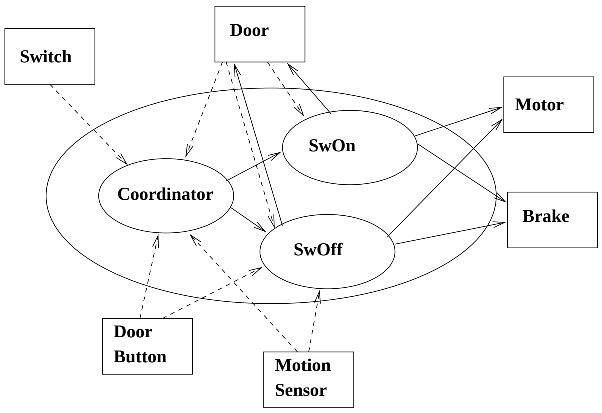

Fig. 4. Phase Decomposition of Train Controller

sensors and actuators between the two phase controllers, this decomposition is probably a bad choice for the train control system. In other systems, such as the milk plant of [7], where substantially different sets of equipment are used in the different phases of operation (eg: filling with milk, emptying, cleaning, etc), such a decomposition would be more effective.

A more sophisticated approach for phase decomposition is to identify an invariant of the form $P_1 \vee \ldots \vee P_n$ 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.

Hierarchical Decomposition The second structural decomposition approach is to introduce hierarchical composition by selecting subsets of the actuators of the system and isolating the control rules which apply only to those actuators. The aim is to identify disjoint groups $A_1, \ldots, A_n$ of actuators such that there are relatively few axioms in the original requirements relating the states of actuators in A_i to those in A_j for $i \neq j$. If there are *no* such axioms, then horizontal structuring can be used, otherwise a coordinator controller will be needed which invokes the controllers for A_i and A_j in such a way that the invariants which link their states are maintained.

In the case of the train control system, a suitable grouping of actuators is $A_1 = \{Motor, Brake\}$ and $A_2 = \{Door\}$. This is because only one safety invariant (number 2) and one operational invariant (number 1) link the states of these two groups.

As with temporal mode decomposition, we generate new sets of axioms which the subcontrollers for A_1 and A_2 must satisfy. These are obtained by selecting the axioms which refer only to the actuators in each subgroup. Axioms which refer to actuators in both A_1 and A_2 are not included in the subcontrollers but remain as obligations on an overseer controller.

The structure of this decomposition of the system is shown on the LHS of Figure 5, and the corresponding state machine module structure on the RHS.

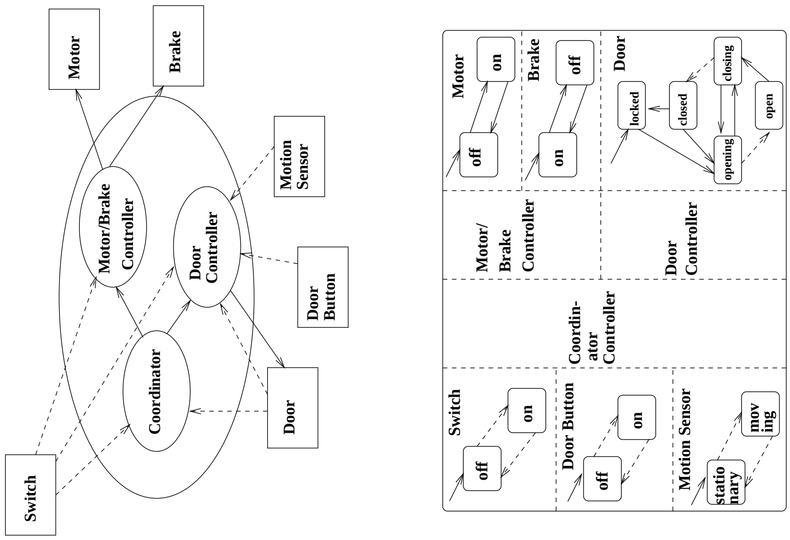

Fig. 5. DCFD and Module Structure of Hierarchical Decomposition

Annealing In this decomposition approach, we identify repeated groups of actuator commands issued by the controller, and package these groups of commands into a single module, so that the controller needs to only issue single commands to this module instead of sets of specific commands to particular actuators. This decomposition assists in making the controller more maintainable (eg., if the precise set of actuators changes in different versions or upgrades of the system).

In the train controller the natural candidates for such subsequences are

$$\begin{aligned} \rho_1 &= \mathit{start\_motor} \frown \mathit{release\_brake} \\ \rho_2 &= \mathit{apply\_brake} \frown \mathit{stop\_motor} \end{aligned}$$

as generations of a monolithic controller for the system. These can be 'annealed' into the generations of a module M' with a single state s and transitions for events *start* and *stop* with the labels

$start/\rho_1$
$stop/\rho_2$

Since M' has no invariants, and no invariants are removed from the main controller, this step does not improve verifiability, unlike the phase and hierarchical decompositions. For this system hierarchical decomposition already achieves all the modularity provided by annealing, so is the preferable approach.

Recognition of Standard Controllers Certain simple control mechanisms occur quite frequently, so it is appropriate to keep a list of ready-built controllers for these cases, which can then be adapted to particular sensors and actuators by renaming, and chained together to define more elaborate control functions. In [8] we describe two common patterns, an 'And' controller which takes inputs from two switches, produces a *goon* command only if both switches have been set *on*, and produces a *gooff* command when either one or the other is set *off* (when the other is *on*). A similar controller is the *priority* controller, which switches on actuator A whenever switch A is on, and switches it off whenever switch A is off, and similarly for actuator B and switch B. However actuator B cannot be on if switch A is on, so A has priority over B.

By chaining together copies of these controllers, it is possible to achieve complex control functions. To select appropriate standard controllers for a new control problem, the invariant of this problem should be examined for patterns similar to those of (for example) the And and Priority controllers.

In the case of the train control system, the priority controller pattern can be recognised in the interaction between the *swstate* (playing the role of *sastate*) and *dbstate* (playing the role of *sbstate*), where an indication $mstate'$ that the motor should be on/off plays the role of *aastate*, and an indication $dstate'$ that the door should open/close plays the role of *abstate*. The following axioms result from this adaption of *PriorityController*:

1. $swstate = off \;\Rightarrow\; mstate' = off$ (cf. operational axiom 3)
2. $dbstate = off \;\Rightarrow\; dstate' = off$ (cf. operational axiom 2)
3. $swstate = on \;\Rightarrow\; mstate' = on$ (cf. operational axiom 1)
4. $dbstate = on \wedge swstate = off \;\Rightarrow\; dstate' = on$ (cf. operational axiom 4)
5. $swstate = on \;\Rightarrow\; dstate' = off$ (cf. operational axiom 5)

Further patterns can be applied until the axioms of the train control system are recovered. For example, the motor can only be on if $mstate' = on$ and $dstate = locked$.

5.2 Translation to B

Phase Decomposition The translation of control actions to B operations follows Section 4, with the addition that several phase controllers may include the same sensor or actuator components. For example, both *SwOn* and *SwOff* include *Motor*. Such a structure is not allowed in B, because invariants of the subordinate machines may be violated without any of their operations being executed: if machines B and C both include D, then C may change the state of D in such a way that B's invariant is violated despite the fact that all operations of B maintain the invariant. In an object-oriented language such as VDM^{++}, no such problem would arise, as the controller C would be dynamically swapped in for the controller B, and only C's invariant would contribute to the system invariant while the phase that it represents is in progress.

A way to 'trick' the B Toolkit into accepting such a structure is described in [6], however this naturally limits the benefits of the formal method in providing verification checks on the controllers.

Here the coordinator has the specification:

```
MACHINE Coordinator
SEES TrainTypes
INCLUDES SwOn, SwOff
VARIABLES swstate
INVARIANT swstate ∈ State ∧
    (swstate = on ∧ dstate = locked  ⇒  mstate = on) ∧
    (swstate = off  ⇒  mstate = off) ∧
    (dbstate = on ∧ swstate = off ∧ motstate = stationary  ⇒
                                         dstate ∈ { open, opening }) ∧
    (swstate = on  ⇒  dstate ∈ { closing, closed, locked })
INITIALISATION swstate := off
OPERATIONS
    start_train =
          PRE swstate = off
          THEN
              swstate := on ||
              init_swon
          END;

   close_completed =
          PRE dstate = closing
          THEN
               IF swstate = on
               THEN swon_close_completed
               ELSE swoff_close_completed
               END
          END;

    ⋮

END
```

Each operation e of *Coordinator* that corresponds to an event e of sensor S must forward e in renamed form P_e to any of the currently active phase controllers

P which refer to the state of S (ie., to any P which is a target of an arrow from S in the DCFD of the system).

Hierarchical Decomposition The translation of control actions to B operations in this case follows Section 4, with the addition that there may be duplicate records of the same sensor state in different B machines representing controllers. For example, in the train control system, both *MBController* and the coordinator controller require a record of *swstate* in order to express their invariants. Such duplication is handled by renaming the variables in the subordinate controllers, and then asserting the equality of the different sensor state representations in the coordinator. Each subcontroller which needs read-only access to a sensor or actuator state to determine the validity of an action, must *SEE* the machine which encapsulates that state.

In this case the coordinator is:

```
MACHINE Coordinator
SEES TrainTypes, Bool_TYPE
INCLUDES MBController, DoorController
VARIABLES swstate
INVARIANT swstate ∈ State ∧
    swstate = mb_swstate ∧ swstate = d_swstate ∧
    (dstate ≠ locked ⇒ mstate = off ∧ bstate = on) ∧
    (swstate = on ∧ dstate = locked ⇒ mstate = on)
INITIALISATION
    swstate := off
OPERATIONS
    start_train =
        PRE swstate = off
        THEN
            swstate := on ||
            mb_start_train ||
            d_start_train
        END;

    stop_train =
        PRE swstate = on
        THEN
            swstate := off ||
            mb_stop_train ||
            d_stop_train
        END;

    open_door =
        PRE dbstate = off
        THEN
            d_open_door ||
            mb_open_door
        END;
```

⋮

END

Events are forwarded by this controller to all subcontrollers which need to respond to it, ie, those subcontrollers which refer to the state of the sensor which produced the event.

Annealing An extra machine is defined to act as the intermediary between the source of the annealed command sequences and the actuators it controls, otherwise the translation to B is as in Section 4.

Chaining of Standard Controllers In the translation to B, each copy of a standard controller produces a B machine (for example, a renamed version of the Priority controller of Section 5.1). In the hierarchical control structure there may be controllers which take inputs from sensors directly, rather than from hierarchically superior controllers. Whenever this is the case, we must promote such operations from the lower to the higher controller, in order to maintain the strict tree structure which B requires.

6 Tool Support

A suite of tools (collectively known as the RSDS tool [2]) have been developed in Visual C++ to support the specification of reactive systems using DCFD and statecharts and their translation into B specifications and implementations.

The RSDS tool currently provides the following features: (i) Supporting the visualisation of a correct DCFD for the entire reactive system and recording the operational, safety and liveness properties for its controllers. (ii) Checking on the existence or correctness of abstraction mappings between two statecharts, including construction of such mappings where they exist. (iii) Transformations (flattening) of statecharts with nesting, conditions, event generation and AND composition into state machines. (iv) Translating a statechart into B source for the purpose of invariant checking and animation/testing and code generation.

Extensions to the RSDS tool are under development allowing for: (i) Visual decomposition of the main controller in the DCFD using a suitable decomposition strategy. (ii) Checking the validity of the operational, safety and liveness properties of the statecharts.

The tool is also being ported to Java.

References

1. J-R Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

2. K. Androutsopoulos. *The Reactive System Design Tool*, ROOS Project report, Department of Computing, Imperial College, 1999.
3. D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the first IEEE Symposium on Logic in Computer Science*, pages 54-64, 1986.
4. International Electrotechnical Commission, *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, 1999.
5. International Society for Measurement and Control. *Batch Control Models and Terminology*, ISA-S88.01-1995, 1995.
6. P. Kan, K. Lano, *Reactive System Development in B*, 1st YUFORIC Workshop, Brisbane, Australia, 1998.
7. K. Lano, D. Clark. *Demonstrating Preservation of Safety Properties in Reactive Control System Development*, 4th Australian Workshop on Industrial Experience with Safety Critical Systems and Software, Canberra, ACT, November 1999.
8. K. Lano, J. Bicarregui, A. Sanchez, *Invariant-based Synthesis and Composition of Control Algorithms using B*, B User Group Meeting, Formal Methods '99.
9. N. G. Leveson. Designing a Requirements Specification Language for Reactive Systems. Invited talk, Z User Meeting, 1998, Springer Verlag 1998.
10. N. G. Leveson, Mats P.E. Heimdahl, Holly Hildreeth, and Jon D. Reese. Requirements specification for process-control systems. In *IEEE Transactions on Software Engineering*, volume 20, no. 9, pp. 684-707. 1995.
11. A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theoretical Aspects of Computing Software*, Lecture Notes in Computer Science, Volume 526, Springer-Verlag, Berlin, 1991, pp. 244-264.
12. A. Sanchez. *Formal Specification and Synthesis of Procedural Controllers for Process Systems*. Springer-Verlag. Lecture Notes in Control and Information Sciences, vol. 212. 1996.