Invariant-based Synthesis and Composition of Control Algorithms using B

K. Lano[1], J. Bicarregui[1], and A. Sanchez[2]

[1] Dept. of Computing, Imperial College,
180 Queens Gate, London SW7 2BZ
[2] Departmento de Ingenieria Electrica.
CINVESTAV-Guadalajara.
Apdo. Postal 31-438.
Guadalajara 45090, Jalisco, Mexico

Abstract. This paper describes techniques for the automatic synthesis of verified controllers for discrete event systems, based on the invariants of behaviour required for such systems. We define alternative structuring approaches for controllers, and the correspondences between this decomposition and structuring in the B formal method, which is used to provide an implementation of the control systems.

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 will be represented as a finite state machine or as a statechart [13]. In current practice, control algorithms are developed by hand, thus introducing possibilities for perhaps very expensive or life-threatening design faults.

The automatic synthesis of control algorithms has the potential to reduce errors in the development of discrete event control systems, and make the development process for such systems more systematic and hence assessable. The emphasis in development can therefore shift to earlier requirements analysis and elicitation phases, with benefits for the correctness and safety properties of such systems.

In the method developed in the ROOS project 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.

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,

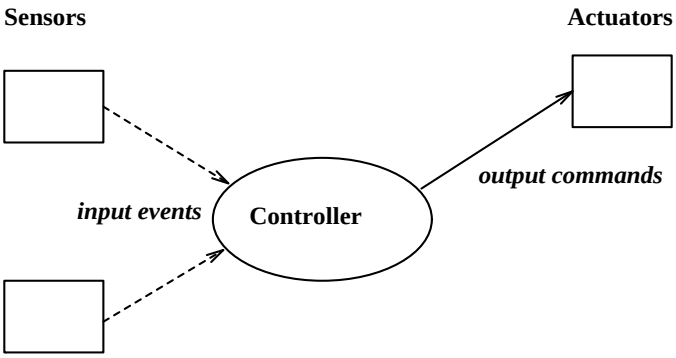

Fig. 1. Structure of Reactive Control Systems

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 [7].

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

The design pattern Chain of Responsibility [6] is particularly relevant to the first approach, whilst the design pattern State is relevant as a means of implementing the third approach.

2 Case Studies

In this paper we will use two simple case studies for illustration purposes. More complex case studies are described in the paper [8].

2.1 Gas Burner

This system controls the valves and igniter of a simple gas burner (Figure 2). When the switch is pressed the controller should try to move the system into a state where the air valve is open, the gas valve is open, the flame is ignited and the igniter is off. If flame appears while the switch is off, the air valve should be opened. The states of all components are binary: $avstate : \{open, closed\}$

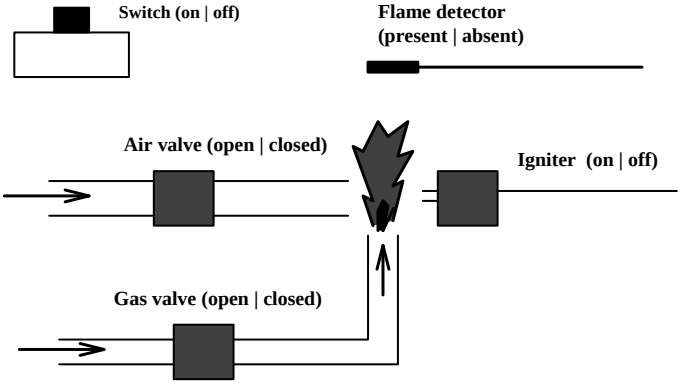

Fig. 2. Components of Gas Burner

represents the air valve state; $gvstate : \{open, closed\}$ represents the gas valve; $istate : \{on, off\}$ the igniter state; $swstate : \{on, off\}$ the switch state, and $fdstate : \{present, absent\}$ the flame detector state.

The invariants of operational behaviour are:

1. $swstate = on \Rightarrow gvstate = open \wedge avstate = open$
2. $swstate = on \wedge fdstate = absent \Rightarrow istate = on$
3. $fdstate = present \Rightarrow avstate = open \wedge istate = off$
4. $swstate = off \Rightarrow gvstate = closed \wedge istate = off$
5. $swstate = off \wedge fdstate = absent \Rightarrow avstate = closed$

An operational constraint is also that the igniter cannot be on unless the gas valve is open:

$$istate = on \Rightarrow gvstate = open$$

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

$$gvstate = open \Rightarrow avstate = open$$

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

Its actuators are:

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

The safety invariants are:

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

3 Reactive Systems Development Using B

B [1] is a model-based formal method which provides modules, known as *machines*, encapsulating data and operations on that data. Modules can be accessed from other modules via mechanisms such as *INCLUDES*, *SEES*, etc. There is extensive tool support for B provided by the B Toolkit [4] and Atelier B [3] toolsets.

The following steps are taken to specify in B a controller for a discrete event system, described in terms of a set of actuators and sensors, and required reactions to events detected by the sensors, and invariants that the state of the system must satisfy [8].

1. Produce a data and control flow diagram (DCFD) of the entire control system, showing all signals from sensors to the controller and all commands from the controller to actuators.
2. Produce component models – describing the events each device (sensor or actuator) generates or responds to. These models are given as state machines.
3. Formalise: (i) required reactions to events; (ii) safety properties/invariants.

4. Partition the DCFD where possible into a set of controllers each managing a disjoint set of actuators.
5. Specify controllers in B: each input event responded to by the controller has a corresponding operation which describes that response (for each possible mode of the controller).
   The invariant of each controller expresses local control invariants for the particular collection of sensors and actuators that it manages.
6. Specify sensors/actuators in B: these specifications represent the controllers *inferred knowledge* about the physical system, on the basis of events it has received and produced.
7. Implement controller using library components. Procedural controller synthesis [12] is used to derive the control algorithms (ordering of actuator signals) required for each event response.
8. Implement sensor/actuator specifications – these will be linked eventually to actual physical devices.
9. Specify and implement an 'outer level' component which detects input events (eg: by polling sensors, by extraction from message queues, etc) and notifies the controller that they have occurred. This is usually defined in terms of the controller modes, following [7].

Animation can be applied to the controller specifications to check that the specified behaviour is actually what the user intends, and that it avoids hazard scenarios.

Proof can be applied to specifications to check internal correctness of machines and interfaces: ie, that the invariants of the machines are maintained by their operations, and that operations are called only in situations where their preconditions hold. Proof can be applied between specifications and implementations to check that the implemented response behaviour meets the specifications[1].

4 Synthesis of Control Algorithms from Invariants

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

A typical operational invariant will have 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 'on' and a guard is satisfied.

Then the reaction to any event which 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 $astate = on$.

[1] Although proof of temporal properties is outside the scope of the B tools at present.

A safety invariant will often have the converse form, ie:

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

stating that the actuator is only activated when it is required and it is safe to operate.

This means that the reaction to any event which sets $sstate \neq s1$ or which invalidates G must set *astate* off if it is not already off.

More precisely, the algorithm for synthesising an abstract B specification from the invariants is as follows:

- For each event e (which affects the state of a particular sensor, say *sstate*):
  1. identify all invariants which may be invalidated by this state change – ie, invariants of the form

     $$sstate = s1 \wedge G1 \;\Rightarrow\; astate = a1$$

  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. 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.
  3. Compose changes to different actuators by ||.

Consider the gas burner system operational invariants given in Section 2. Together they can be used to generate a complete abstract control algorithm (without orderings of individual component actions). If the initial state of the system is $(off, absent, closed, closed, off)$ where we order component states as $(swstate, fdstate, avstate, gvstate, istate)$, then we have the algorithm presented in Figure 3. The axioms which give the effect of each transition are shown beside it. The B code of the operations is as follows:

```
switch_on =
  PRE swstate = off
  THEN
    swstate := on ||
```

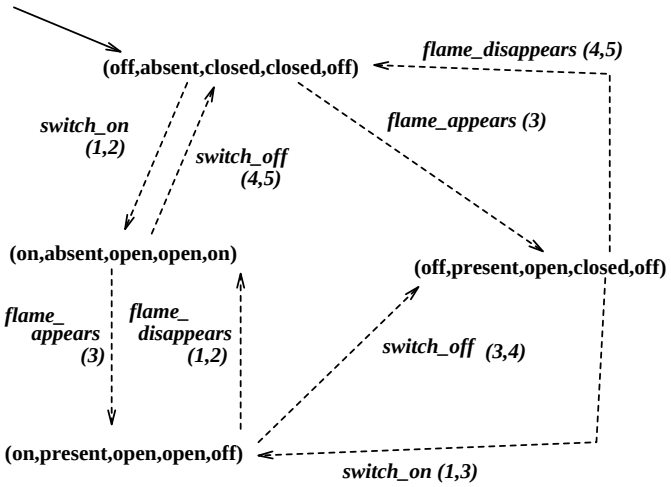

Fig. 3. Abstract Derived Gas Burner Controller

```
    gvstate := open ||
    avstate := open ||
    IF fdstate = absent
    THEN
      istate := on
    END
  END;

switch_off =
  PRE swstate = on
  THEN
    swstate = off ||
    gvstate := closed ||
    istate := off ||
    IF fdstate = absent
    THEN
      avstate := closed
    END
  END;

flame_appears =
  PRE fdstate = absent
  THEN
    fdstate := present ||
    avstate := open ||
    istate := off
  END;

flame_disappears =
```

```
PRE fdstate = present
THEN
  fdstate := absent ||
  IF swstate = on
  THEN
    istate := on
  ELSE
    avstate := closed
  END
END
```

Axioms can also be used to derive the orderings of individual actuator commands within each control reaction. The safety constraint $gvstate = open \Rightarrow avstate = open$ means that the air valve must be opened before the gas valve, in reactions where both are opened. In reactions where both are closed, the gas valve must be closed first.

The operational constraint $istate = on \Rightarrow gvstate = open$ means that the gas valve must be opened before the igniter is switched on, in reactions which set both actuators on. In reactions which set both off, the igniter must be switched off first.

In some cases, a number of possibilities for the ordering of actuator commands within control reactions will remain after such analysis. The designer must then make some selection of an admissible ordering.

A safety invariant of the form

$$astate = x \Rightarrow sstate = s$$

can be rewritten as the logically equivalent

$$sstate \neq s \Rightarrow astate \neq x$$

which can be treated as for the operational invariants described above: any event which results in *sstate* leaving state s must also ensure $astate \neq x$ (if there is more than one possibility for the new value of *astate*, then the designer must specify a value).

An invariant of the form

$$astate = x \Rightarrow sstate_1 = s_1 \wedge \ldots \wedge sstate_n = s_n$$

is equivalent to the separate invariants

$$sstate_1 \neq s_1 \Rightarrow astate \neq x$$
$$\vdots$$
$$sstate_n \neq s_n \Rightarrow astate \neq x$$

As another example, operational invariant 4 of the train control system implies that *becomes_stationary* has the abstract reaction:

```
becomes_stationary =
  PRE motstate = moving
  THEN
    motstate := stationary ||
    IF dbstate = on & swstate = off
    THEN
      start_open
    END
  END
```

All the operations of this system can be synthesised from the invariants given above in a similar way.

5 Guidelines for Reactive Controller Structuring

Two contrasting approaches to decomposing the controller of a reactive system have been developed. A "top-down" approach based on the system invariants is described in this section. This tends to produce quite large controller modules based around physical subparts of the controlled system, or alternative modes of its behaviour. Alternatively, an approach based on synthesis of necessary control functionality from chaining of small controller modules can be taken (Sections 7, 8). This "bottom-up" approach tends to produce lots of small control modules akin to logic gates.

The invariants of a reactive system can be used to select suitable controller decompositions, as follows.

Fault Detection If there is an invariant I on a group of sensor states only, then this represents an assumption about the normal state of the environment. Fault-detection tests can be used to check if this fails to hold, and raise an exception or force a change of mode to a fault-handling mode. These should be in fault-detection layers which are wrappers around the main controller.

Horizontal Decomposition If there are disjoint groups of actuators A and B and no invariants linking any actuator in A to any in B, then these groups can be controlled independently by horizontal decomposition.

Vertical Decomposition Conversely, if there is some invariant linking a set A of actuators and S of sensors, then there must be some supervisor controller which manages (directly or indirectly) each of these actuators, in order to ensure the invariant, and which has events for all of the events generated from the sensors in S.

Subordinate Controllers If there are invariants which link a proper subset of the actuators, then this suggests the creation of a subordinate controller which manages this set. This is particularly so if the states are completely linked, ie, the value of one actuator state is a function of one or more others.

These guidelines have been applied in the case studies of a steam boiler [2, 5], which used fault-detection layers and vertical composition, and the production cell [9, 8], which used horizontal and vertical composition. In the train control

system it can be seen that the brake and motor states are functionally linked, so should be managed by a local Motor/Brake controller.

6 Standard Sensors and Actuators

A very simple form of sensor is a binary switch (Figure 4). The corresponding

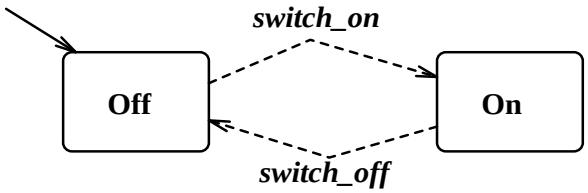

Fig. 4. STD of Switch

template B machine is:

```
MACHINE Switch
SETS State = {on, off}
VARIABLES sstate
INVARIANT sstate: State
INITIALISATION sstate := off
OPERATIONS
  switch_on =
    PRE sstate = off
    THEN sstate := on
    END;

  switch_off =
    PRE sstate = on
    THEN sstate := off
    END

END
```

A common three-state model is a timer structure, with two controlled transitions *start* and *reset* and an uncontrolled transition for *expires* (Figure 5). This model is also applicable to many other devices, such as a bar-code reader [8].

Another frequently encountered actuator/sensor is a 4-state model such as a simple valve (Figure 6). This has the corresponding B model:

```
MACHINE Valve
SETS VState = {closed, opening, open, closing}
VARIABLES vstate
INVARIANT vstate: VState
INITIALISATION vstate := closed
```

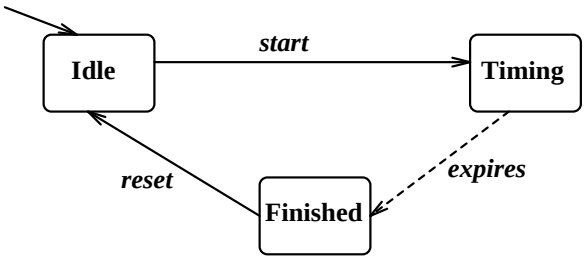

Fig. 5. STD of Timer

```
OPERATIONS
  start_open =
    PRE vstate: {closed, closing}
    THEN vstate := opening
    END;

  start_close =
    PRE vstate: {open, opening}
    THEN vstate := closing
    END;

  completes_open =
    PRE vstate = opening
    THEN vstate := open
    END;

  completes_close =
    PRE vstate = closing
    THEN vstate := closed
    END
END
```

A version with failures is given in Figure 7.

6.1 Standard Data Stores/Repositories

Often, auxiliary information is needed for a control algorithm. For example, fault detection may require that we keep a record of commands sent in the previous cycle to detect if these commands have been carried out or not [5]. Repository components can also be defined as state machines, usually with attributes.

The most basic example is a single variable of bounded numeric or enumerated type, with actions $update(v : T)$ and $read() : T$ to modify and access this value. Such components are immediately implementable using the B library components *Nvar* and *Vvar*.

Other necessary data structures include sequences with actions $add(x : T)$, $remove()$, $head() : T$, $is\_empty() : BOOL$, etc. Sequences are directly implementable using the B *seq* and *seq_obj* machines.

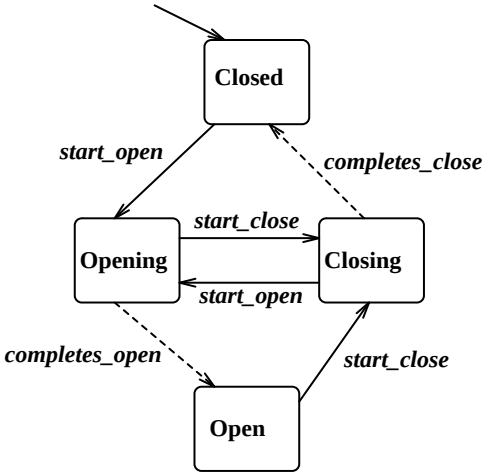

Fig. 6. STD of Valve

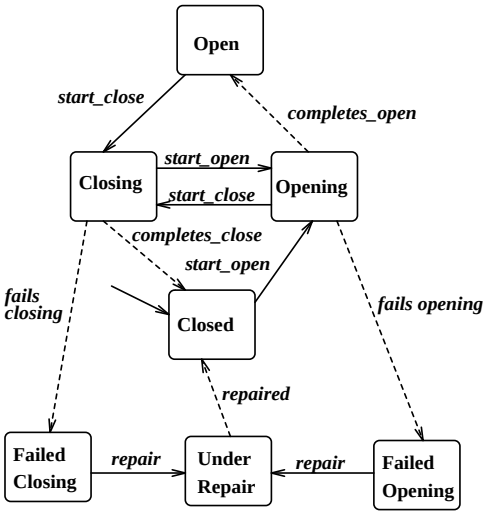

Fig. 7. STD of Valve with Failures

7 Patterns for Control Algorithms

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.

7.1 And Combination of Inputs

This controller 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*). Figure 8 shows the structure of this control system. A similar algorithm can be used if the output is produced

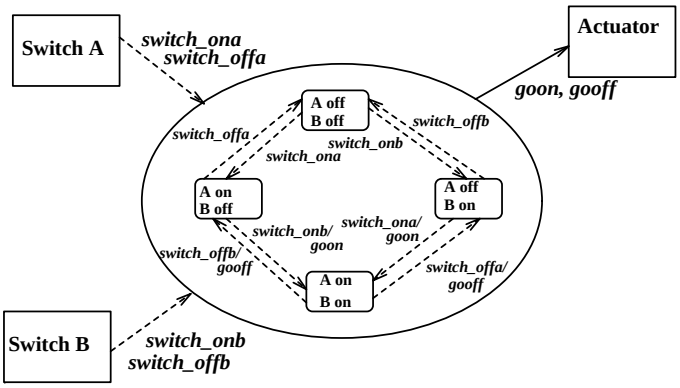

Fig. 8. And Controller System

if *either* switch goes on (an *OrController*).

7.2 Priority Controller

In this case two actuators are controlled. The controller 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. Figure 9 shows the system in this case.

The B controller specification is:

```
MACHINE PriorityController
INCLUDES ActuatorA, ActuatorB
SETS State = {on, off}
VARIABLES sastate, sbstate
INVARIANT sastate: State & sbstate: State &
```

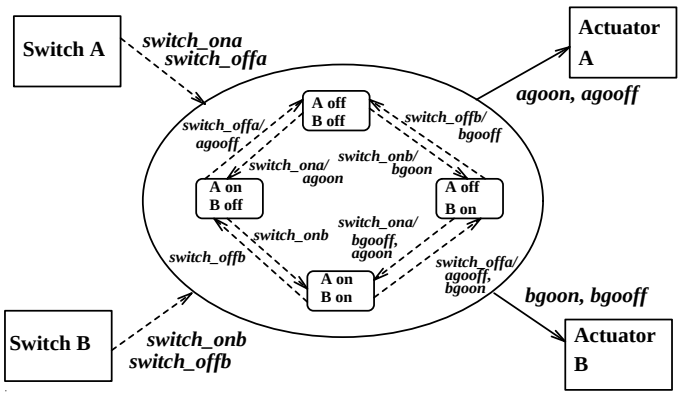

Fig. 9. Priority Controller System

```
  (sastate = off => aastate = off) &
  (sbstate = off => abstate = off) &
  (sastate = on  => aastate = on) &
  (sbstate = on & sastate = off => abstate = on) &
  (sastate = on  =>  abstate = off)
INITIALISATION
  sastate := off || sbstate := off
OPERATIONS
  switch_ona =
    PRE sastate = off
    THEN
      sastate := on ||
      IF sbstate = on
      THEN
        bgooff
      END ||
      agoon
    END;

  switch_onb =
    PRE sbstate = off
    THEN
      sbstate := on ||
      IF sastate = off
      THEN
        bgoon
      END
    END;

  switch_offa =
    PRE sastate = on
    THEN
```

```
      sastate := off ||
      agooff ||
      IF sbstate = on
      THEN
        bgoon
      END
    END;

  switch_offb =
    PRE sbstate = on
    THEN
      sbstate := off ||
      IF sastate = off
      THEN
        bgooff
      END
    END

END
```

7.3 Valve Controller

In this case a single switch controls a single valve actuator/sensor. The B code is:

```
MACHINE ValveController
INCLUDES Valve
PROMOTES completes_open, completes_close
SETS State = {on, off}
VARIABLES sstate
INVARIANT sstate: State &
  (sstate = on  =>  vstate: {opening, open}) &
  (sstate = off =>  vstate: {closing, closed})
INITIALISATION sstate := off
OPERATIONS
  switch_on =
    PRE sstate = off
    THEN
      sstate := on ||
      start_open
    END;

  switch_off =
    PRE sstate = on
    THEN
      sstate := off ||
      start_close
    END

END
```

8 Composition by Chaining

The output (as a stream of commands) of one of these standard controllers can be used as input for another. Eg: we can chain an And controller into a Valve controller to control a system where the valve must be on iff two sensors are both on. An n-ary And controller can be formed by chaining n-1 binary And controllers together.

A particular example of composition is the train controller. We can express this as a composition of a priority controller (with inputs from the switch (priority) and door switch), outputting to the motor/brake aggregate (in place of actuator A) and to an And controller whose other input is the motion sensor (going stationary acts as the switch on in this case). The output of this And controller goes to a valve controller whose output goes to the door actuator. Figure 10 shows this.

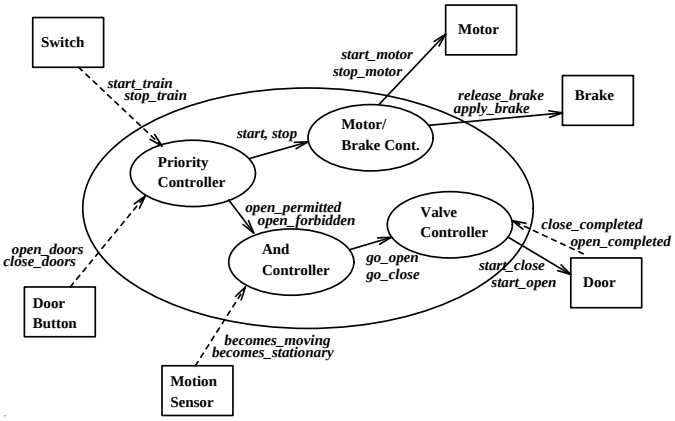

Fig. 10. Composed Train Controller System

This form of composition is at a very low level compared with that of horizontal and vertical composition, and is akin to composition of logic gates in digital circuitry.

9 Tool Development

A suite of tools has been developed in Visual C++ for the definition and transformation of state machine descriptions of a reactive system, and the translation of these state machines into B specifications and implementations [10]. The following facilities are provided:

1. Checks on the existence or correctness of abstraction mappings between two state machines, including construction of such mappings where they exist.

2. Transformations of statecharts and reactive system descriptions with nesting, conditions, event calling and AND composition into simple state machines.
3. Translating a state machine into a B machine for purposes of invariant checking and animation/testing.

Conclusions

We have defined techniques for the synthesis of control algorithms from formal requirements statements, and a method for their implementation using the B formal method and tool support.

We are extending the tool support for the graphical development of reactive control systems using the process of Section 3, with facilities for the design of actuator and sensor models, the automated synthesis or manual development of control algorithms, and verification of safety and operational invariants on these models.

References

1. J-R Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J R Abrial, E Borger, H Langmaack (Eds.), *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, LNCS 1165, Springer-Verlag, 1997.
3. Digilog Ltd., Atelier B, `http://www.atelierb.societe.com/index_uk.html`, 1998.
4. B Core UK Ltd., B Toolkit, `http://www.b-core.com/btoolkit.html`, 1998.
5. M. Ali, *B Specification of Steam Boiler*, MSc thesis, Dept. of Computing, Imperial College, 1998.
6. E Gamma, R Helm, R Johnson and J Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1994.
7. International Society for Measurement and Control. *Batch Control Models and Terminology*, ISA-S88.01-1995, 1995.
8. K. Lano, P. Kan, *Combining Scheduling Theory and Formal Methods in the Development of a Flexible Manufacturing System*, 4th Theory and Formal Methods Workshop, University of Bath, 1998.
9. A. Lötzbeyer, R. Mühlfeld. *Task Description of a Flexible Production Cell with Real Time Properties*, FZI, Karlsruhe, 1996.
10. A. Patel, *Safety Analysis of State Machine Models*, MSc thesis, Dept. of Computing, Imperial College, 1998.
11. PRESTO P4 Project. *Integrated Design of Control and Automation Systems*, PRESTO Document 200197A11, Centre for Process Systems Engineering, Imperial College, 1997.
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.
13. Rational Software et al, *UML Notation Guide*, Version 1.1, http://www.rational.com/uml, 1997.