Structuring Reactive Systems in B AMN

K. Lano K. Androutsopoulos P. Kan
Department of Computer Science, King's College London, Strand, London WC2R 2LS

Abstract

B has been widely used for high-integrity system development, for example in the railway industry. However there are few published guidelines on how to structure B specifications for particular types of system, such as reactive control systems. In this paper we describe a method to support graphical design of systems using B AMN, and develop guidelines for expressing the structuring requirements of reactive systems in B.

1 Introduction

Reactive systems can usually be characterised as transformations from sequences or sets of input signals into sequences of output signals. Their function is to respond to input signals from sensors by sending commands to actuator devices, in such a way that required invariants of safety and operational behaviour remain true for the controlled system.

A development process for reactive systems should be repeatable in the sense that given any sets S of sensor components, A of actuators, and I of operational and safety invariants, the procedure can be applied to provide a set of control algorithms and a design structure connecting these algorithms to each other and to the sensors S and actuators A, in such a way that I is satisfied.

In our method, RSDS, reactive control systems are partitioned in such a way that the development procedure can be applied to the subparts of the system in the same way as it is applied to the entire system. That is, a subpart can be regarded as a self-contained reactive system whose 'sensors' include both sensors of the original system and 'virtual sensors' which serve to transmit necessary information from other parts of the original system. Its actuators in turn will include both actuators of the original system, and outputs to other subparts of this system. It will also have its own safety and operational properties, derived from those of the original system.

The partitioning strategy chosen is described in Section 2, which defines our graphical notation, SRS statecharts, for reactive systems, and in Section 3, which describes the overall development process for our approach, RSDS. Section 4 describes the synthesis process for deriving controllers from invariants. Section 5 identifies how structuring in B can be used to implement reactive system structuring requirements. Section 6 illustrates the development procedure on a case study of a fault-tolerant production cell. Section 7 addresses the issue of event ordering by the outer level of a control system.

2 Structured Reactive System (SRS) Statechart Notation

SRS [13] is designed to be a modular subset of statechart notation, compatible with B AMN. A transition in an SRS statechart has a label 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 a logical combination of tests *in* s where s is a state of the statechart, 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. A formal definition of SRS is given in [13].

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.

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.

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:

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

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 [7].

OR and AND composition SRS 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 ('flattened') to reduce a statechart S to a state machine *flatten*(S). Abstraction and refinement morphisms are defined for SRS statecharts in such a way that they produce state machine abstraction and refinement morphisms on the flattened state machines [12]. This enables us to avoid explicit flattening and instead work with structured statecharts and morphisms defined between these.

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 to which it can send events.

$receivers_S$ is acyclic: $M \notin receivers_S^*[\{M\}]$ where $receivers_S^*$ is the transitive closure of $receivers_S$ considered as a relation.

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 [16].

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

Figure 1 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.

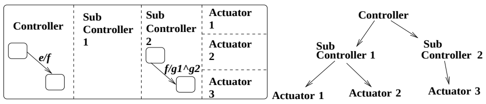

Figure 1. Typical subsystem structure

3 Development Method

In the RSDS (Reactive System Design Support) method reactive control systems are represented using DCFD notation (eg: Figure 2).

The behaviour of sensors, controllers and actuators is specified using SRS modules in the notation of Section 2. Initially only sensors and actuators have ex-

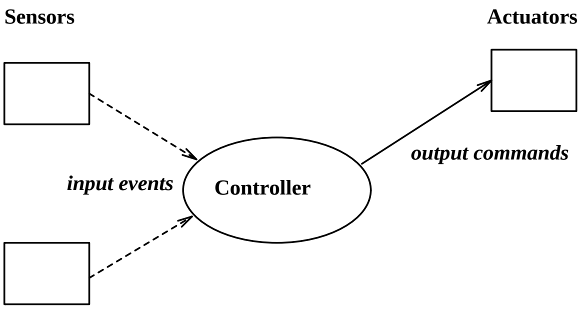

Figure 2. Structure of Reactive Control Systems

plicit state machine/module representations, and the specification of the mediating controller(s) is 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 [12].

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 $States_{flatten(M)}$ (ie, the basic states of 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.

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:* A separate controller is specified for the control reactions to be carried out in each *mode* or *phase* of the system [9].

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 [12], as follows:

1. *Hierarchical composition*: If this is based on splitting the actuators into disjoint sets $A_1, \ldots, A_n$ which are controlled by separate subcontrollers C_1, …, C_n of a supervisor controller C, then the invariants which only refer to actuator states from A_i are placed in C_i, invariants which link actuators from different A_i sets are placed in C.

2. *Horizontal composition*: Same as hierarchical, except that all invariants will be placed in one or other subcontroller, with none in the coordinator C.

3. *Phase decomposition*: The invariant P of C will be split up into disjoint cases $P_1, \ldots, P_n$ such that

   $$P \;\equiv\; P_1 \vee \ldots \vee P_n$$

   Each of the P_i becomes the invariant of a subcontroller C_i for that phase. In the case that the phases are distinguished by the value of a variable *sstate* for a particular sensor, ie,

   $$P_i \;\equiv\; (sstate = v_i \wedge Q_i)$$

   for each $i : 1 \,..\, n$, then P is equivalent to

   $$\begin{array}{l}(sstate = v_1 \;\Rightarrow\; Q_1) \;\wedge \ldots \wedge \\ (sstate = v_n \;\Rightarrow\; Q_n)\end{array}$$

   (assuming that $v_i \neq v_j$ if $i \neq j$ and these are the only possible values of *sstate*). The invariant of C_i can then be simplified to Q_i.

In each case the original control system design and synthesis task is reduced to a set of simpler synthesis and design tasks.

When a controller cannot be decomposed further, the synthesis process of the following section is applied to derive an SRS module or B machine to implement its behaviour.

4 Controller Synthesis

The interactive synthesis procedure supported by the RSDS tool prompts the designer to select suitable invariant clauses to be used in defining and checking proposed reactions to each event that may be input to the system [12].

The use of abstractions of subcontroller states and abstraction morphisms to these abstractions can help reduce the state space that needs to be considered during synthesis. E.g. for the press controller of Section 6 there are only 7 high level states, instead of approximately 9000 possible concrete states.

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' for D. However in cases such as a temporal decomposition of controllers, typically some modules (representing actuators used in more than one phase) will be in the *receivers* sets of two different controllers, a structure which is disallowed in B because of the additional complexity of proof needed.

4.1 Translation from SRS to B AMN

For a state machine M, the corresponding B abstract machine is $\mathcal{B}(M)$ defined as:

```
MACHINE M
INCLUDES N_1, ..., N_u
VARIABLES m_state
INVARIANT m_state : State_M ∧ Inv_M
INITIALISATION m_state := init_M
OPERATIONS
   α = PRE m_state : { sr_1, ..., sr_n }
         THEN
           IF m_state = sr_1 ∧ G_1
           THEN m_state := tr_1 || act
           ELSE ...
           END
         END;

      ⋮
END
```

where Inv_M is the conjunction of operational and safety invariants attached to M. An operation of the above form is included for each event $\alpha \in Events_M$, where

$$\{t_1, \ldots, t_n\} = \{t : Trans_M \mid event_M(t) = \alpha\}$$

and t_i has source sr_i, guard G_i, generations $\gamma_1 \frown \ldots \frown \gamma_m$ and target tr_i. The N_i are the machines corresponding to the elements of $receivers(M)$. act is $\gamma_1 \parallel \ldots \parallel \gamma_m$.

The implementation of this machine is generated along with the specification by the RSDS tool:

```
IMPLEMENTATION M_1
REFINES M
IMPORTS m_Vvar(State_M), N_1, ..., N_u
INVARIANT m_state = m_Vvar
INITIALISATION m_STO_VAR(init_M)
OPERATIONS
   α =
      VAR ms
      IN ms ⟵ m_VAL_VAR;
        IF ms = sr_1 ∧ G_1
        THEN m_STO_VAR(tr_1); act'
        ELSE ...
        END
      END;

   ⋮
END
```

where $act' = \gamma_1; \ldots; \gamma_m$.

The following theorem shows that refinement of control algorithms can either be carried out within the SRS notation or within B AMN, depending on the preferences of the developer or the requirements of standards.

Theorem If M' and M are statemachines, and M' is an adequate refinement of M via σ, then the translation of M' to a B machine $\mathcal{B}(M')$ refines the translation $\mathcal{B}(M)$ of M.

5 Structuring Requirements for Reactive Systems

Several kinds of structuring are typically required for the specification and design of a reactive system: command-recipient structures, with sharing (for example where two controllers both send commands to the same actuator) or without, and aggregation or collection structures (eg, where there are several instances of a particular controller or actuator in the system, with isomorphic structure). Additionally certain forms of reflexive or cyclic module dependency structures are needed, in cases where components combine the roles of a sensor and an actuator. We will identify how these structures can be expressed in B in the following sections.

5.1 Shared Command-Recipient Structures

Although the purely hierarchical structure of tree-structured SRS specifications can be mapped directly into a corresponding INCLUDES structure (at the B specification level) and IMPORTS (at the B implementation level), other forms of structure such as DAGs (directed acyclic graphs) cannot be generally translated.

This difficulty arises with phase decomposition in particular, in which controllers C_1 and C_2 for different phases may both need to send commands to the same actuator A. In B this would mean that the machines

$\mathcal{B}(C_1)$ and $\mathcal{B}(C_2)$ both INCLUDE $\mathcal{B}(A)$, which is an invalid structure.

Such 'shared include' structures are excluded in B because if M and N both include a machine S, then the operations of M may break the invariant of N by modifying the data of S, and likewise the invariant of M can be violated by operations of N. In the case of phase decomposition there is no actual danger of inconsistency because all phase controllers are included in an overseer controller which responds to events by forwarding the event to the controller for the currently active phase.

In the simple case, there are two phases, distinguished by the value of some sensor variable (*swstate* in the train control case study [12]) or an additional variable *pstate* giving the current phase. In this case the separate phases have invariants P_1 and P_2, say, and the supervisor controller has an invariant I:

$$(pstate = p1 \;\Rightarrow\; P_1) \;\wedge$$
$$(pstate = p2 \;\Rightarrow\; P_2)$$

The operations of the supervisor all have the form

$e \;=$
  IF $pstate \;=\; p1$
  THEN $p1\_e$
  ELSE $p2\_e$
  END

where $p1\_e$ is an operation of the first phase controller, and $p2\_e$ of the second. Then the invariant I is maintained by e because

$$I \;\Rightarrow\; (pstate = p1 \;\Rightarrow\; [p1\_e]I) \wedge$$
$$(pstate = p2 \;\Rightarrow\; [p2\_e]I)$$

since $P_1 \;\Rightarrow\; [p1\_e]P_1$ and similarly for P_2. I is true initially as the initialisation has the form

$$pstate := p1 \mid\mid p1\_init$$

if the initial state is $p1$. Any operation to change the mode has the form

$change\_mode \;=$
  IF $pstate \;=\; p1$
  THEN
    $pstate \;:=\; p2 \;\mid\mid\; p2\_init \;\mid\mid\; p1\_exit(p2)$
  ELSE
    $pstate \;:=\; p1 \;\mid\mid\; p1\_init \;\mid\mid\; p2\_exit(p1)$
  END

which also establishes I.

Therefore there is no proof overhead in establishing I as an invariant despite the shared includes structure. In general the sharing approach of [5] cannot be used to implement the DAG structure, because the operations of one phase controller will not preserve the invariant of the other. For example in the train control system of [12], the phase controller *SwOn* has the invariant $dstate : \{closing, closed, locked\}$, whilst *SwOff* has an operation

$swoff\_dbpressed \;=$
  IF $motstate \;=\; stationary$
  THEN $start\_open\_doors$
  END

which breaks this invariant. Instead, the approach of [3] can be used, with a machine K which EXTENDS two machines N and M which share by INCLUDES a machine S being expressed as

$$K \;=\; N \boxtimes M$$

5.2 Aggregate and Collection Structures

If a system contains several copies of a component, such as the press in the fault-tolerant production cell, then a specification of the system needs to express this collection in a concise and extensible manner. Ideally we could specify a component independently of how many copies of that component exist in the system, and then aggregate the number needed. B provides a simple syntactic mechanism of *renaming* to partly support this approach: if C is a machine, then $id.C$ is a new machine all of whose features (constants, variables and operations) are renamed by prefixing with id. Thus if we have specified a press controller as:

```
MACHINE Press
INCLUDES PressMotor, PressTimer
VARIABLES
  pstate, stm, pok, bs, lows, mids, ups
INVARIANT
  pstate : PState ∧
  pstate = w2r ⇒
      (bs = no or stm = FALSE) ∧
      lows = no ∧ mids = yes ∧ ...
INITIALISATION
  pstate := w2r || bs := no || ...
OPERATIONS
  blank_arrives =
     PRE pstate = w2r ∧ bs = no
     THEN
        bs := yes ||
        IF stm = TRUE
        THEN
          pstate := processing ||
          motor_up || set_ptimer(pmu)
        END
    END;

  ⋮

END
```

then the *PressSystemController* (the controller hierarchically above *PressController*) would need to INCLUDE two renamed copies of this machine: $press1.Press$ and $press2.Press$. This technique has several disadvantages:

1. It statically codes the number of copies of the component in the aggregation, so that this cannot be changed during an execution of the system;
2. It multiplies the number of variables and operations and hence increases analysis time;
3. The separate copies must both be implemented. Since prefixing does not apply to IMPORTed machines, the developer must carry out a textual renaming of the implementation of one copy to use in others.

For these reasons it is usually preferable to use a more generic approach involving parameterisation of the component module ('lifting' a specification of an instance of an entity E to be a specification of a class of entity instances [4]). A machine E representing an instance will have the form

MACHINE E
VARIABLES $att_1,\ att_2,\ \ldots,\ att_n$
INVARIANT
  $att_1 :\ T_1\ \wedge\ \ldots\ \wedge\ att_n :\ T_n\ \wedge$
  Inv_E
INITIALISATION $Init_E$
OPERATIONS
  $\alpha(x)$ = PRE Pre THEN S END;
  $\vdots$
END

Parameterisation of E by a set P of instance identifiers produces the machine:

MACHINE $E(P)$
VARIABLES $es,\ att_1,\ att_2,\ \ldots,\ att_n$
INVARIANT
  $es\ \subseteq\ P\ \wedge$
  $att_1 :\ es\ \rightarrow\ T_1\ \wedge\ \ldots$
  $att_n :\ es\ \rightarrow\ T_n\ \wedge$
  $\forall\ ei.(ei :\ es\ \Rightarrow\ Inv_E(ei))$
INITIALISATION $es,\ att_1,\ \ldots\ :=\ \{\}, \{\},\ \ldots$
OPERATIONS
  $ee\ \longleftarrow\ create\_E\ =$
    PRE $es\ \neq\ P$
    THEN
      ANY oo
      WHERE $oo :\ P\ -\ es$
      THEN
        $es\ :=\ es\ \cup\ \{\ oo\ \}\ ||$
        $Init_E(oo)\ \ ||$
        $ee\ :=\ oo$
      END
    END;

  $\alpha(ei, x)$ = PRE $ei :\ es\ \wedge\ Pre(ei)$
    THEN $S(ei)$ END;
  $\vdots$
END

where $Q(ei)$ denotes Q with each attribute reference att_j replaced by $att_j(ei)$. This approach provides the extensibility which the renaming approach lacks, and in addition only increases the number of variables by 1.

5.3 Reflexive Structures

Some components in a reactive system may play a role both as actuators (receiving commands from the control system) and as sensors (sending signals to the control system). This structure is permitted in SRS because communication from the sensor side to the controller is by event synchronisation, whilst commands are sent to actuators by generated events. Figure 3 shows an example taken from the production cell case study, where a *Timer* component is sent commands *start_timer* and *reset_timer* by the controller for the rotary table, and itself generates *timeout* events which are input to the controller. Similar examples of reflexive structures also arise in [6]. To express such structures in B we define components such as *Timer* as B machines as usual, but with operations for each command supported by the actuator, and operations to record the effect of each sensor event of the component (ie, spontaneous changes in its state). The B machine for *Timer* for example has the form:

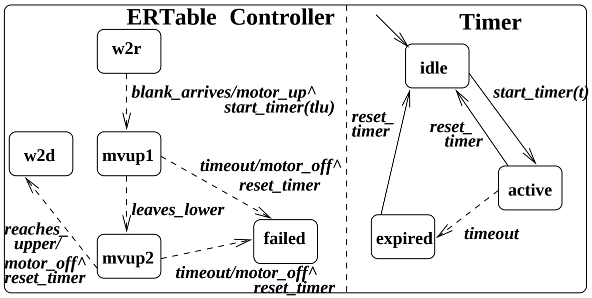

Figure 3. Example Reflexive Structure

MACHINE $Timer$
VARIABLES $tstate,\ setvalue$
INVARIANT
  $tstate :\ \{\ active,\ idle\ \}\ \wedge$
  $setvalue :\ \mathbb{N}$
INITIALISATION
  $tstate, setvalue\ :=\ idle, 0$

```
OPERATIONS
    start_timer(tt)  =
            PRE  tt :  ℕ
            THEN
               tstate   :=   active   ||
               setvalue   :=   tt
            END;

  reset_timer   =
         BEGIN
            tstate   :=   idle   ||
            setvalue   :=   0
         END
END
```

In this case the *reset_timer* operation can also be used to record the effect of a *timeout* event being received by the controller, and the controller response.

6 Fault Tolerant Production Cell

This system 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 [15] differs from the simple production cell [14] 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.

Some global safety properties of the system are:

$$robotm \neq off \;\Rightarrow\; arm1ext = pos\_ret \wedge arm2ext = pos\_ret$$

(if the robot is rotating, both its arms are fully retracted), and:

$$ms \neq off \;\Rightarrow\; stm = true$$

(if a press motor is on, it must be safe for the press to move). Violation of any of these invariants is caught by a failure detection controller (FDC) which puts the component or system into a *failed* state where all actuators are switched off.

6.1 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. For example, the robot controller needs to inform the press controllers of any movement of an arm towards them, ie, a change of state of *stm*. Thus there is a *Press System Controller* coordinating the robot and presses in Figure 4.

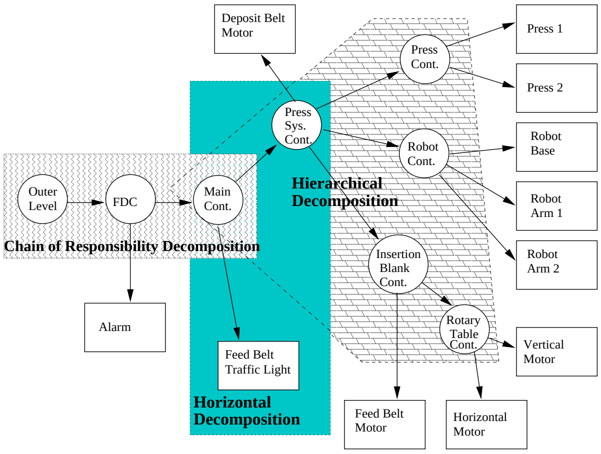

Figure 4. Decomposition of Controllers

In some cases components can be controlled entirely independently of other components, for example the feed belt traffic light control depends only on a single sensor S1 and on the failure status of the system, no invariants link it to other component states, so horizontal composition can be used. 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.

6.2 Invariants and control rules for normal behaviour

Invariants can be derived for the requirements of normal operational behaviour. In the case of the press, these include: (1):

$$pstate = w2r \wedge bs = no \wedge \bigcirc bs = yes \;\wedge\; stm = true \;\Rightarrow\; \bigcirc\, ms = up$$

(if the blank arrives while the press is positioned waiting for the blank, then start moving up, if it is safe to move), and (2):

$$pstate = w2r \wedge bs = no \wedge \bigcirc bs = yes \ \wedge$$
$$stm = true \ \Rightarrow \ \bigcirc pts = active \wedge \bigcirc pval = pmu$$

(also start the press timer and set its timeout period to pmu).

6.3 Translation to B

Using the process described in Section 4, the code of the controllers can be expressed in B by determining what controller actions are necessary to maintain the invariants. For a single press, invariants (1) and (2) lead to the following B operation:

```
blank_arrives =
    PRE pstate = w2r ∧ bs = no
    THEN
      bs := yes ||
      IF stm = TRUE
      THEN
        pstate := processing ||
        motor_up || set_ptimer(pmu)
      END
    END;
```

Abstraction morphisms are reflected in the B specifications via the use of variables expressing abstraction states ($pstate$: $\{failed, w2r, processing, completing, w2d, returning\}$ in this case) as well as variables representing concrete states. If $\sigma : C \rightarrow A$ is an abstraction morphism of modules, then the B machine for C has invariants expressing the definition of σ, in the form

$$astate = x \ \Rightarrow \ cstate : \sigma^{-1}[\{x\}]$$

for each possible abstract state x. Any state change $cstate := y$ in an operation must be combined with a state change $astate := \sigma(y)$.

7 Ordering of Events by the Outer Level

If a polling approach is taken (or events are communicated to the control system across a communication system which does not guarantee preservation of event orders) then the outer level of B specification can only determine that a certain set of events happened between two polling cycles, it does not have information on the order of these events. Therefore a decision has to be made on what order events should be sent to the sequential control system defined in SRS and B.

In general events may be associated with a priority which enforces a certain order in their consideration. This is related to the level of hazard which they may lead to: if events e_1 and e_2 are detected to occur in a given interval, then we must deal with the event which could lead to the most serious hazard condition first.

In particular events to which the reaction can be delayed temporarily without safety consequences should be scheduled after more urgent events.

In the train case study of [12], if events $db\_pressed$ and $becomes\_moving$ are both detected as having occurred, then they should be ordered as $becomes\_moving$; $db\_pressed$ because opening the doors can be postponed, whilst delaying the reaction to the train starting to move can lead to hazard situations (eg: the doors opening while the train is moving). Figure 5 shows that processing the $db\_pressed$ event first leads to a hazard state where the actual motion state is $moving$ but the control system is not aware of this and so allows the door to begin opening.

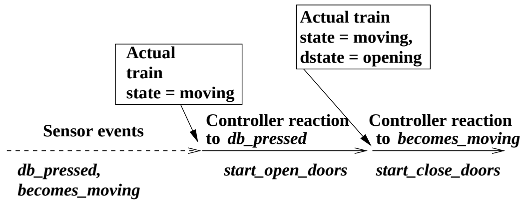

Figure 5. Train System Hazard

In the case of the gas burner system of [10] the events of $switch\_on$ and $flame\_appears$ likewise must be ordered as $flame\_appears$; $switch\_on$. The reaction to the flame spontaneously appearing (leading to the hazard state $fdstate = on \wedge avstate = closed$) takes precedence over starting up the cell. Again if the reaction to $switch\_on$ was taken first, a hazardous action (opening the gas valve with the flame already present) would occur.

In the fault-tolerant production cell, sensors $S1$ (at the beginning of the feed belt) and $S2$ (at the end) can both change from no to yes in the same cycle. If the motor is active, the resulting actual system state after these two events is

$$S1 = yes \wedge S2 = yes \wedge bm = yes \wedge fbtl = green$$

The most hazardous element here is the motor being on ($bm = yes$) whilst a blank is at $S2$. Thus the events must be ordered as $S2\_blank\_arrives$; $S1\_blank\_arrives$. The possibility that blanks are put on the start of the feedbelt while another is there is less critical than a blank falling off the end.

A hazard analysis of a system should therefore produce priority measures for events, and allow a systematic construction of the outer level component. Ideally, priorities could be assigned to sensors, so that sensors can be analysed separately for the events which have occurred. In the case of the gas burner [10] the ordering is: *flame detector*; *switch*. In the case of the train system the ordering is *motion sensor*; *door sensor*; *train switch*; *door button*.

In the fault tolerant production cell, any sensor which detects the arrival of a press, robot arm, robot or table at particular vertical or rotational positions has a high priority, since the events for these all require quick reactions (to stop the appropriate motors). Similarly $S2\_blank\_arrives$ is in this category. At a lower priority level are the other blank sensors (which generally act as initiators of movement, rather than terminators).

Conclusions

We have described how structured modular statechart descriptions can be mapped into structured specifications in B, and how the forms of structures required by reactive system design can be expressed within B. The SCR approach of Heitmeyer et al. [8] also addresses the specification and analysis of reactive systems, however it emphasises mode-based decomposition instead of hierarchical decomposition, and proof of single-state properties is performed by model checking rather than the use of proof in B. Our approach is primarily aimed at providing a graphical front-end to development in B, so it uses (suitably restricted) statechart notations and explicit subsystem decomposition instead of tables and intermediate variables to express state changes in response to events.

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] D. Bert, M-L. Potet, Y. Rouzaud. *A Study on Components and Assembly Primitives in B*, Proceedings of 1st B Conference, pp 47–62, 1996.

[4] J.C. Bicarregui, K.C. Lano, T.S.E. Maibaum, *Objects, Associations and Subsystems: a hierarchical approach to encapsulation*, ECOOP 97, Springer Verlag LNCS, 1997.

[5] M. Büchi, R. Back, *Compositional Symmetric Sharing in B*, FM '99, Springer-Verlag LNCS, 1999.

[6] A. Chavoya, A. Sanchez, K. Lano, *Formal Implementation of Procedural Controllers for Event-Driven Sequential Systems in Chemical Processes*, CINVESTAV, Mexico, January 2000.

[7] I. Hayes, *A Survey of Data Refinement and Full Abstraction in VDM and Z*, Dept. of Computer Science, University of Queensland, 1991.

[8] C. Heitmeyer, *Formal Methods for Developing Software Specifications: Paths to Wider Usage*, Naval Research Laboratory, Washington, DC 20375, USA, 1999.

[9] International Society for Measurement and Control. *Batch Control Models and Terminology*, ISA-S88.01-1995, 1995.

[10] K. Lano, A. Sanchez, *Design of Real-time Control Systems for Event-driven Operations*, FME '97, Springer-Verlag LNCS, 1997.

[11] 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.

[12] 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.

[13] K. Lano, D. Clark, K. Androutsopoulos, P. Kan, *Invariant-based Synthesis of Fault-tolerant Systems*, FTRTFT 2000, Pune, India, 2000.

[14] C. Lewerentz, T. Lindner (eds.), *Formal Development of Reactive Systems*, LNCS Vol. 891, Springer-Verlag, 1995.

[15] A. Lötzbeyer, R Mühlfeld, *Task Description of a Flexible Production Cell with Real Time Properties*, FZI, Karlsruhe, 1996.

[16] F. Mejia, *Formalising Existing Safety-Critical Software*, FMERail Workshop No. 2, London, UK, 1998. `http://www.ifad.dk/Projects/fmerail.htm`.

[17] A. Sanchez. *Formal Specification and Synthesis of Procedural Controllers for Process Systems*. Springer-Verlag. Lecture Notes in Control and Information Sciences, vol. 212. 1996.