**SOFTWARE VERIFICATION RESEARCH CENTRE**

**SCHOOL OF INFORMATION TECHNOLOGY**

**THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072**
**Australia**

**TECHNICAL REPORT**

**No. 99-31**

# A Case Study in Software Safety Assurance Using Formal Methods

**Brenton Atchison, Peter Lindsay, David Tombs**

**September 1999**

**Phone: +61 7 3365 1003**
**Fax: +61 7 3365 1533**

**Note:** Most SVRC technical reports are available via anonymous FTP, from svrc.it.uq.edu.au in the directory /pub/SVRC/techreports. Abstracts and compressed postscript files are available via http://svrc.it.uq.edu.au.

# A Case Study in Software Safety Assurance Using Formal Methods

Brenton Atchison, Peter Lindsay, David Tombs

Software Verification Research Centre
School of Information Technology
The University of Queensland
Queensland 4072, Australia

email: {brenton,pal,tombs}@svrc.uq.edu.au

**Abstract**

This report describes a formal approach to verification and validation of safety requirements for embedded software, by application to a simple control-logic case study. The logic is formally specified in Z. System safety properties are formalised by defining an abstract model of the system's physical behaviour in Z, including its hazardous states and dominant sensor failures. The Possum specification-animation tool is then used to check that the logic meets its safety requirements. Finally, the logic is implemented in SPARK Ada and SPARK Examiner is used to formally verify the implementation meets its specification. Design safety validation and source code verification are completely automated, removing the need for human intervention.

**Keywords:** safety-critical systems, formal methods, safety assurance, V&V

## 1 Introduction

This report describes a formal approach to assuring safety of the design and implementation of embedded software for simple control systems. The approach differs from more traditional formal approaches in that much of it is fully automated and it largely avoids the need for complex mathematical proofs. It is thus likely to be far more cost-effective, while offering as high (and arguably even higher) assurance.

The approach is illustrated on the development of a safety argument for a simple case study concerning the software control logic for a hydro-mechanical press. Background to the case study is presented in section 2, including the operational concept and system architecture.

### 1.1 Approach to Safety Assurance

There follows an outline of the proposed software safety assurance process:

1. System safety requirements are assumed to have been derived by an appropriate hazard analysis, including consideration of possible software-input (sensor) failure modes. For the case study, the results of such an analysis are described in section 3, but a full description of the task is outside the scope of this report. The requirements are expressed as properties of the press's physical behaviour and control system sensor and actuator values. [Atchison, 1997 #8] contains more details of the hazard analysis activities.
2. Control-logic design is expressed as a finite-state input/output machine. Safety of the logic design is validated by analysing all possible behaviours of the logic in its operational environment (including the possibility of single sensor failures). Formally this is achieved by specification animation using a Z specification of the control logic and an abstract model of the press and its sensors and actuator. The behaviours leading to hazardous system states are analysed, and it is

argued that the residual risk of logic-related system failures is acceptably low. The software design is presented in section 4 and the safety validation process is described in section 5. The Possum tool [Hazel, 1997 #7] is used for specification animation.

3. The control logic is implemented in SPARK Ada, with formal annotations derived directly from the Z specification. (With appropriate tool support this step could be fully automated.) The SPARK Examiner toolset [2] is used to formally verify that the implementation meets its specification. The process used is described in sections 6 and 7.

The result is a fully tool-supported safety argument for the control-logic software for the press, such as could form the core of a software safety case. (Safety case ingredients not covered here include failures' likelihood, and system-integration test results showing that the installed software behaves as implemented.)

The approach made it possible to discover deficiencies in the control-logic design, and to replay the analysis automatically upon making modifications to the logic. The systematic, repeatable nature of the approach represents a significant improvement over manual processes, without the overhead of full formal development. As such, we believe it has the potential to be a highly cost-effective, high integrity approach to development of safety assurance for embedded software.

There are necessarily some activities of the safety assurance process that cannot be treated by formal functional analysis, in particular the assessment of failure likelihoods and residual risk. It is intended that the analysis presented in this paper will provide information for these activities but they are not presented here.

## 1.2 Formal Specification Notation

The Sum specification language [5] is used to specify both the control logic and operational environment. Sum is a variant of the Z specification language [6] devised primarily to facilitate the production of modular specifications and ease specification readability. Unique features of the Sum language relevant to this case study are:

1. A collection of declarations and definitions may be grouped into a module. Modules may be imported, giving visibility to the referenced entities.
2. State machines are easily represented by modules through the use of predefined *State, Init* and *Op* schemas. *State* schemas represent the state encapsulated by the module through a collection of typed variables. The state is initialised by the *Init* schema. State transitions are captured by *Op* schemas which specify the relationship between state variables before and after a transition. The modified state variables are identified by an appended dash. The scope of variables that can be referenced by the *Init* and *Op* schemas is restricted to the module *State* schema variables by default but can be extended arbitrarily. The *changes_only* expression in a schema specifies which part of the state may be changed by an operation.
3. Preconditions can be explicitly associated with *Op* schemas in order to convey more information about the intended specification. A precondition is identified with the prefix *pre* and represent assumptions about the state prior to invocation of an operation.

Possum interprets queries made in Sum and responds with simplifications to those queries. Arbitrary Sum expressions and predicates can be evaluated and a Sum state machine can be "executed" by stepping consecutively through operations of that machine with active (true) preconditions.

The control logic Sum specification is manually translated to SPARK Ada annotations which provide a functional specification embedded within the program. The SPARK analysis tools enable proofs that the program satisfies the specification.

# 2 Case Study

## 2.1 Operational Concept

The (hypothetical) case study is based on a system first described by the HISE group at the University of York [3] and which was purportedly inspired by a real system. The case study concerns a 50 tonne hydro-mechanical press which is used to produce body parts for a certain make of motor vehicle. The press is loaded and unloaded by a single operator. Unformed sheets of metal (workpieces) arrive on a conveyor belt roughly once per minute. The operator loads a workpiece from the roll-off area into the press, then pushes a button which causes the press to close: that is, the plunger falls to the bottom under gravity, pressing the workpiece into its desired shape. The press then opens again, and the operator unloads the formed product from the press and places it onto a second conveyor belt.

The press is opened by activating an electric motor and engaging clutches which drive hydro-mechanical winding gear. The press plunger is held against the top stop by running the motor continuously. There is a point, called the point of no return (PoNR), after which it is pointless – and may in fact be dangerous – to try to open a closing press because the falling plunger's momentum is so great.

Under normal operation, the press will close in approximately 2 seconds, and open in approximately 4 seconds. There would normally be 420 operations of the press per day. The industrial press is illustrated in Figure 1.

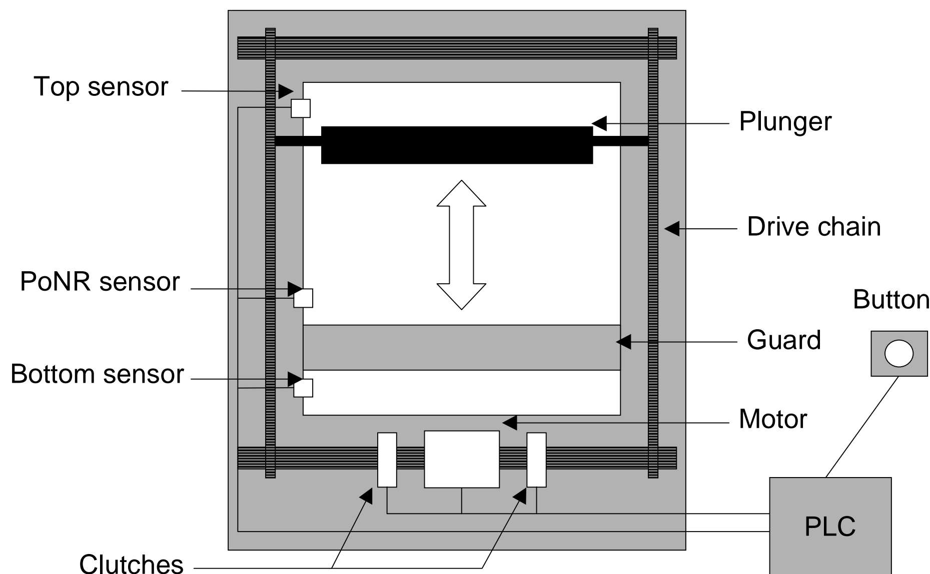


*Figure 1 - Industrial Press*

## 2.2 System Architecture

The system architecture extends existing hydro-mechanical winding gear with the push button, position sensors and a PLC based control system. A functional block diagram of the architecture is illustrated by Figure 2.

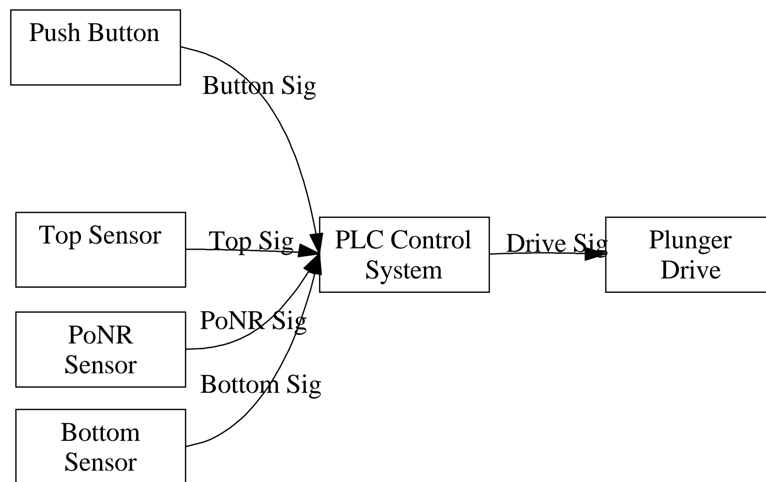


*Figure 2 - Industrial Press Control System Architecture*

The position of the press plunger is measured by three micro-switch sensors, positioned at the top, bottom and physical point of no return, which are 'toggled' by a lever fitted to the centre of the plunger whenever the plunger passes the switch. Table 2-1 indicates how the sensor values are interpreted.

| Sensor | Interpretation of high signal |
|---|---|
| Top | plunger is at top of travel |
| PoNR | plunger is below point of no return |
| Bottom | plunger is at bottom of travel |
| Button | button is pressed |

*Table 2-1 - Press System Sensor Signals*

The control logic of the press is implemented in software executing on a PLC. It scans sensor signals from an input register at frequent intervals and writes the motor drive signal to an output register where appropriate. The motor drive signal is scanned by electronic components that activate the electro-mechanical clutch and motor system.

# 3 Safety Analysis

Although a complete safety analysis is outside the scope of this report, a summary of results is provided by way of context.

The press includes a physical guard which allows the operator to put his or her arms into the press, but not the head or torso. The main remaining operational safety hazard is that the operator, or a second person, will have his or her hands crushed by the closing press. This hazard is to be mitigated by inclusion of an "abort" facility, whereby the motor drive will be engaged if the button is released while the plunger is falling above the PoNR.

Any attempt to raise a plunger falling below the PoNR is hazardous, since it will slow plunger descent without actually stopping it reaching the bottom (thus giving the operator more time to put his hands in a closing press), or may even cause the winding gear to break. A partial mitigation of this hazard is ensure that the button is placed far enough away to allow a plunger falling past the PoNR to reach the bottom before the operator can travel from the button to the press.

The "primary" system safety requirements for normal (fault-free) operation are:

1. The motor drive shall be active when button is released while plunger is above the PoNR.
2. The motor drive shall not be activated when the plunger is falling below the PoNR.

A further, "secondary" safety requirement is that sensor failures should not cause a hazard. More precisely, it will be required that all single critical persistent sensor failures be detected and revealed within one operational cycle of the press. Sensor failures may have a variety of causes, including electrical and mechanical faults. Persistent failures (such as breakages) only shall be considered here, since they are the most likely and the most pernicious (especially if allowed to go undetected for many operational cycles of the press). In a full safety case, these and other possible hardware failures would be identified and assessed by a separate analysis such as an FMEA [4].

Rather than derive software specific safety requirements, we present a model of the software design and investigate whether safety is preserved under operating conditions, even in the presence of single persistent sensor faults.

# 4 Software Design

This section describes the control logic chosen for the case study. This report will not attempt to record how this particular design was chosen, except to say that the logic corresponds closely to the intuitive operation of the press as described above, with tests for physically impossible sensor-value combinations. Section 5 below presents the detailed assurance to show that the design meets the safety requirements described above.

## 4.1 Informal Design Specification

The software is designed with typical scan architecture consisting of input module, control logic module and output module. Normal operation of the software is a repeated cycle consisting of:

1. Scan inputs from sensors;
2. Execute control logic; and
3. Write outputs to motor drive actuator

The state transition diagram in Figure 3 defines the control logic module. Sensor value combinations not represented in the diagram result in a null transition. Under normal (fault-free) operation of the press, logical states correspond to physical states of the press as described by Table 4-1. The motor drive output is only modified on entry to the opening and closing states. At power up, the state machine is initialised to the opening state.

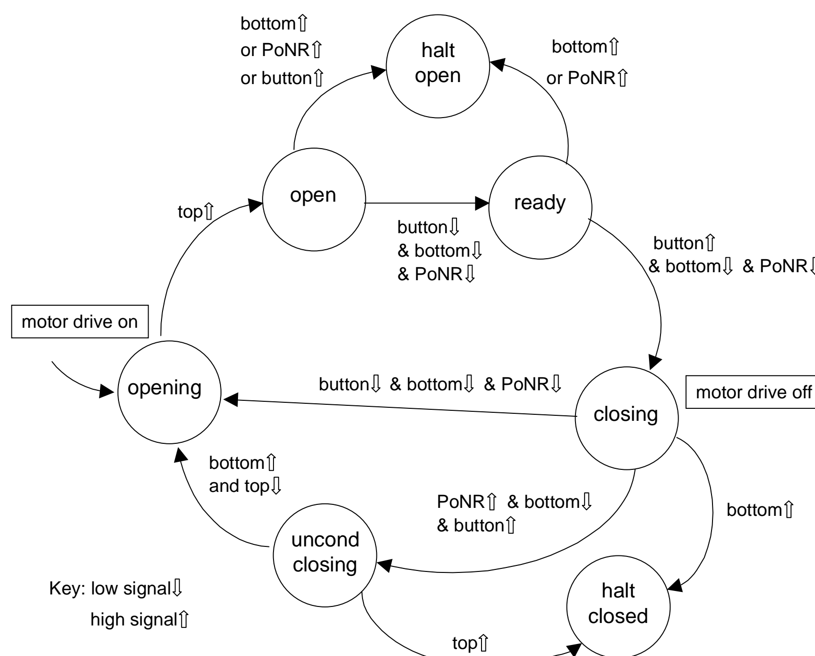

*Figure 3 - Press Control Logic*

| Logical State | Interpretation |
|---|---|
| Opening | Plunger is rising |
| Open | Plunger has reached top |
| ready | Plunger has reached top and button is not currently pressed |
| closing | Plunger is falling above PoNR |
| uncond closing | Plunger is falling below PoNR and will close unconditionally |
| halt open | Operation is halted with plunger at top |
| halt closed | Operation is halted with plunger on press bed |

*Table 4-1 Software states*

Note that the operator is required to release the push button before the plunger reaches the top, otherwise the press will halt open. We do not consider the procedure for restarting the press after fault detection here. Clearly there would need to be procedures for safely shutting down the press to allow repair. Similarly, the software would be augmented with facilities for reporting the nature of the failure detected, but these are not treated here.

Table 4-2 summarises the point of detection of critical sensor failures. It is assumed that the critical sensor failures are determined by a separate analysis. Some failures may also be detected at other points in the operation.

| Sensor failure mode | Point of detection |
|---|---|
| Bottom sensor stuck low | Not detected – non critical fault |
| Bottom sensor stuck high | Bottom sensor high signal received when plunger is closing above PoNR |
| Top sensor stuck low | Not detected – non critical fault |
| Top sensor stuck high | Top sensor high signal received when plunger is closing below PoNR |
| PoNR sensor stuck low | Bottom sensor high signal received before PoNR sensor low signal received when plunger is descending |
| PoNR sensor stuck high | PoNR sensor high signal received when plunger is at top of travel |
| Button sensor stuck low | Non detected – non critical fault |
| Button sensor stuck high | Button sensor high signal received as plunger reaches top of travel. |

*Table 4-2 - Detection of critical sensor failures*

## 4.2 Formal Design Specification

The software design is translated into Sum using three modules, as illustrated by Figure 4. The complete Sum specifications are in Appendix A. The formal specification is described below. The ASCII representation of Sum is translated into a more traditional Z style, interspersed with commentary. The specification is presented in a modular fashion with Sum modules delimited by an outer frame and schemas within separated by internal frames.

*Machine*

*Sensors* *Actuator*

*Figure 4 - Formal Software Specification Structure*

The *Sensors* and *Actuator* modules represent the input and output interfaces of the software by the types and values of the sensor and motor drive signals. They are an abstraction of the interfaces only and do not specify the interface protocol to be used. The following introduces a number of typed variables denoting Sensor and Actuator states. Types for the variables; *IN_SIG* and *OUT_SIG* are defined within the module by the set of potential variable values.

```
Sensors
    IN_SIG ::= high | low
    state
        button, top, PoNR, bottom: IN_SIG
```

```
Actuator
    OUT_SIG ::= a_on | a_off
    state
        motor: OUT_SIG
```

The control logic is specified by the *Machine* module. This module imports the interface modules in order to read and manipulate interface signal states. The type for the local *control* variable, which records the local control state, is introduced within the module.

```
Machine
    import Sensors
    import Actuator
    CONTROL ::= opening | open | ready | closing | uncond_closing |
                halt_open | halt_closed
    state
        control: CONTROL
        Sensors.state
        Actuator.state
```

Initialisation sets values for the control logic state and the motor drive signal, but not the sensor values.

*init*

$control' = opening$
$Actuator.motor' = Actuator.a_on$

State machine transitions are modelled by an *Op* schema for each machine state. Execution of the state machine is assumed to occur by invoking the operational schema with an active precondition.

An example operational schema follows. The schema concerns the transitions from the Closing state of the state machine, including the null transition which results in no change. Note that, since the *changes_only* expression cannot be embedded in a Sum if expression, the values of *control* and *Actuator.motor* are set on every branch.

*op At_Closing*

```
pre control = closing
if Sensors.bottom = Sensors.high
then
  (control' = halt_closed ∧ Actuator.motor' = Actuator.a_off)
else
  (if Sensors.PoNR = Sensors.high
  then
    (control' = uncond_closing ∧ Actuator.motor' = Actuator.a_off)
  else
    (if Sensors.button = Sensors.low
    then
      (control' = opening ∧ Actuator.motor' = Actuator.a_on)
    else
      (control' = closing ∧ Actuator.motor' = Actuator.a_off)
    fi)
  fi)
fi
changes_only{control, Actuator.motor}
```

For convenience, execution of the state machine is captured by a single operation. The result is deterministic since the preconditions of all operations are disjoint, as can be checked by inspection.

$$Transition == (At_Opening \vee At_Open \vee At_Ready \vee At_Closing \vee At_Uncond_Closing \vee At_Halt_Open \vee At_Halt_Closed)$$

The operational semantics of this specification are assumed to be that the *Transition* operation is repeatedly invoked until no further progress is possible. Sensor values may change arbitrarily between invocations.

## 4.3 Hardware Interface Specification

The software interacts with the sensors and motor drive via input and output registers. Sensor signals are mapped onto a single register at memory address 100001[1]. Its content is specified by Table 4-3.

[1] The underlying hardware and addresses of registers is imaginary and has been conceived for this paper.

| Sensor | Register Protocol |
|---|---|
| Top | 11xxxxxx: high signal<br>00xxxxxx: low signal |
| PoNR | xx11xxxx: high signal<br>xx00xxxx: low signal |
| Bottom | xxxx11xx: high signal<br>xxxx00xx: low signal |
| Button | xxxxxx11: high signal<br>xxxxxx00: low signal |

*Table 4-3 - Press System Sensor Signals*

The motor output register is located at memory address 100011. The motor drive interprets 11111111 as an ON signal and 00000000 as an OFF signal.

# 5 Software Design Safety Assurance

## 5.1 Strategy

This section describes our approach to verification that the software design satisfies the safety requirements described above. The approach is based on formal modelling of system states as they relate to safety, and exhaustive analysis of possible system behaviours using the Possum specification-animation tool [7].

In order to verify the safety requirements of the press physical system, the software state machine is animated within an (abstract) environment simulating the equipment under control. Different animations are used to explore the effect of software control on physical system behaviour under normal (fault-free) conditions and in the presence of single persistent sensor failures. Press operation and sensor failures are modelled as a finite state machine and formally specified in Sum.

Verification of system safety requirements is then performed by systematically searching all possible behaviours of the combined system using specification animation. The system states reached are automatically compared with the system safety requirements and unsafe operational scenarios are identified and analysed.

## 5.2 Abstract Model of Press Operation

To simulate the operation of the press we introduce a model comprising six possible physical states of the plunger (see Table 5-1), with transitions as illustrated in Figure 5. The simulation model abstracts away from timing properties such as exactly when transitions in the physical state occur. The interpretation of when transitions take place is as follows:

- ‘motor drive on’ means that the motor drive is applied for sufficiently long to achieve the indicated effect on the state of the plunger;
- ‘motor drive off’ means that the motor drive is off for sufficiently long to have the indicated effect;
- all other cases result in a null transition.

Note that the model is more liberal than reality, in the sense that it considers more states than may be physically possible.

In defining the press simulation model, a number of simplifying assumptions have been made about aspects of press operation, in particular:

1. There are no failures of the electromechanical plunger drive mechanism, and the motor drive has the desired effect on the plunger.
2. The press sensors are installed in their correct positions; in particular, the PoNR sensor is installed close to the true "point of no return".
3. The system only exhibits single, permanent sensor failures.
4. The controller operates according to the control logic design. Note however that no assumption is made about processor response time.

We expect that possible violations of these assumptions would be dealt with in other parts of the safety case, such as a consideration of possible hardware failures and a software timing analysis.

The position of the plunger in each state is used by the animator to determine what would be the corresponding press-sensor values under fault-free operation: e.g. in the at_bottom state, the bottom and PoNR sensor values would be high and the top sensor value would be low. To model persistent sensor failures, the simulation is extended by transitions corresponding to sensors failing and thereafter reporting a constant value.

A formal specification of the model is given in section 5.4

| Simulator state | Physical interpretation |
|---|---|
| at_bottom | Plunger is below bottom sensor |
| below_PoNR | Plunger is between bottom and PoNR sensors; continuous application of the motor drive will prevent press closing and will eventually raise plunger above PoNR sensor |
| falling_to_bottom | Similar to below_PoNR but motor drive cannot prevent press closing due to downward momentum |
| above_PoNR | Plunger is between PoNR and top sensor; continuous application of the motor drive will prevent plunger passing PoNR sensor and will eventually raise plunger above top sensor |
| falling_past_PoNR | Similar to above_PoNR but motor drive cannot prevent plunger passing PoNR sensor due to downward momentum |
| at_top | Plunger is above top sensor |

*Table 5-1 - Plunger states in press simulation*

## 5.3 Animation Design

In the animation, the press simulation model takes actuator values from the control software as inputs and assigns press sensor values as outputs. Together, the simulation model, the control software specification, and push and release of the button will emulate operation of the Press.

The animation consists of a systematic exploration of the possible physical behaviours of the press simulation model under software and operator control. The "system state" comprises all possible combinations of states of the system components (simulator, software and button). The primary system safety requirements from section 3 are formalised as properties of the system state and checked at each step of the animation. The secondary safety requirement – that single persistent sensor failures are detected and revealed – is demonstrated by showing that the press halts within one cycle of such a failure occurring.

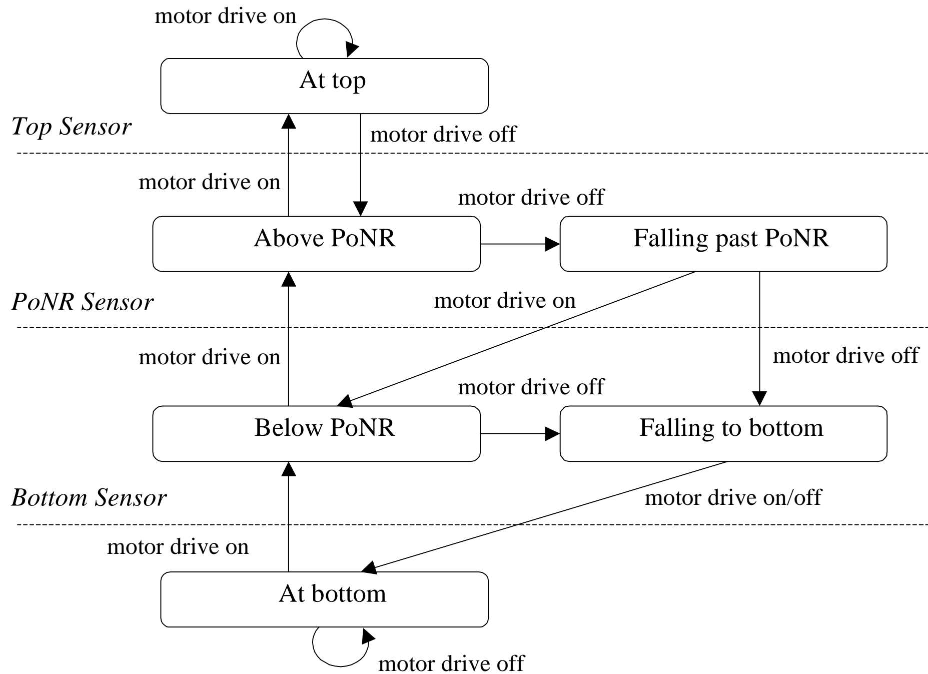


*Figure 5 - Press Simulation Model*

Separate animations are performed for fault-free operation and for each sensor failure mode. The animation is performed using a depth-first search of all reachable system states. The search is constructed from basic animation events as described in Table 5-2.

| Animation event | Interpretation |
|---|---|
| Init | Initialise simulation state (software in opening state, plunger at bottom, button released) |
| Control_Transition | Single (possibly null) transition of the software state machine without change to plunger state or sensor values |
| Button_Transition | Toggle of the button sensor value (push or release as appropriate), with no other changes |
| Plunger_Transition | Single (possibly null) transition of the plunger state and press sensor values, based on the current actuator state |
| Sensor_Failure | Activates the appropriate sensor failure mode (e.g. Bottom_Fail_High) |
| STOP | Terminate this run of the animation |

*Table 5-2 – Basic animation events*

The algorithms for animating the "normal operation" and "bottom sensor stuck high" scenarios are described in Figure 6 using a CCS-like process description language [8]. The algorithms keep track of what system states have been visited, and backtrack when an already-visited state is reached. By their exhaustive nature, the algorithms clearly will determine all possible system states reachable from the initial state.

The algorithms return a transition table for the complete system, as well as example runs (sequences of states corresponding to possible behaviours of the system) to aid in analysis.

*a)*

```
Normal -> Init ;P

P -> if state_already_visited then STOP
        else Q
Q-> Control_Transition ;P
         | Plunger_Transition ;P
         | Button_Transition ; P
```

*b)*

```
Bottom_Stuck_High -> Init ;P

P-> if state_already_visited then STOP
         else Q
Q-> Control_Transition ;P
         | Plunger_Transition ;P
         | Button_Transition ; P
         | Bottom_Fail_High ;R

R -> if state_already_visited then STOP
         else S

S -> Control_Transition ;R
         | Plunger_Transition ;R
         | Button_Transition ;R
```

*Figure 6 -Animation algorithm for a) normal operation b) bottom sensor fails high*

## 5.4 Animation Specification

The system for animation is specified by extending the existing Sum software specification with a new *Simulator* module through the shared import of interface states, as illustrated by Figure 7. The specification is explained in more detail below. The complete specification is given in Appendix B. <<Requires update!!>>

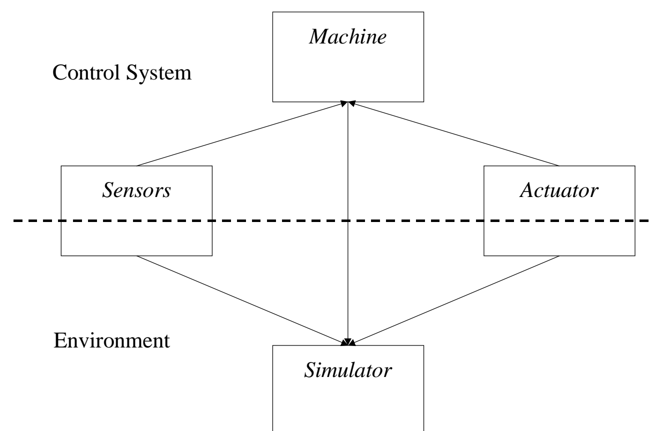


*Figure 7 - Simulation Environment*

### 5.4.1 Simulation State

The state of the *Simulation* module extends the machine state with a record of sensor health, physical state of the plunger and button, and an indication of safety. For convenience we define identifiers for useful sets of physical states and rename values of sensor variables.

```
Simulator
  // ------------------------------------------
  // Industrial Press Simulation Environment
  // The environment module provides a simulation environment for the Press
  // logic state machine. The simulation is performed at a physical level.
```

```
// The modelled physical state of the press is used to drive sensor signals.
// These, in turn, drive the control logic which then causes a physical
// state change. It is possible to activate sensor failures and
// investigate their effect.
// ----------------------------------------
import Sensors
import Actuator
import Machine
// -------------------------------
// State
// In addition to encapsulating the logic, sensor and motor drive states,
// the state represents the physical movement of the plunger.
// -------------------------------
SENSOR_HEALTH ::= broken | ok
PLUNGER ::= at_bottom | below_PoNR | above_PoNR | at_top
        falling_past_PoNR | falling_to_bottom
states_above_top == {at_top}
states_above_PoNR == {above_PoNR, at_top, falling_past_PoNR}
states_above_bottom == {s: PLUNGER | s ≠ at_bottom}
SAFETY ::= safe | abort_failed | unsafe_motor_drive
BUTTON ::= pressed | released
high == Sensors.high
low == Sensors.low

┌─state──────────────────────────────────────────────
│   Sensors.state
│   Actuator.state
│   Machine.state
│   plunger: PLUNGER
│   button: BUTTON
│   safety: SAFETY
│   button_health, top_health, PoNR_health, bottom_health: SENSOR_HEALTH
├────────────────────────────────────────────────────
```

The *Init* schema initialises the complete simulation state, including the value of sensors and physical states of the press plunger and push button. Note that no initial value is assigned to the safety indicator variable.

```
┌─init───────────────────────────────────────────────
│   Machine.init
│   Sensors.button' = Sensors.low
│   Sensors.top' = Sensors.low
│   Sensors.PoNR' = Sensors.high
│   Sensors.bottom' = Sensors.high
│   button_health' = ok
│   top_health' = ok
│   PoNR_health' = ok
│   bottom_health' = ok
│   plunger' = at_bottom
│   button' = released
└────────────────────────────────────────────────────
```

### 5.4.2 Evaluating Safety

Safety is automatically evaluated through animation by updating the *safety* variable in accordance with the hazardous states defined by Section 3. This is achieved by including the condition of safety within the *state* schema as a state invariant. The state invariant is then included as a condition within each operational schema, resulting in modification of the *safety* variable throughout the animation.

```
state
...
safety = if ((plunger = falling_to_bottom and
                (Actuator.motor = Actuator.a_on)) then
              unsafe_motor_drive
            else
              if ((plunger in states_above_PoNR) and
                  (Actuator.motor /= Actuator.a_on) and
                  (button = released)) then
                  abort_failed
              else
                  safe
              fi
            fi
```

### 5.4.3 Simulation Operations

Top level operations corresponding to the animation events of Table 5-2 are defined, corresponding to transitions of the simulator, the button and the control logic.

The *Control_Transition* simply renames the transition of the state machine.

```
op Control_Transition
    Machine.Transition
    changes_only{Machine.control, Actuator.motor, safety}
```

The physical simulation model described in Section 5.2 is captured by a set of valid transitions. These are recorded by a local function variable *next_plunger_state* defined as follows.

```
next_plunger_state: (PLUNGER × Actuator.OUT_SIG) → PLUNGER

next_plunger_state(at_bottom, Actuator.a_off) = at_bottom
next_plunger_state(at_bottom, Actuator.a_on) = below_PoNR
next_plunger_state(below_PoNR, Actuator.a_off) = falling_to_bottom
next_plunger_state(below_PoNR, Actuator.a_on) = above_PoNR
next_plunger_state(above_PoNR, Actuator.a_off) = falling_past_PoNR
next_plunger_state(above_PoNR, Actuator.a_on) = at_top
next_plunger_state(at_top, Actuator.a_off) = above_PoNR
next_plunger_state(at_top, Actuator.a_on) = at_top
next_plunger_state(falling_past_PoNR, Actuator.a_off) = falling_to_bottom
next_plunger_state(falling_past_PoNR, Actuator.a_on) = below_PoNR
next_plunger_state(falling_to_bottom, Actuator.a_off) = at_bottom
next_plunger_state(falling_to_bottom, Actuator.a_on) = at_bottom
```

A single operation is used to execute the physical movement of the plunger and update sensor values accordingly. Sensor values are only modified if the corresponding sensor is "healthy" (not failed stuck).

```
op Plunger_Transition
    plunger' = next_plunger_state(plunger, Actuator.motor)
    Sensors.top' =
        (if top_health = ok then
            (if (next_plunger_state (plunger, Actuator.motor) ∈
                states_above_top) then high else low fi )
        else
            Sensors.top
        fi )
    Sensors.PoNR' =
        (if PoNR_health = ok then
            if (next_plunger_state (plunger, Actuator.motor) ∈
                states_above_PoNR) then low else high fi )
        else
            Sensors.PoNR
        fi )
    Sensors.bottom' =
        (if bottom_health = ok then
            (if (next_plunger_state(plunger, Actuator.motor) ∈
                states_above_bottom) then low else high fi )
        else
            Sensors.bottom
        fi )
    changes_only{plunger, Sensors.top, Sensors.PoNR, Sensors.bottom, safety}
```

Operations are provided to push and release the button. The button signal is only modified if the button sensor is healthy. Only the example of pushing the button is shown here.

```
op Push_Transition
    Sensors.button' = (if (button_health = ok) then high else Sensors.button fi )
    button' = pressed
    changes_only{Sensors.button, button, safety}
```

Sensor failures are forced through a number of operations, one for each failure mode. The following specifies the bottom sensor failing with a permanent high signal.

```
op Bottom_Fail_High
    Sensors.bottom = high
    bottom_health' = broken
    changes_only{Sensors.bottom, bottom_health}
```

### 5.4.4 Algorithm Implementation

The search algorithm of Figure 6 is implemented by a Tcl program [9] integrated with the Possum tool to execute the appropriate sequence of operations. The program executes a traditional stack-based implementation of a depth-first search. During the search, a list of all visited states is maintained. In order to backtrack during the search, some auxiliary operations are provided to reset the system state.

The program produces a transcript of all runs explored, together with a summary table of reachable system states and corresponding system state transitions.

## 5.5 Animation Results

### 5.5.1 Presentation of Results

The animation results are generated separately for fault-free operation and for each permanent sensor failure mode. The number of visited states for each operational scenario is listed in Table 5-3. The states encountered in fault-free operation are also visited while exploring the effects of sensor failures.

| Animation mode | #states |
|---|---|
| Normal (fault-free) | 32 |
| Bottom Stuck Low | 64 |
| Bottom Stuck High | 74 |
| PoNR Stuck Low | 66 |
| PoNR Stuck High | 78 |
| Top Stuck Low | 64 |
| Top Stuck High | 88 |
| Button Stuck Low | 64 |
| Button Stuck High | 64 |
| Total (unique) | 338 |

*Table 5-3 - Number of visited states*

For each mode, all generated paths are recorded and the results are summarised in a transition table. Example results for fault-free operation and for the bottom sensor high failure are presented in Sections 5.5.2 and 5.5.3. A summary of all animation results is presented in Appendix C. Some abbreviations have been used to capture the states but their meaning should be evident.

### 5.5.2 Fault-Free Operation

A transition table summarising all behaviours under fault-free operation is presented in Figure 8. There are three hazardous states that occur when the button is initially released while the plunger is falling above the PoNR. However, these states are transitory and are safety is immediately restored upon next execution of the software. Unless the software halts or is extremely slow, no accidents can arise.

```
=========================================================
Model check for Industrial Press Control System
=========================================================
Number of states = 32
---------------------------------------------------------------------------
    plunger    control     button    motor    safe      C.T  B.T P.T
---------------------------------------------------------------------------
1   at_bottom  opening     released  a_on     safe      1    2    4
2   at_bottom  opening     pressed   a_on     safe      2    1    3
3   below_PoNR opening     pressed   a_on     safe      3    4    6
4   below_PoNR opening     released  a_on     safe      4    3    5
5   above_PoNR opening     released  a_on     safe      5    6    16
6   above_PoNR opening     pressed   a_on     safe      6    5    7
7   at_top     opening     pressed   a_on     safe      8    16   7
8   at_top     open        pressed   a_on     safe      9    11   8
9   at_top     halt_open   pressed   a_on     safe      9    10   9
10  at_top     halt_open   released  a_on     safe      10   9    10
11  at_top     open        released  a_on     safe      12   8    11
12  at_top     ready       released  a_on     safe      12   13   12
```

```
13  at_top     ready       pressed   a_on     safe      14   12   13
14  at_top     closing     pressed   a_off    safe      14   15   18
15  at_top     closing     released  a_off    no_abort  16   14   17
16  at_top     opening     released  a_on     safe      11   7    16
17  above_PoNR closing     released  a_off    no_abort  5    18   20
18  above_PoNR closing     pressed   a_off    safe      18   17   19
19  past_PoNR  closing     pressed   a_off    safe      19   20   28
20  past_PoNR  closing     released  a_off    no_abort  21   19   23
21  past_PoNR  opening     released  a_on     safe      21   22   4
22  past_PoNR  opening     pressed   a_on     safe      22   21   3
23  to_bottom  closing     released  a_off    safe      24   28   32
24  to_bottom  uncond_cls  released  a_off    safe      24   25   27
25  to_bottom  uncond_cls  pressed   a_off    safe      25   24   26
26  at_bottom  uncond_cls  pressed   a_off    safe      2    27   26
27  at_bottom  uncond_cls  released  a_off    safe      1    26   27
28  to_bottom  closing     pressed   a_off    safe      25   23   29
29  at_bottom  closing     pressed   a_off    safe      30   32   29
30  at_bottom  halt_closed pressed   a_off    safe      30   31   30
31  at_bottom  halt_closed released  a_off    safe      31   30   31
32  at_bottom  closing     released  a_off    safe      31   29   32
---------------------------------------------------------------------------
```

*Figure 8 – States visited under Fault-free operation*

### 5.5.3 Bottom Stuck High Failure Operation

Figure 9 illustrates the list of states visited under a permanent bottom high sensor failure. Again there are temporary hazardous states that are controlled by immediate software execution. Other hazardous states of longer duration are highlighted.

```
=========================================================
Model check for Industrial Press Control System Bottom_Fail_High
=========================================================
Number of states 74
------------------------------------------------------------------------------------
    plunger    control     button    motor    safe      bottom  C.T  B.T  P.T  F.T
------------------------------------------------------------------------------------
1   at_bottom  opening     released  a_on     safe      ok      1    2    4    41
2   at_bottom  opening     pressed   a_on     safe      ok      2    1    3    42
3   below_PoNR opening     pressed   a_on     safe      ok      3    4    6    32
4   below_PoNR opening     released  a_on     safe      ok      4    3    5    31
5   above_PoNR opening     released  a_on     safe      ok      5    6    18   34
6   above_PoNR opening     pressed   a_on     safe      ok      6    5    7    33
7   at_top     opening     pressed   a_on     safe      ok      8    18   7    22
8   at_top     open        pressed   a_on     safe      ok      9    13   8    21
9   at_top     halt_open   pressed   a_on     safe      ok      9    10   9    12
10  at_top     halt_open   released  a_on     safe      ok      10   9    10   11
11  at_top     halt_open   released  a_on     safe      broken  11   12   11   11
12  at_top     halt_open   pressed   a_on     safe      broken  12   11   12   12
13  at_top     open        released  a_on     safe      ok      14   8    13   20
14  at_top     ready       released  a_on     safe      ok      14   15   14   74
15  at_top     ready       pressed   a_on     safe      ok      16   14   15   73
16  at_top     closing     pressed   a_off    safe      ok      16   17   24   72
17  at_top     closing     released  a_off    no_abort  ok      18   16   23   69
18  at_top     opening     released  a_on     safe      ok      13   7    18   19
19  at_top     opening     released  a_on     safe      broken  20   22   19   19
20  at_top     open        released  a_on     safe      broken  11   21   20   20
21  at_top     open        pressed   a_on     safe      broken  12   20   21   21
22  at_top     opening     pressed   a_on     safe      broken  21   19   22   22
23  above_PoNR closing     released  a_off    no_abort  ok      5    24   26   68
24  above_PoNR closing     pressed   a_off    safe      ok      24   23   25   65
25  past_PoNR  closing     pressed   a_off    safe      ok      25   26   48   64
26  past_PoNR  closing     released  a_off    no_abort  ok      27   25   35   61
27  past_PoNR  opening     released  a_on     safe      ok      27   28   4    30
28  past_PoNR  opening     pressed   a_on     safe      ok      28   27   3    29
29  past_PoNR  opening     pressed   a_on     safe      broken  29   30   32   29
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 30 | past_PoNR | opening | released | a_on | safe | broken | 30 | 29 | 31 | 30 |
| 31 | below_PoNR | opening | released | a_on | safe | broken | 31 | 32 | 34 | 31 |
| 32 | below_PoNR | opening | pressed | a_on | safe | broken | 32 | 31 | 33 | 32 |
| 33 | above_PoNR | opening | pressed | a_on | safe | broken | 33 | 34 | 22 | 33 |
| 34 | above_PoNR | opening | released | a_on | safe | broken | 34 | 33 | 19 | 34 |
| 35 | to_bottom | closing | released | a_off | safe | ok | 36 | 48 | 54 | 60 |
| 36 | to_bottom | uncond_cls | released | a_off | safe | ok | 36 | 37 | 39 | 47 |
| 37 | to_bottom | uncond_cls | pressed | a_off | safe | ok | 37 | 36 | 38 | 44 |
| 38 | at_bottom | uncond_cls | pressed | a_off | safe | ok | 2 | 39 | 38 | 43 |
| 39 | at_bottom | uncond_cls | released | a_off | safe | ok | 1 | 38 | 39 | 40 |
| 40 | at_bottom | uncond_cls | released | a_off | safe | broken | 41 | 43 | 40 | 40 |
| 41 | at_bottom | opening | released | a_on | safe | broken | 41 | 42 | 31 | 41 |
| 42 | at_bottom | opening | pressed | a_on | safe | broken | 42 | 41 | 32 | 42 |
| 43 | at_bottom | uncond_cls | pressed | a_off | safe | broken | 42 | 40 | 43 | 43 |
| 44 | to_bottom | uncond_cls | pressed | a_off | safe | broken | 45 | 47 | 43 | 44 |
| 45 | to_bottom | opening | pressed | a_on | bad_drv | broken | 45 | 46 | 42 | 45 |
| 46 | to_bottom | opening | released | a_on | bad_drv | broken | 46 | 45 | 41 | 46 |
| 47 | to_bottom | uncond_cls | released | a_off | safe | broken | 46 | 44 | 40 | 47 |
| 48 | to_bottom | closing | pressed | a_off | safe | ok | 37 | 35 | 49 | 57 |
| 49 | at_bottom | closing | pressed | a_off | safe | ok | 50 | 54 | 49 | 56 |
| 50 | at_bottom | halt_closed | pressed | a_off | safe | ok | 50 | 51 | 50 | 53 |
| 51 | at_bottom | halt_closed | released | a_off | safe | ok | 51 | 50 | 51 | 52 |
| 52 | at_bottom | halt_closed | released | a_off | safe | broken | 52 | 53 | 52 | 52 |
| 53 | at_bottom | halt_closed | pressed | a_off | safe | broken | 53 | 52 | 53 | 53 |
| 54 | at_bottom | closing | released | a_off | safe | ok | 51 | 49 | 54 | 55 |
| 55 | at_bottom | closing | released | a_off | safe | broken | 52 | 56 | 55 | 55 |
| 56 | at_bottom | closing | pressed | a_off | safe | broken | 53 | 55 | 56 | 56 |
| 57 | to_bottom | closing | pressed | a_off | safe | broken | 58 | 60 | 56 | 57 |
| 58 | to_bottom | halt_closed | pressed | a_off | safe | broken | 58 | 59 | 53 | 58 |
| 59 | to_bottom | halt_closed | released | a_off | safe | broken | 59 | 58 | 52 | 59 |
| 60 | to_bottom | closing | released | a_off | safe | broken | 59 | 57 | 55 | 60 |
| 61 | past_PoNR | closing | released | a_off | no_abort | broken | 62 | 64 | 60 | 61 |
| 62 | past_PoNR | halt_closed | released | a_off | no_abort | broken | 62 | 63 | 59 | 62 |
| 63 | past_PoNR | halt_closed | pressed | a_off | safe | broken | 63 | 62 | 58 | 63 |
| 64 | past_PoNR | closing | pressed | a_off | safe | broken | 63 | 61 | 57 | 64 |
| 65 | above_PoNR | closing | pressed | a_off | safe | broken | 66 | 68 | 64 | 65 |
| 66 | above_PoNR | halt_closed | pressed | a_off | safe | broken | 66 | 67 | 63 | 66 |
| 67 | above_PoNR | halt_closed | released | a_off | no_abort | broken | 67 | 66 | 62 | 67 |
| 68 | above_PoNR | closing | released | a_off | no_abort | broken | 67 | 65 | 61 | 68 |
| 69 | at_top | closing | released | a_off | no_abort | broken | 70 | 72 | 68 | 69 |
| 70 | at_top | halt_closed | released | a_off | no_abort | broken | 70 | 71 | 67 | 70 |
| 71 | at_top | halt_closed | pressed | a_off | safe | broken | 71 | 70 | 66 | 71 |
| 72 | at_top | closing | pressed | a_off | safe | broken | 71 | 69 | 65 | 72 |
| 73 | at_top | ready | pressed | a_on | safe | broken | 12 | 74 | 73 | 73 |
| 74 | at_top | ready | released | a_on | safe | broken | 11 | 73 | 74 | 74 |

*Figure 9 - States visited with bottom sensor stuck high*

Figure 10 illustrates one of the unsafe scenarios produced by the animation. In this scenario, the bottom sensor fails high while the plunger is falling above the PoNR. As a result, the software fails to abort press closure when the button is released and the plunger falls uncontrollably to the press bottom. The sensor failure is detected when the plunger returns to the top of travel, and the press operation is halted.

| | plunger | control | button | motor | safe | bottom |
|---|---|---|---|---|---|---|
| 1 | at_bottom | opening | released | a_on | safe | ok |
| 2 | below_PoNR | opening | released | a_on | safe | ok |
| 3 | above_PoNR | opening | released | a_on | safe | ok |
| 4 | at_top | opening | released | a_on | safe | ok |
| 5 | at_top | open | released | a_on | safe | ok |
| 6 | at_top | ready | released | a_on | safe | ok |
| 7 | at_top | ready | pressed | a_on | safe | ok |
| 8 | at_top | closing | pressed | a_off | safe | ok |
| 9 | above_PoNR | closing | pressed | a_off | safe | ok |

```
10  above_PoNR closing     pressed   a_off    safe      broken
11  above_PoNR uncond_cls  pressed   a_off    safe      broken
12  above_PoNR uncond_cls  pressed   a_off    safe      broken
13  above_PoNR uncond_cls  released  a_off    no_abort  broken
14  past_PoNR  uncond_cls  released  a_off    no_abort  broken
15  to_bottom  uncond_cls  released  a_off    safe      broken
16  at_bottom  uncond_cls  released  a_off    safe      broken
17  at_bottom  opening     released  a_on     safe      broken
18  at_bottom  opening     released  a_on     safe      broken
19  below_PoNR opening     released  a_on     safe      broken
20  above_PoNR opening     released  a_on     safe      broken
21  at_top     opening     released  a_on     safe      broken
22  at_top     halt_open   released  a_on     safe      broken
------------------------------------------------------------------------------------
##  Path complete: cycle or halted
------------------------------------------------------------------------------------
```

*Figure 10 - An example hazardous scenario*

This scenario is typical of the remaining hazardous states of press operation. Section 5.5.4 presents a more complete analysis of the results.

### 5.5.4 Analysis of Results

Inspection of the animation output reveals that all persistent failure modes are detected and handled within one operational cycle. Despite this, 22 unsafe states are still encountered that are not immediately rectified by the control logic. Such states require further analysis to determine acceptability, as follows.

1. The bottom sensor fails high while the plunger is below PoNR and falling to bottom. In this case the motor drive will be unsafely activated before the plunger reaches the bottom of the press. The sensor fault will then be detected when the plunger reaches top of the travel, and the press will halt open. An argument for accepting this risk would probably be based on the low likelihood that the sensor would fail within this small window of opportunity.
2. The bottom sensor fails high while the plunger is falling above the PoNR. The fault will be detected immediately and the press will halt closed; in the meantime however the abort facility will be lost for the rest of that cycle. An argument for accepting this risk would probably be based on the low likelihood that the abort facility would be required in the small window of opportunity.
3. The PoNR sensor fails low at almost any point in the cycle, the press closes under normal operation, the plunger falls past the PoNR, and the operator then erroneously releases the button before the plunger has reached the bottom. As a result, the software, thinking the plunger has not yet passed the PoNR, will try to abort operation by activating the motor drive (unsafely, as it turns out). The press will close and then immediately start re-opening. The sensor fault will be detected the next time the plunger reaches the bottom under normal operation, and the press will halt closed. However, it is possible that this fault would go undetected for several operations if the operator continues to release the button before the plunger reaches the bottom. This can be prevented by operational procedures to maintain button pressure until the plunger begins to rise.
4. The PoNR sensor fails high while the plunger is falling above the PoNR. Again, the abort facility will be lost temporarily, but the fault will be detected when the plunger reaches the bottom and the press will halt closed.
5. An unusual (and extremely unlikely) scenario occurs when the PoNR sensor fails high after the plunger begins descent but prior to passing the top sensor. In this case, the software may cycle through to the uncond_closing state where an assumed failure of the top sensor is detected and operation is halted. The plunger is allowed to fall and descent cannot be aborted. The very small window of opportunity for this scenario would render it acceptable.
6. The button sensor fails high while the press is fully open or closing, resulting in the loss of the abort facility. The sensor fault will be detected when the plunger next reaches top of travel. The

risk of this failure may be reduced by an additional form of protection, for example a beam to detect human presence in the press vicinity.

# 6 Implementation

This section describes the implementation of the Press software using the SPARK restricted subset of Ada and its associated toolset [2]. Annotations inserted to assist code verification are included but not discussed until section 7. The complete code listing is in Appendix D.

## 6.1 SPARK background

SPARK is a subset of Ada that excludes many unsafe features. The SPARK kernel is well- defined, easy to understand, yet suited to programming in the large. Ada features that are hard to specify or inappropriate in a high-integrity context are excluded.

As well as the Ada subset, SPARK contains two layers of annotation, or formal comment. Annotations constrain the Ada semantics to enable static flow analysis and proof against specification.

Static analysis is a mandatory aspect of SPARK Ada development that provides a rigorous sanity check on the static structure of the code, over and above normal type-checking. Data flow analysis checks the direction of data flow: that variables are written before they are read. Information flow analysis further checks dependencies between variables: that an output depends only on a specified set of inputs. In SPARK these dependencies between variables are 'declared' as part of a procedure specification through (mandatory) **global** and **derives** annotations[2]. The SPARK Examiner tool performs flow analysis using these annotations.

The second, optional, layer of annotation (**pre**, **post**, **assert** and **check**) state conditions that must hold on the program at different points of its execution. For example, **pre** states a subprogram's preconditions and **post** its postconditions; together they act as specification of the subprogram's behaviour. The conditions are expressed in a predicate logic called FDL that relates Ada program objects. The SPARK Examiner generates a collection of verification conditions (VCs - also known as proof obligations) on the SPARK program using the optional annotations. VCs are generated by tracing backwards over the program flow graph from an asserted final state to an initial state.

VCs must be discharged to prove that the program meets its specification. For example, a procedure body must achieve the post-condition on its procedure specification and a loop assertion must be valid at every iteration. The SPARK Simplifier tool will normally discharge each VC automatically. Otherwise, the more powerful interactive SPARK Proof Checker is needed.

The original SPARK source is submitted to a normal Ada compiler for translation to machine code. If the SPARK analysis reveals no errors then the compilation should succeed. Some Ada programs cannot be handled by the SPARK toolkit, in particular those containing low-level and IO code; these must be validated separately.

## 6.2 Implementation description

### 6.2.1 Program units

Ada and SPARK code is produced manually by translating from the formal Sum specification. Code production is also informed by an understanding of the informal transition diagram.

The Ada architecture broadly mirrors the Sum specification. There are however notable differences owing to the way the two languages handle state. In Sum, state and behaviour are present in each schema definition, and the machine's total state and behaviour is derived via Sum's semantic rules.

[2] Strictly, data flow analysis (the **global** annotations) is mandatory and information flow analysis (the **derives** annotations) is optional, but we regard both forms as essential in a rigorous development.

Ada is an imperative programming language and state and behaviour must be programmed explicitly. This leads to a slightly different module structure, with four Ada program units.

package Sensors - defines sensor values and read operation
package Actuator - defines actuator values and write operation
package Transitions - defines machine states, initial state, and state transitions
procedure Machine – declares state variables and executes main control loop

Figure 7 illustrates the Ada program structure, with arrows indicating unit dependencies. Units not subject to SPARK analysis are shaded grey. Comparison with Figure 4 shows how Ada differs from the Sum. The most obvious change is that the specification module *Machine* is implemented by the main procedure Machine and the supporting Transitions package. Clearly, there is no implementation of the simulation environment.

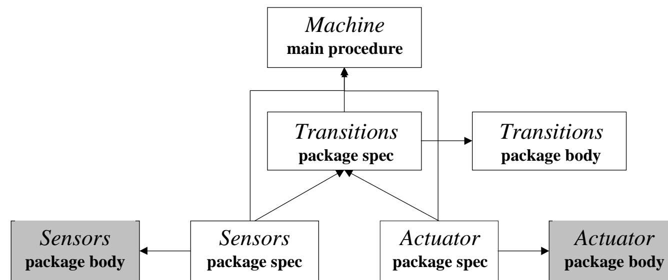


*Figure 7 - Software Implementation Structure*

The translation of objects and functions is relatively intuitive, except that some name modifications are made to improve coding style and to enable the SPARK toolset to verify the code. Additional code is provided to implement the implicit operational semantics.

The other packages in the hierarchy declare static types and subprograms, which animate the machine when invoked. The Sensors and Actuator packages correspond to the Sum interface modules with the same names. They contain type definitions and access methods to hardware devices. The bodies of these packages access device registers directly and are not formally analysed; they are shaded grey on the diagram.

The Sum *init* and *op* schemas become procedures in a new package called Transitions with no direct access to state. This corresponds to input and output schema variables, an alternative specification style, and eases verification.

For reasons of programming style and convenience, and for ease of analysis, state and behaviour are declared in the top-level procedure, Machine. Machine repeatedly executes transitions on the state until a halt state is reached, thereby implementing the implicit operation of the Sum machine.

Specific elements of the program are discussed below. For ease of reference, each program segment is labelled with the host program unit.

### 6.2.2 Type and variable declarations

Translation of Sum types into Ada types and variables is straightforward, with the exception that slightly different approaches were taken in aggregation of type declarations.

Sensor and actuator signal types translate exactly to enumerated types; however there is an extra type that captures the complete sensor state. This is done to simplify procedural access to the state.

```
type In_Sig is (high, low);
```

```
type State is record
    top     : In_Sig;
    PoNR : In_Sig;
    bottom : In_Sig;
    button : In_Sig;
end record; ...................................................................................... package Sensors

type Out_Sig is (a_on, a_off); ........................................................................... package Actuator
```

The definition of control logic states translates naturally into an Ada enumeration type.

```
type Control_Type is (opening, open, ready, closing, uncond_closing, halt_open, halt_closed);
    package Transitions
```

The program state is declared centrally in main procedure Machine, as three variables. This corresponds to the total state of the Sum modules.

```
control : Transitions.Control_Type;
sensors_state : Sensors.State;
motor : Actuator.Out_Sig; ..................................................................... procedure Machine
```

### 6.2.3 External device control

The Sum specification does not capture external device control but assumes that device state is directly accessible. The Ada code abstracts from device control in a similar way through access procedures that read the sensor values and switch the motor drive.

```
procedure Read (Value: out State);
  --# global State_Seq;
  --# derives Value, State_Seq from State_Seq; ......................................... package Sensors

procedure Write (Value: in Out_Sig);
  --# global State_Seq;
  --# derives State_Seq from Value, State_Seq; ................................ package Actuator
```

The **global** and **derives** annotations, together with the procedure parameter modes, specify data and information flow. The physical hardware devices access mapped memory attached to the host processor directly and asynchronously, as defined in Section 2.2. State_Seq is an imaginary variable that represents the sequence of values of mapped memory. Read gets a new value from the devices, which it also updates. Write sends a new value to the devices, and also depends on their current value.

The bodies of the Read and Write procedures directly address this memory via a low-level Ada representation of the address space. The implementation is inherently hardware-specific; it cannot be analysed by SPARK and must be verified by inspection and test. Specimen implementations are presented in Appendices D.2 and D.4.

Note that the flow annotations specify a general behaviour for all device readers and writers, whatever implementation is chosen.

The specification in Section 2.2 requires the press to halt if one of the sensor registers contains an undefined value ‘10’ or ‘01’. If the sensor Read procedure detects such a value it returns a Sensors_State where all values are stuck ‘high’ permanently. The press should then halt within one normal operational cycle.

### 6.2.4 Initialisation

An Ada procedure specification is written corresponding to the SUM initialisation schema.

```
procedure Init          (control  : out Control_Type;
                         motor    : out Actuator.Out_Sig);
  --# derives control, motor from ;
```

```
--# post(control = opening and motor = a_on); ................................ package Transitions
```

This procedure assigns initial values to the state machine and the motor drive at system start-up. The **derives** annotation indicates that the output values of both control and motor do not depend on anything, i.e. that the function is a true initialisation. The post-condition specifies what the initial values are. It is copied from the SUM.

The implementation is trivial and listed in Appendix D.4.

### 6.2.5 State transitions

A SPARK procedure is defined for each state transition specified by a SUM operation schema. Each procedure has three parameters: the set of sensors, the current state, and the motor effector. It also has SPARK information flow annotations and pre and post-conditions translated from the SUM specification. Here is the procedure specification corresponding to the *At_Closing* schema.

```
procedure At_Closing    (sensors_state: in Sensors.State;
                         control   : in out Control_Type;
                         motor     : out Actuator.Out_Sig);
  --# derives control from control, sensors_state
  --#        & motor from sensors_state;

--# pre        control = closing;
  --# post     (sensors_state.bottom = high
  --#              -> (control = halt_closed and motor = a_off))
  --#      and ((not (sensors_state.bottom = high)
  --#           and (sensors_state.PoNR = high))
  --#              -> (control = uncond_closing and motor = a_off))
  --#      and ((not (sensors_state.bottom = high or
  --#             sensors_state.PoNR = high)
  --#            and (sensors_state.button /= high))
  --#              -> (control = opening and motor = a_on))
  --#      and (not (sensors_state.bottom = high or
  --#                 sensors_state.PoNR = high or
  --#                sensors_state.button /=high)
  --#        -> (control = closing and motor = a_off)); ................................ package Transitions
```

The modes of the three parameters and their information flow derivations correspond directly to their roles in the state machine.

- sensors_state is pure input, and therefore a non-changing **in** parameter.
- control may change during the procedure depending on the old state and sensor values, so it is an **in out** parameter with accompanying derivation rules.
- motor is specified as an **out** parameter whose value is derived from the sensor input values. This information flow rule follows from the formal Sum specification, which assigns a value to the motor on every transition, regardless of its previous state. Informal analysis of the system indicates that the motor is known to be in state a_off on entry (and is likewise known on entry to every other state transition), but this fact is not stated by the Sum and is accordingly ignored.

The pre- and postconditions specify what the function must achieve. They are obtained by manual transliteration of Sum into FDL. The translation is fairly immediate up to substitution of variable names. The one difference is that FDL does not have an if ... then ... else ... form. SUM rules of the form if A then B else C fi are translated to the semantically equivalent (A -> B) and (not A -> C). The form is also provably equivalent to (A and B) or (not A and C).

The bodies of the transition functions are straightforward. There are no further annotations. The body of At_Closing is given here.

```
procedure At_Closing        (sensors_state: in Sensors.State;
```

```
                         control   : in out Control_Type;
                         motor     : out Actuator.Out_Sig) is
  begin
    if sensors_state.bottom = high then
      control := halt_closed;
      motor := a_off;
    elsif sensors_state.PoNR = high then
      control := uncond_closing;
      motor := a_off;
    elsif sensors_state.button /=high then
      control := opening;
      motor := a_on;
    else
      motor := a_off;
    end if;
end At_Closing; ..................................... package body Transitions
```

The At_Closing example illustrates how flow annotations and code interact. Parameter control is assigned a new value if the condition on the sensors holds. Therefore it depends on the sensors. Otherwise, it does not change, and its value depends on its own initial value. Hence the derivation control **from** control, sensors_state is respected. The Motor variable is assigned a different value according to the sensors' values. Therefore the derivation motor **from** sensors_state is respected. Note that motor is assigned a value inside each conditional case, which is a necessary condition on a parameter of mode **out**.

It is visually apparent that the body satisfies the specification. A discussion of how this fact is proven follows in sections 7.1.and 7.2.

### 6.2.6 Alternative programming styles

The Ada procedures could have been written in other ways and still reflect the Sum. Programmer choice is informed by good Ada style, and by the information flow rules, which place tight restrictions on how variables are used.

One might make use of the observed, but unspecified, fact that the motor parameter is known on entry to each transition. With this information, its parameter mode could be changed to **in out**, information flow rules simplified, and redundant assignments removed. Some transitions never switch the motor drive, for example At_Opening. In these cases motor could be omitted as a parameter altogether. However, it is considered better coding style to make all procedure signatures equivalent. It is also unwise to rely on assumptions not explicit in the formal specification without revisiting that specification.

The parameter control always takes the same value on entry; in our example it is closing. One alternative is to make control an **out** parameter, change its **derives** statement to control **from** sensors_state and omit the precondition. However, if were this was done, an assignment to the parameter must be performed inside the **else** clause in the body, because an **out** parameter must always be assigned on exit.

It happens that At_Closing and other state transitions are called at only one place in the program, inside the main control loop (see 6.2.7), and their actual parameters are the state variables mentioned in Section 6.2.2. An alternative programming style would make the transition procedures parameterless, and access the variables as globals. SPARK makes all global data use visible, so there would be no cost to review and analysis. The style adopted here has the advantage of simplifying package dependencies and easing long-term maintenance.

The information flow rules are probably the hardest part of SPARK to program correctly, and very sensitive to slight changes in design, as indicated above. This sensitivity should be expected. Information flow analysis is a very stringent check that variables and parameters are only used in the way that the designer intended. In practise, it revealed many flaws in the program code during early development. Once information flow had been established, formal proof followed without difficulty.

### 6.2.7 Main program

The main Ada program first declares and initialises the state variables, and switches the motor accordingly.

```
Transitions.Init (control,motor);
Actuator.Write (motor); .......................................................................... procedure Machine
```

It then enters a loop that emulates the assumed operation of the Sum specification and informal state machine, by repeatedly activating transitions where preconditions allow. Specifically, the loop does three things at each iteration: it reads the current value of the sensors; it executes the transition procedure for the current state; and it switches the motor drive on or off as appropriate.

The loop only terminates if the state machine enters one of two terminal states, halt_open and halt_closed.

```
while (control /= Transitions.halt_open)
    and then (control /= Transitions.halt_closed)
  --# assert true;        -- SPARK demands a loop invariant

  loop
    Press_Sensors.Read (sensors_state);

    case control is
    when Transitions.opening =>
       Transitions.At_Opening (sensors_state, control, motor);
    when Transitions.open =>
       Transitions.At_Open (sensors_state, control, motor);
    when Transitions.ready =>
       Transitions.At_Ready (sensors_state, control, motor);
    when Transitions.closing =>
       Transitions.At_Closing (sensors_state, control, motor);
    when Transitions.uncond_closing =>
       Transitions.At_Uncond_Closing (sensors_state, control, motor);
    when Transitions.halt_open | Transitions.halt_closed =>
       null;              --# check false;     -- means that this path can never be taken
    end case;

    Press_Actuator.Write (motor);
end loop; ....................................................................................... procedure Machine
```

The assertion at the start of the loop is a requirement of SPARK, and states an invariant that must hold on every iteration. Because the state machine is deterministic, exactly one operation is valid at each repetition and True is a sufficient invariant.

The last limb of the case clause is necessary according to the rules of Ada but is never executed. This fact is asserted by the check False annotation, which means that paths leading to that limb must yield falsehood; i.e. no such path can exist.

The final step of the main program is to assert that it terminates only in a halt_open or halt_closed state.

```
--# assert (control = Transitions.halt_open) or (control = Transitions.halt_closed) ; procedure
   Machine
```

# 7 Implementation Safety Assurance

## 7.1 Generation of verification conditions

The SPARK Examiner tool generates verification conditions (VCs) for each procedure it examines. VCs are derived from the executable code and optional proof annotations, and state theorems that must be satisfied in order to claim that the code meets its specification, as stated by the proof annotations.

They are obtained by walking backwards over the program flow graph, from a fixed point given by one annotation to an earlier annotation, symbolically undoing assignments on the way (see [2] for more information). VCs are the relationships between variables remaining at the end of this process. They are expressed in FDL and are typically verbose, difficult to read, and their derivation from the SPARK code is unintuitive.

The SPARK press software contains proof annotations as pre- and post-conditions on each transition procedure, and a loop invariant and termination condition on the main control loop. There are therefore VCs pertaining to these parts of the program. Altogether there are 19 VCs generated by initialisation and transition procedures and 15 by the main procedure, including the control loop.

As an example, four verification conditions are generated for procedure At_Closing and recorded in file `at_closi.vcg`. The compete listing is shown in Appendix E but one verification condition is shown here.

```
procedure_at_closing_1.
H1:    control = closing .
H2:    fld_bottom(sensors_state) = high .
        ->
C1:    (fld_bottom(sensors_state) = high) -> ((halt_closed =
           halt_closed) and (a_off = a_off)) .
C2:    ((not (fld_bottom(sensors_state) = high)) and (fld_ponr(
           sensors_state) = high)) -> ((halt_closed =
           uncond_closing) and (a_off = a_off)) .
C3:    ((not ((fld_bottom(sensors_state) = high) or (fld_ponr(
           sensors_state) = high))) and (fld_button(
           sensors_state) <> high)) -> ((halt_closed =
           opening) and (a_off = a_on)) .
C4:    (not ((fld_bottom(sensors_state) = high) or ((fld_ponr(
           sensors_state) = high) or (fld_button(
           sensors_state) <> high)))) -> ((halt_closed =
           closing) and (a_off = a_off)) .
```

The FDL notation, particularly record field selection, is somewhat clumsy and obscure. In fact, the verification conditions are trivial, and can be discharged using name equivalence and modus ponens. The body of At_Closing achieves its specification in a straightforward, obvious manner, so one might expect the VCs to be simple.

There are four VCs because the postcondition on the procedure has four conjuncts. By comparing with the SPARK specification it will be observed that the hypotheses of the VCs are derived from the precondition together with the antecedents of the postcondition. The conjectures of the VCs are calculated by assuming the conclusions of the postconditions and walking backwards through the procedure body.

## 7.2 Discharge of verification conditions

The SPARK Simplifier generates two sets of log files: a record of the proofs performed (*.slg); and a summary of proven theorems (*.siv). Inspection of the files shows that all 34 VCs are discharged.

A proof of `procedure_at_closing_1`, taken from file `at_closi.slg` is shown in Appendix E.

As expected, the VC is discharged automatically by simplifying equivalences in the conjecture and by showing contradictory hypotheses. Other VCs for other procedures are longer, but do not require logic that is any more complex.

The summary of proven theorems for procedure at_closing, taken from file `at_closi.siv` is shown in Appendix E.

Discharge of all VCs proves formally that the SPARK code satisfies its specification, as stated by the proof annotations. Because the annotations were translated directly from Sum, it also proves rigorously that the SPARK satisfies the Sum specification.

# 8 Discussion

## 8.1 Formal Methods and Design Safety Analysis

There are a number of ways that formal methods can be used to support safety analysis. In addition to providing higher levels of rigour, formal methods can improve the efficiency of the analysis task by enabling various levels of automation. In keeping with our aim of addressing lightweight assurance methods, we discuss the use of formal methods for animation, static analysis, model checking and hazard analysis. Assurance by theorem proving is not considered.

### 8.1.1 Animation

Animation and simulation of formal specifications has been proposed as a useful and cost effective validation tool and has been implemented for many formalisms, including Z dialects [7, 10], VDM [11], the B method [12], the SCR method [13] and RSML [14]. The advantages of animation include:

1. Providing enhanced understanding of specifications through immediate feedback;
2. Discovery of specification inconsistencies and simple errors; and
3. Validation of functions and expected behaviour through executed test cases.

For animation to be possible, the specification must be in an executable form. Hayes and Jones note the dangers of encouraging executable specifications [15], suggesting that useful specification strategies and styles are precluded and it is impossible to include assumptions or clauses that are not computable. Furthermore, executable specifications may not allow nondeterminism; a useful and sometimes necessary specification tool. They conclude that executable specifications would be best classified as rapid prototyping.

Even interpreted as a protoyping activity, interactive animation and simulation of a formal specification offers insight into a specification and improves the effectiveness of using formal methods. However, animation suffers the limitations of testing in that, results drawn from one animation scenario do not necessarily imply more universal properties about the specification behaviour. For the case study presented here it was possible to model system safety properties and software as finite state machines, and to carry out an exhaustive analysis by performing a complete search of the reachable state space. In more general situations, however, animation alone does not provide the rigorous assurance required of safety critical systems.

### 8.1.2 Static Analysis

Some formal specification languages and tools offer limited forms of automated static analysis for consistency, completeness, non determinism, reachability and deadlock [13] [14] [16]. Such tools can detect subtle errors in specifications and, where safety properties can be expressed in an appropriate form, they can be used to perform safety analysis. It has been shown that such simple checks can identify a large number of specification errors [17]. However, the analysis cannot be generally extended to more sophisticated functional safety requirements.

### 8.1.3 Model Checking

Model checking is a verification technique that has traditionally been used for rigorous hardware verification [18]. The success of hardware model checking has lead to increased application to software verification [19] [20] [21]. The aim of model checking is to systematically explore the behaviour of an operational system model for satisfaction of a set of desired functional properties. It suffers the same general problems noted for animation above, but can be more efficient in certain domains.

A number of input languages have been proposed for the specifications to be verified but most require translation into an internal representation for efficient execution. The specification langauges are commonly restricted to constructive styles, often based on finite state machines, where each state can be determined trivially from existing states. One exception is Jackson's Nitpick [22] which accepts specifications in a large subset of Z. In this case, implicit specification styles are allowed, including the use of state invariants and definition of behaviour through combination of operational schemas [6].

The largest obstacle in making software model checking feasible has been the large state spaces introduced by complex software data structures. Recent innovations have begun to successfully address this issue through state space abstraction techniques and efficient executions [20] [21] [22]. Current technology allows state spaces in the order of $10^{20}$ to be explored using practical amounts of time and computing resources.

If it is feasible, safety analysis through model checking promises to be an efficient and rigorous option. One potential obstacle lies in communicating the assurance gained from successful verification. Unlike proof, there is no reasoned argument that can be audited and the integrity of the tool must be relied upon.

### 8.1.4 Hazard Analysis

Safety analysis has traditionally been performed using semi-formal hazard analysis techniques such as Fault Tree Analysis [23]. Such techniques are usually used to decompose an identified environmental hazard into its causal failure modes and, where the failure modes relate to software, complementary safety requirements are typically derived. Assurance of software safety is then generated by demonstrating that the software specification satisfies the derived safety properties and that the specification is implemented correctly [24] [25] [26].

Application of hazard analysis is typically a manual process but some work has been done using formal specifications as the basis for partial automation of hazard analysis [14] [27]. These techniques typically use the causal relationships implicit in a specification to derive a structure the analysis. Others have created formal interpretations of hazard analysis techniques for the purpose of making the analysis more rigorous [28] [29]. In these cases, failure modes are typically expressed as properties formulated in some temporal logic but there is no proposed process for generating them automatically from a formal specification.

### 8.1.5 Proposed Approach

The safety analysis approach described in this paper initially makes use of animation technology and the expected advantages were exhibited. In particular, many minor errors in the formal specification were discovered through initial experimentation with the model. It also facilitated an interactive approach to software design, where the specification was built and "tested" incrementally. This allowed many flaws to be removed before more rigorous analysis began.

The use of animation technology to perform the rigorous safety analysis is quite different to the model checking approaches described above. In particular, the model checking engine was not built within the animation tool but was integrated externally via an application program interface. This allowed the full range of animation capabilities to be used, in particular, the specification could make use of the full expressive power of the Z language and safety analysis was performed on the specification directly. However, the performance of the analysis was substantially lower than those demonstrated for custom model checking tools. Furthermore, the model checking engine was custom built for the application

using knowledge of the application. To be applied more widely to larger, generic systems, the toolset would require significant modification. Alternatively, mature model checking technology could be used.

Regardless of the tools used, the approach to safety assurance differs significantly to traditional hazard analysis methods. In particular, we demonstrate safety of the complete software behaviour in the context of a modelled environment, rather than derive separate software safety properties and environmental assumptions.

This approach requires additional rigour in specifying assumptions about the software execution environment but it eliminates the two-step process of applying property-based hazard analysis through the system design, then validating the derived safety properties against a software specification. In fact, the rigorous environmental modelling can be viewed as an advantage since it exposes operational assumptions for external validation.

## 8.2 Production and verification of SPARK Ada Code

Some of our process is manual, particularly generation of the SPARK Ada code and proof annotations from the Sum specification. This requires maintenance of different descriptions of the system, at essentially the same level, and is a source of potential errors of translation. A more automated process would be more reliable, and also certifiable. The following different, and as yet incomplete, solutions have been proposed.

The Cogito group [30] have proposed a method for the refinement of Sum specifications to Ada code within the Cogito framework. The Ada is translated from an Intermediate Language, a subset of SUM, which contains schema descriptions of imperative statements and control structures, and which can be assembled into *op* schemas that are similar to procedures in a programming language. The Intermediate Language can be written directly, or refined from normal SUM and the refinement steps verified by Ergo, the theorem prover for Cogito.

The Cogito toolset includes a translator from IL into a subset of Ada. This subset is not SPARK-compliant. It contains forbidden features such as generic packages and declare blocks; moreover, it does not offer any annotations. However it does reflect many of the principles underlying SPARK, such as the prohibitions on unbounded types, aliasing and reading of unitialised variables. Further, Cogito retains flow information, and so has the potential to generate mandatory SPARK annotations, although it does not do so presently.

As well as program statements, an IL schema may specify conditions that do not translate directly to program, perhaps rules derived from software requirements. These are retained as Z rules on *op* schemas. At present the Ada translator converts them into simple Ada comments. In principle however, they could be translated to SPARK proof annotations, and discharged using the SPARK proof checker.

The DERA Compliance Notation [31] is built upon ProofPower, a commercial Z theorem prover. It uses a literate programming paradigm to link Z specifications, SPARK Ada code and natural language text. The user builds an Ada framework for their system, without initially providing the details of the implementation. Instead, requirements are expressed as Z statements and SPARK annotations. Progressively, these are replaced, manually, by Ada code. An associated Compliance Tool checks that the Ada code is compatible with the Z, by generating verification conditions in similar manner to the SPARK tool. VCs can be discharged using ProofPower or another Z prover such as CADiZ. Within the Notation, fragments of code are scattered throughout a descriptive text. There are tools that assemble Ada and Z fragments into complete programs for submission to a compiler, animator, or other tool.

The University of York have developed a method to convert a Z specification into the refinement calculus - essentially an implementable subset of Z together with static pre- and post- conditions [32]. It is claimed that the implementable subset is translatable into SPARK Ada, although the work is less advanced compared with Cogito. The validity of the refinement process can verified using a theorem prover like CADiZ.

## 8.3 Limitations of Approach

Although formal methods have been used extensively to verify and validate functional aspects of the case study system, some aspects have not been formalised and remain as assumptions. For a complete safety case, these aspects must be addressed by separate analyses.

### 8.3.1 Hardware Interface

An informal specification for the hardware interface is provided in Section 4.3 and the corresponding implementation in Section 6.2.3. However, the interface is not considered in the formal design or implementation assurance. In principle, it may be possible to extend the system model with a specification of the hardware design and its relationship to the software address space. However, the formalisation would add little to the informal specification. The interface specification is an important part of the design but assurance of its correctness must be provided by a separate process.

### 8.3.2 Failure Analysis and Risk Assessment

The effect of some failures, such as electro-mechanical faults in the drive mechanism, are not treated formally and the analysis assumes that they do not occur. Although the control system design is validated under sensor failures, an assumption is made that only single, permanent sensor failures will be experienced. Even under the assumed sensor failure modes, some hazardous behaviours are still found to remain.

A separate risk assessment is necessary to show that the likelihood of violating the assumptions is sufficiently low, and that the residual risk of remaining hazardous conditions is acceptable.

### 8.3.3 Timing

In Section 5.5.4 we note that some of the hazardous states can be ignored due to their transient nature, effectively eliminating the exposure of risk. In fact, this relies on the assumption of comparative execution time of the software and the behaviour of the environment. While discharging the assumption is quite simple for the system presented in this report, the analysis may be non-trivial for more complex applications.

## 8.4 Future Work

Future work could include development of the Possum animation tool to facilitate more efficient model checking. Alternatively, the use of an existing model checker could be explored. Regardless of the toolset used, the system design safety assurance approach described in this paper should be explored on a number of applications, including those with modularised software designs. An interesting direction would be to apply the approach to partitioned software systems in which the critical software is modelled and shown to be safe, assuming integrity of the design partition. The obligation to verify the partition would be then be transferred to the code verification.

We intend to investigate mechanising the translation from the Sum specification to the SPARK annotations. This would remove a source of error introduced by the manual translation and remove the burden of multiple representations, particularly with regard to change control. It is possible that some constraints on the specification style would arise but these may actually assist the specification activity.

# 9 Conclusions

This paper has demonstrated an approach to applying formal methods to verify that a given software design meets its system-safety requirements and that given source code (in Ada) meets its design requirements. We have aimed for lightweight application of formal methods which maximises the opportunities for automation of assurance tasks and reduces the requirement for verification by formal proof. The method employs a widely used formal specification language (Z) and a mixture of systematic animation, similar to model checking, and SPARK Ada code analysis.

Full validation of design safety was possible for the particular application because of its relatively simple nature, and the fact that its behaviours could be reduced to a finite set for animation. Although such full validation may not be possible in more general situations, the approach none-the-less offers clarity, traceability and automation of much of the analysis.

In the proposed method, assurance is achieved by:

1. Preliminary hazard analysis that derives top level safety requirements relate to desirable invariants of the operational system state.
2. Use of animation to execute all behaviours of a formally specified software control system in the simulated context of its environment. The environment includes a model of the controlled physical system as well as simulated sensor failure modes. Automatic comparison of executed states with the safety requirements identifies operational scenarios that can lead to unsafe states.
3. Use of the SPARK toolset to verify correctness of the Ada code against the formal software design. The Ada code is written to conform to the provided specification, with the style constrained to facilitate verification. Proof annotations are inserted Complete verification is required. Translation from the formal specification to SPARK verification annotations is required at present but we aim to increase the mechanisation of this process in the future.

The method was applied to a small case study with a control system design modelled as a state machine. Results of the verification confirm that the software is generally safe but some scenarios that violate the safety requirements are identified. Two of the scenarios are unexpectedly introduced by the detection mechanisms designed to mitigate other sensor failure modes.

The proposed approach could be applied effectively with current model checking technology and emerging code verification toolsets. While it imposes greater requirements on the modelling of system and environmental behaviour, the approach can simplify the assurance effort by eliminating the need for manual system hazard analysis or safety verification.

# 10 Acknowledgments

We thank Paul Strooper for comments on earlier versions of this paper, Dan Hazel for support of the animation and the Cogito group for the Sum formatting tool used to prepare this report.

# 11 References

[1] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor, "Cogito: A methodology and system for formal software development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, pp. 599-617, 1995.

[2] J. G. P. Barnes, *High Integrity Ada - The SPARK Approach*: Addison-Wesley, 1997.

[3] J. McDermid and T. Kelly, "Industrial Press: Safety Case," High Integrity Systems Engineering Group, University of York 1996.

[4] US Department of Defense, *MIL-STD-1629A, Procedures for Performing a Failure Mode Effects and Criticality Analysis*, 1980.

[5] O. Traynor, P. Kearney, E. Kazmierczak, L. Wang, and E. Karlson, "Extending Z with modules," *Australasian Computer Science Communications*, vol. 17, pp. 513-522, 1995.

[6] J. Woodcock and J. Davies, *Using Z*: Prentice-Hall, 1996.

[7] D. Hazel, P. Strooper, and O. Traynor, "An animator for the SUM specification language," presented at Proceedings Asia-Pacific Software Engineering Conference and International Computer Science Conference, 1997.

[8] R. Milner, "A Calculus of Communicating Systems," *Lecture Notes in Computer Science*, vol. 92, 1980.

[9] B. Welch, *Practical Programming in Tcl and Tk*: Prentice Hall, 1995.

[10] M. Hewitt, C. O'Halloran, and C. T. Sennet, “Experiences with PiZA, an Animator for Z,” presented at Proceedings ZUM'97, 1997.

[11] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore, *Mural: A Formal Development System*: Springer Verlag, 1991.

[12] J. Bicarregui, J. Dick, B. Matthews, and E. Woods, “Making the most of formal specification through animation, testing and proof,” *Science of Computer Programming*, vol. 29, pp. 53-78, 1997.

[13] C. Heitmeyer, J. K. Jr, B. Labaw, and R. Bharadwaj, “SCR*: A Toolset for Specifying and Analyzing Requirements,” presented at Proc. 10th Annual Conference on Computer Assurance (COMPASS'95), 1995.

[14] V. Ratan, K. Partridge, J. Reese, and N. Leveson, “Safety Analysis Tools for Requirements Specifications,” presented at Proc. 11th Annual Conference on Computer Assurance (COMPASS'96), 1996.

[15] I. J. Hayes and C. B. Jones, “Specifications are not (necessarily) executable,” *IEE/BCS Software Engineering Journal*, vol. 6, pp. 320-338, 1989.

[16] D. Harel, H. Lachover, A. Aaamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhenbrot, “STATEMATE: A working environment for the development of complex reactive systems,” *IEEE Transactions on Software Engineering*, vol. 16, pp. 403-414, 1990.

[17] R. R. Lutz, “Targeting Safety-Related Errors During Software Requirements Analysis,” presented at Proc. First ACM SIGSOFT Symposium of Software Engineering, Los Angeles, 1993.

[18] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, “Symbolic model checking: $10^{20}$ states and beyond,” presented at Proc. 5th Annual Symposium on Logic in Computer Science, 1990.

[19] J. M. Atlee and J. Gannon, “State-Based Model Checking of Event-Driven System Requirements,” *IEEE Transactions on Software Engineering*, vol. 19, pp. 24-40, 1993.

[20] W. Chan, R. Anderson, P. Beame, S. Burns, F. Mudugno, D. Notkin, and J. D. Reese, “Model Checking Large Software Specifications,” *IEEE Transactions on Software Engineering*, vol. 24, pp. 498-519, 1998.

[21] C. Heitmeyer, J. James Kirby, B. Labaw, M. Archer, and R. Bharadwaj, “Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications,” *IEEE Transactions on Software Engineering*, vol. 24, pp. 927-947, 1998.

[22] D. Jackson and C. A. Damon, “Elements of Style: Analysing a Software Design Feature with a Counterexample Detector,” *IEEE Transactions on Software Engineering*, vol. 22, pp. 484-495, 1996.

[23] N. H. Roberts, W. E. Vesely, D. F. Haasl, and F. F. Goldberg, *Fault Tree Handbook*: Systems and Reliability Research Office of U.S. Nuclear Regulatory Commission, 1981.

[24] Australian Department of Defence, *Def(Aust) 5679 The procurement of computer-based safety critical systems*, 2.0 ed: Codification and Standardisation Authority, 1998.

[25] RTCA Inc., *Software considerations in airborne systems and equipment certification, RTCA/DO178-B:*, 1992.

[26] UK Ministry of Defence, *The Procurement of Safety Critival Software in Defence Equipment. Defence Standard 00-55*, 1995.

[27] S. Liu and J. McDermid, “A Model-Oriented Approach to Safety Analysis Using Fault Trees and a Support System,” *J. Systems Software*, vol. 35, pp. 151-164, 1996.

[28] K. M. Hansen, A. P. Ravn, and V. Stavridou, “From Safety Analysis to Software Requirements,” *IEEE Transactions on Software Engineering*, vol. 24, pp. 573-584, 1998.

[29] J. Gorski and A. Wardinski, “Formalising Fault Trees,” presented at Proceedings of the Safety Critical Systems Symposium, 1995.

[30] P. Kearney and L. Wildman, “From Formal Specifications to Ada Programs,” The University of Queensland, SVRC Technical Report 98-24, 1998.

[31] C. O'Halloran, C. T. Sennett, and A. Smith, “Demonstrating the Compliance of Ada Programs with Z Specifications,” presented at Proceedings of the 5th Refinement Workshop, 1992.

[32] D. T. Jordan, C. J. Locke, J. A. McDermid, C. E. Parker, B. A. P. Sharp, and I. Toyn, “Literate Formal Development of Ada from Z for Safety-Critical Application,",” presented at SAFECOMP'94, 1994.

# Appendix A Software Design Sum Specifications

Following Sum modules are provided:

1. Sensors
2. Actuator
3. Machine

```
Sensors
  // ------------------------------------------------
  // Industrial Press Sensors Module
  // Press_Sensors models the state of the Press Sensor signals.
  // Each sensor may produce a high or low signal.
  // ------------------------------------------------
  IN_SIG ::= high | low
  // Binary input signal
  state
    button, top, PoNR, bottom: IN_SIG
```

```
Actuator
  // ----------------------------------------------------------
  // Industrial Press Actuator Drive
  // The motor drive module is imported by the state machine.
  // The only contents are the type and state of the motor drive
  // signal.
  // ----------------------------------------------------------
  // -------------------------------
  // Type Declarations
  // -------------------------------
  OUT_SIG ::= a_on | a_off
  // -------------------------------
  // State
  // -------------------------------
  state
    motor: OUT_SIG
```

```
Machine
  // -----------------------------------------------------------------
  // Industrial Press SW State Machine Module
  // Captures the logic of the Press control system.
  // Values of Press sensor and drive signals are imported as variables
  // to be manipulated by the control system.
  // -----------------------------------------------------------------
  import Sensors
  import Actuator
  // -------------------------------
  // Type Declarations
```

```
// -------------------------------
CONTROL ::= opening | open | ready | closing | uncond_closing |
            halt_open | halt_closed
// ----------------------------------------------------------------
// State
// Control logic state includes the logical software state
// as well as the state of press sensor and drive signals.
// ----------------------------------------------------------------
state
    control: CONTROL
    Sensors.state
    Actuator.state

// -------------------------------------------------
// Initialisation
// -------------------------------------------------
init
    control' = opening
    Actuator.motor' = Actuator.a_on

// --------------------------------------------------
// Operations
// Each operation corresponds to behaviour exercised
// in a particular state.  The precondition of each operation
// is the associated state.
// For each operation, current values of sensor signals are used
// to determine the appropriate state transition and change the
// Press motor drive signal.
// Only the software state and motor signal may be modified.
//
// Execution of the state machine is assumed to occur by repeatedly
// invoking the operation with an active (true) preconditon.
// Selection of operations is deterministic since operation
// preconditions are mutually exclusive.
// -------------------------------------------------------------
op At_Opening
    pre control = opening
    if Sensors.top = Sensors.high
    then
      (control' = open ∧ Actuator.motor' = Actuator.a_on)
    else
      (control' = opening ∧ Actuator.motor' = Actuator.a_on)
    fi
    changes_only{control, Actuator.motor}

op At_Open
    pre control = open
    if (Sensors.bottom = Sensors.high) ∨
      (Sensors.PoNR = Sensors.high) ∨
```

```
      (Sensors.button = Sensors.high)
    then
      (control' = halt_open ∧ Actuator.motor' = Actuator.a_on)
    else
      (control' = ready ∧ Actuator.motor' = Actuator.a_on)
    fi
    changes_only{control, Actuator.motor}

op At_Ready
    pre control = ready
    if Sensors.bottom = Sensors.high ∨
      Sensors.PoNR = Sensors.high
    then
      (control' = halt_open ∧
      Actuator.motor' = Actuator.a_on)
    else
      (if Sensors.button = Sensors.high
      then
        (control' = closing ∧ Actuator.motor' = Actuator.a_off)
      else
        (control' = ready ∧ Actuator.motor' = Actuator.a_on)
      fi)
    fi
    changes_only{control, Actuator.motor}

op At_Closing
    pre control = closing
    if Sensors.bottom = Sensors.high
    then
      (control' = halt_closed ∧ Actuator.motor' = Actuator.a_off)
    else
      (if Sensors.PoNR = Sensors.high
      then
        (control' = uncond_closing ∧ Actuator.motor' = Actuator.a_off)
      else
        (if Sensors.button = Sensors.low
        then
          (control' = opening ∧ Actuator.motor' = Actuator.a_on)
        else
          (control' = closing ∧ Actuator.motor' = Actuator.a_off)
        fi)
      fi)
    fi
    changes_only{control, Actuator.motor}

op At_Uncond_Closing
    pre control = uncond_closing
    if Sensors.top = Sensors.high
    then
```

(*control'* = *halt_closed* ∧ *Actuator.motor'* = *Actuator.a_off*)
else
(if *Sensors.bottom* = *Sensors.high* ∧
*Sensors.top* = *Sensors.low*
then
(*control'* = *opening* ∧ *Actuator.motor'* = *Actuator.a_on*)
else
(*control'* = *uncond_closing* ∧
*Actuator.motor'* = *Actuator.a_off*)
fi)
fi
*changes_only*{*control, Actuator.motor*}

*op At_Halt_Open*
pre *control* = *halt_open*
*changes_only*{}

*op At_Halt_Closed*
pre *control* = *halt_closed*
*changes_only*{}

*Transition* == (*At_Opening* ∨ *At_Open* ∨ *At_Ready* ∨ *At_Closing* ∨ *At_Uncond_Closing* ∨ *At_Halt_Open* ∨ *At_Halt_Closed*)

# Appendix B Simulation Environment Sum Specifications

The *Simulator* module specification is provided.

*Simulator*

```
// ----------------------------------------
// Industrial Press Simulation Environment
// The environment module provides a simulation environment for the Press
// logic state machine.  The simulation is performed at a physical level.
// The modelled physical state of the press is used to drive sensor signals.
// These, in turn, drive the control logic which then causes a physical
// state change.  It is possible to activate sensor failures and
// investigate their effect.
// ----------------------------------------
import Sensors
import Actuator
import Machine
// -------------------------------
// State
// In addition to encapsulating the logic, sensor and motor drive states,
// the state represents the physical movement of the plunger.
// -------------------------------
SENSOR_HEALTH ::= broken | ok
PLUNGER ::= at_bottom | below_PoNR | above_PoNR | at_top
        falling_past_PoNR | falling_to_bottom
states_above_top == {at_top}
states_above_PoNR == {above_PoNR, at_top, falling_past_PoNR}
states_above_bottom == {s: PLUNGER | s ≠ at_bottom}
SAFETY ::= safe | abort_failed | unsafe_motor_drive
BUTTON ::= pressed | released
high == Sensors.high
low == Sensors.low

    next_plunger_state: (PLUNGER × Actuator.OUT_SIG) → PLUNGER
    ----------------------------------------------------------
    next_plunger_state(at_bottom, Actuator.a_off) = at_bottom
    next_plunger_state(at_bottom, Actuator.a_on) = below_PoNR
    next_plunger_state(below_PoNR, Actuator.a_off) = falling_to_bottom
    next_plunger_state(below_PoNR, Actuator.a_on) = above_PoNR
    next_plunger_state(above_PoNR, Actuator.a_off) = falling_past_PoNR
    next_plunger_state(above_PoNR, Actuator.a_on) = at_top
    next_plunger_state(at_top, Actuator.a_off) = above_PoNR
    next_plunger_state(at_top, Actuator.a_on) = at_top
    next_plunger_state(falling_past_PoNR, Actuator.a_off) = falling_to_bottom
    next_plunger_state(falling_past_PoNR, Actuator.a_on) = below_PoNR
    next_plunger_state(falling_to_bottom, Actuator.a_off) = at_bottom
    next_plunger_state(falling_to_bottom, Actuator.a_on) = at_bottom
```

```
__state________________________________________
    Sensors.state
    Actuator.state
    Machine.state
    plunger: PLUNGER
    button: BUTTON
    safety: SAFETY
    button_health, top_health, PoNR_health, bottom_health: SENSOR_HEALTH
_______________________________________________
    safety = if ((plunger = falling_to_bottom and
            (Actuator.motor = Actuator.a_on)) then
            unsafe_motor_drive
        else
            if ((plunger in states_above_PoNR) and
          (Actuator.motor /= Actuator.a_on) and
          (button = released)) then
          abort_failed
           else
          safe
           fi
        fi
_______________________________________________
```

```
// ------------------------------------------------
// Initialisation
// Initialises sensor signals, logic state machine and physical
// state.
// ------------------------------------------------
```

```
__init_________________________________________
    Machine.init
    Sensors.button' = Sensors.low
    Sensors.top' = Sensors.low
    Sensors.PoNR' = Sensors.high
    Sensors.bottom' = Sensors.high
    button_health' = ok
    top_health' = ok
    PoNR_health' = ok
    bottom_health' = ok
    plunger' = at_bottom
    button' = released
_______________________________________________
```

```
// ------------------------------------------------
// Operations
// Simulation operations are provided to manipulate the system
// at a physical level.
// The button may be pressed or released.
```

```
//  Sensors may be forced to fail at any time.  Failures are considered
//  to be permanent.
//  The plunger moves between physical states in accordance with
//  expected physical laws.  The granularity of movement is to allow
//  realistic simulation of movement to be interleaved with sensor failures
//  and operator behaviour.
//  Plunger movements are driven by current movement and motor drive.
//  Each movement modifies sensor signals as expected and allows execution
//  of the control logic.
//  -------------------------------------------------
//  -------------------------------------------------
//  Button operations
//  -------------------------------------------------
```

*op Push_Transition*

*Sensors.button'* = (if (*button_health* = *ok*) then *high* else *Sensors.button* fi )
*button'* = *pressed*
*changes_only*{*Sensors.button, button, safety*}

*op Release_Transition*

*Sensors.button'* = (if (*button_health* = *ok*) then *low* else *Sensors.button* fi )
*button'* = *released*
*changes_only*{*Sensors.button, button, safety*}

```
//  -------------------------------------------------
//  Sensor Failure operations
//  -------------------------------------------------
```

*op Top_Fail_High*

*Sensors.top'* = *high*
*top_health'* = *broken*
*changes_only*{*Sensors.top, top_health*}

*op Top_Fail_Low*

*Sensors.top'* = *low*
*top_health'* = *broken*
*changes_only*{*Sensors.top, top_health*}

*op PoNR_Fail_High*

*Sensors.PoNR'* = *high*
*PoNR_health'* = *broken*
*changes_only*{*Sensors.PoNR, PoNR_health*}

*op PoNR_Fail_Low*

```
    Sensors.PoNR' = low
    PoNR_health' = broken
    changes_only{Sensors.PoNR, PoNR_health}
```

```
op Bottom_Fail_High
    Sensors.bottom' = high
    bottom_health' = broken
    changes_only{Sensors.bottom, bottom_health}
```

```
op Bottom_Fail_Low
    Sensors.bottom' = low
    bottom_health' = broken
    changes_only{Sensors.bottom, bottom_health}
```

```
op Button_Fail_High
    Sensors.button' = high
    button_health' = broken
    changes_only{Sensors.button, button_health}
```

```
op Button_Fail_Low
    Sensors.button' = low
    button_health' = broken
    changes_only{Sensors.button, button_health}
```

```
//  --------------------------------------------------
// Plunger Transition
//
//  Movement is governed by a state machine which transitions between
//  states in accordance with current motor drive.
//  --------------------------------------------------
op Plunger_Transition
    plunger' = next_plunger_state(plunger, Actuator.motor)
    Sensors.top' =
     (if top_health = ok then
        (if (next_plunger_state (plunger, Actuator.motor) ∈
           states_above_top) then high else low fi )
     else
          Sensors.top
     fi )
    Sensors.PoNR' =
     (if PoNR_health = ok then
        if (next_plunger_state (plunger, Actuator.motor) ∈
           states_above_PoNR) then low else high fi )
```

```
        else
            Sensors.PoNR
        fi )
      Sensors.bottom' =
        (if bottom_health = ok then
           (if (next_plunger_state(plunger, Actuator.motor) ∈
              states_above_bottom) then low else high fi )
        else
            Sensors.bottom
        fi )
      changes_only{plunger, Sensors.top, Sensors.PoNR, Sensors.bottom, safety}

  // ---------------------------------------------------------
  // Can also remain at a steady physical state in order to test
  // behaviour when no sensor signals change.
  // ---------------------------------------------------------
  op Control_Transition
      Machine.Transition
      changes_only{Machine.control, Actuator.motor, safety}
```

# Appendix C Simulation Results

The results of the simulations under each of the sensor failures are presented below.

### Simulation Under No Sensor Failure

```
=========================================================
Model check for Industrial Press Control System
=========================================================
Number of states = 32
-----------------------------------------------------------------------------------
    plunger    control     button    motor    safe      C.T  B.T P.T
-----------------------------------------------------------------------------------
1   at_bottom  opening     released  a_on     safe      1    2    4
2   at_bottom  opening     pressed   a_on     safe      2    1    3
3   below_PoNR opening     pressed   a_on     safe      3    4    6
4   below_PoNR opening     released  a_on     safe      4    3    5
5   above_PoNR opening     released  a_on     safe      5    6    16
6   above_PoNR opening     pressed   a_on     safe      6    5    7
7   at_top     opening     pressed   a_on     safe      8    16   7
8   at_top     open        pressed   a_on     safe      9    11   8
9   at_top     halt_open   pressed   a_on     safe      9    10   9
10  at_top     halt_open   released  a_on     safe      10   9    10
11  at_top     open        released  a_on     safe      12   8    11
12  at_top     ready       released  a_on     safe      12   13   12
13  at_top     ready       pressed   a_on     safe      14   12   13
14  at_top     closing     pressed   a_off    safe      14   15   18
15  at_top     closing     released  a_off    no_abort  16   14   17
16  at_top     opening     released  a_on     safe      11   7    16
17  above_PoNR closing     released  a_off    no_abort  5    18   20
18  above_PoNR closing     pressed   a_off    safe      18   17   19
19  past_PoNR  closing     pressed   a_off    safe      19   20   28
20  past_PoNR  closing     released  a_off    no_abort  21   19   23
21  past_PoNR  opening     released  a_on     safe      21   22   4
22  past_PoNR  opening     pressed   a_on     safe      22   21   3
23  to_bottom  closing     released  a_off    safe      24   28   32
24  to_bottom  uncond_cls  released  a_off    safe      24   25   27
25  to_bottom  uncond_cls  pressed   a_off    safe      25   24   26
26  at_bottom  uncond_cls  pressed   a_off    safe      2    27   26
27  at_bottom  uncond_cls  released  a_off    safe      1    26   27
28  to_bottom  closing     pressed   a_off    safe      25   23   29
29  at_bottom  closing     pressed   a_off    safe      30   32   29
30  at_bottom  halt_closed pressed   a_off    safe      30   31   30
31  at_bottom  halt_closed released  a_off    safe      31   30   31
32  at_bottom  closing     released  a_off    safe      31   29   32
-----------------------------------------------------------------------------------
```

### Simulation Under Bottom Sensor Low Failure

```
=========================================================
Model check for Industrial Press Control System Bottom_Fail_Low
=========================================================
Number of states = 64
-----------------------------------------------------------------------------------
    plunger     control    button    motor    safe      bottom  C.T  B.T  P.T  F.T
-----------------------------------------------------------------------------------
1   at_bottom  opening     released  a_on     safe      ok      1    2    4    64
2   at_bottom  opening     pressed   a_on     safe      ok      2    1    3    63
3   below_PoNR opening     pressed   a_on     safe      ok      3    4    6    35
4   below_PoNR opening     released  a_on     safe      ok      4    3    5    36
5   above_PoNR opening     released  a_on     safe      ok      5    6    18   26
6   above_PoNR opening     pressed   a_on     safe      ok      6    5    7    27
7   at_top     opening     pressed   a_on     safe      ok      8    18   7    28
8   at_top     open        pressed   a_on     safe      ok      9    13   8    29
9   at_top     halt_open   pressed   a_on     safe      ok      9    10   9    12
10  at_top     halt_open   released  a_on     safe      ok      10   9    10   11
11  at_top     halt_open   released  a_on     safe      broken  11   12   11   11
12  at_top     halt_open   pressed   a_on     safe      broken  12   11   12   12
13  at_top     open        released  a_on     safe      ok      14   8    13   20
14  at_top     ready       released  a_on     safe      ok      14   15   14   21
15  at_top     ready       pressed   a_on     safe      ok      16   14   15   22
16  at_top     closing     pressed   a_off    safe      ok      16   17   46   23
17  at_top     closing     released  a_off    no_abort  ok      18   16   45   24
18  at_top     opening     released  a_on     safe      ok      13   7    18   19
19  at_top     opening     released  a_on     safe      broken  20   28   19   19
20  at_top     open        released  a_on     safe      broken  21   29   20   20
```

```
21  at_top     ready      released  a_on   safe      broken  21  22  21  21
22  at_top     ready      pressed   a_on   safe      broken  23  21  22  22
23  at_top     closing    pressed   a_off  safe      broken  23  24  30  23
24  at_top     closing    released  a_off  no_abort  broken  19  23  25  24
25  above_PoNR closing    released  a_off  no_abort  broken  26  30  32  25
26  above_PoNR opening    released  a_on   safe      broken  26  27  19  26
27  above_PoNR opening    pressed   a_on   safe      broken  27  26  28  27
28  at_top     opening    pressed   a_on   safe      broken  29  19  28  28
29  at_top     open       pressed   a_on   safe      broken  12  20  29  29
30  above_PoNR closing    pressed   a_off  safe      broken  30  25  31  30
31  past_PoNR  closing    pressed   a_off  safe      broken  31  32  42  31
32  past_PoNR  closing    released  a_off  no_abort  broken  33  31  37  32
33  past_PoNR  opening    released  a_on   safe      broken  33  34  36  33
34  past_PoNR  opening    pressed   a_on   safe      broken  34  33  35  34
35  below_PoNR opening    pressed   a_on   safe      broken  35  36  27  35
36  below_PoNR opening    released  a_on   safe      broken  36  35  26  36
37  to_bottom  closing    released  a_off  safe      broken  38  42  44  37
38  to_bottom  uncond_cls released  a_off  safe      broken  38  39  41  38
39  to_bottom  uncond_cls pressed   a_off  safe      broken  39  38  40  39
40  at_bottom  uncond_cls pressed   a_off  safe      broken  40  41  40  40
41  at_bottom  uncond_cls released  a_off  safe      broken  41  40  41  41
42  to_bottom  closing    pressed   a_off  safe      broken  39  37  43  42
43  at_bottom  closing    pressed   a_off  safe      broken  40  44  43  43
44  at_bottom  closing    released  a_off  safe      broken  41  43  44  44
45  above_PoNR closing    released  a_off  no_abort  ok      5   46  48  25
46  above_PoNR closing    pressed   a_off  safe      ok      46  45  47  30
47  past_PoNR  closing    pressed   a_off  safe      ok      47  48  56  31
48  past_PoNR  closing    released  a_off  no_abort  ok      49  47  51  32
49  past_PoNR  opening    released  a_on   safe      ok      49  50  4   33
50  past_PoNR  opening    pressed   a_on   safe      ok      50  49  3   34
51  to_bottom  closing    released  a_off  safe      ok      52  56  62  37
52  to_bottom  uncond_cls released  a_off  safe      ok      52  53  55  38
53  to_bottom  uncond_cls pressed   a_off  safe      ok      53  52  54  39
54  at_bottom  uncond_cls pressed   a_off  safe      ok      2   55  54  40
55  at_bottom  uncond_cls released  a_off  safe      ok      1   54  55  41
56  to_bottom  closing    pressed   a_off  safe      ok      53  51  57  42
57  at_bottom  closing    pressed   a_off  safe      ok      58  62  57  43
58  at_bottom  halt_closed pressed  a_off  safe      ok      58  59  58  61
59  at_bottom  halt_closed released a_off  safe      ok      59  58  59  60
60  at_bottom  halt_closed released a_off  safe      broken  60  61  60  60
61  at_bottom  halt_closed pressed  a_off  safe      broken  61  60  61  61
62  at_bottom  closing    released  a_off  safe      ok      59  57  62  44
63  at_bottom  opening    pressed   a_on   safe      broken  63  64  35  63
64  at_bottom  opening    released  a_on   safe      broken  64  63  36  64
-----------------------------------------------------------------------------------
```

## Simulation Under Button Sensor High Failure

```
=========================================================
Model check for Industrial Press Control System Bottom_Fail_High
=========================================================
Number of states 74
-----------------------------------------------------------------------------------
    plunger    control    button    motor  safe      bottom  C.T B.T P.T .T
-----------------------------------------------------------------------------------
1   at_bottom  opening    released  a_on   safe      ok      1   2   4   41
2   at_bottom  opening    pressed   a_on   safe      ok      2   1   3   42
3   below_PoNR opening    pressed   a_on   safe      ok      3   4   6   32
4   below_PoNR opening    released  a_on   safe      ok      4   3   5   31
5   above_PoNR opening    released  a_on   safe      ok      5   6   18  34
6   above_PoNR opening    pressed   a_on   safe      ok      6   5   7   33
7   at_top     opening    pressed   a_on   safe      ok      8   18  7   22
8   at_top     open       pressed   a_on   safe      ok      9   13  8   21
9   at_top     halt_open  pressed   a_on   safe      ok      9   10  9   12
10  at_top     halt_open  released  a_on   safe      ok      10  9   10  11
11  at_top     halt_open  released  a_on   safe      broken  11  12  11  11
12  at_top     halt_open  pressed   a_on   safe      broken  12  11  12  12
13  at_top     open       released  a_on   safe      ok      14  8   13  20
14  at_top     ready      released  a_on   safe      ok      14  15  14  74
15  at_top     ready      pressed   a_on   safe      ok      16  14  15  73
16  at_top     closing    pressed   a_off  safe      ok      16  17  24  72
17  at_top     closing    released  a_off  no_abort  ok      18  16  23  69
18  at_top     opening    released  a_on   safe      ok      13  7   18  19
19  at_top     opening    released  a_on   safe      broken  20  22  19  19
20  at_top     open       released  a_on   safe      broken  11  21  20  20
21  at_top     open       pressed   a_on   safe      broken  12  20  21  21
```

```
22  at_top     opening     pressed   a_on     safe      broken  21   19   22   22
23  above_PoNR closing     released  a_off    no_abort  ok      5    24   26   68
24  above_PoNR closing     pressed   a_off    safe      ok      24   23   25   65
25  past_PoNR  closing     pressed   a_off    safe      ok      25   26   48   64
26  past_PoNR  closing     released  a_off    no_abort  ok      27   25   35   61
27  past_PoNR  opening     released  a_on     safe      ok      27   28   4    30
28  past_PoNR  opening     pressed   a_on     safe      ok      28   27   3    29
29  past_PoNR  opening     pressed   a_on     safe      broken  29   30   32   29
30  past_PoNR  opening     released  a_on     safe      broken  30   29   31   30
31  below_PoNR opening     released  a_on     safe      broken  31   32   34   31
32  below_PoNR opening     pressed   a_on     safe      broken  32   31   33   32
33  above_PoNR opening     pressed   a_on     safe      broken  33   34   22   33
34  above_PoNR opening     released  a_on     safe      broken  34   33   19   34
35  to_bottom  closing     released  a_off    safe      ok      36   48   54   60
36  to_bottom  uncond_cls  released  a_off    safe      ok      36   37   39   47
37  to_bottom  uncond_cls  pressed   a_off    safe      ok      37   36   38   44
38  at_bottom  uncond_cls  pressed   a_off    safe      ok      2    39   38   43
39  at_bottom  uncond_cls  released  a_off    safe      ok      1    38   39   40
40  at_bottom  uncond_cls  released  a_off    safe      broken  41   43   40   40
41  at_bottom  opening     released  a_on     safe      broken  41   42   31   41
42  at_bottom  opening     pressed   a_on     safe      broken  42   41   32   42
43  at_bottom  uncond_cls  pressed   a_off    safe      broken  42   40   43   43
44  to_bottom  uncond_cls  pressed   a_off    safe      broken  45   47   43   44
45  to_bottom  opening     pressed   a_on     bad_drv   broken  45   46   42   45
46  to_bottom  opening     released  a_on     bad_drv   broken  46   45   41   46
47  to_bottom  uncond_cls  released  a_off    safe      broken  46   44   40   47
48  to_bottom  closing     pressed   a_off    safe      ok      37   35   49   57
49  at_bottom  closing     pressed   a_off    safe      ok      50   54   49   56
50  at_bottom  halt_closed pressed   a_off    safe      ok      50   51   50   53
51  at_bottom  halt_closed released  a_off    safe      ok      51   50   51   52
52  at_bottom  halt_closed released  a_off    safe      broken  52   53   52   52
53  at_bottom  halt_closed pressed   a_off    safe      broken  53   52   53   53
54  at_bottom  closing     released  a_off    safe      ok      51   49   54   55
55  at_bottom  closing     released  a_off    safe      broken  52   56   55   55
56  at_bottom  closing     pressed   a_off    safe      broken  53   55   56   56
57  to_bottom  closing     pressed   a_off    safe      broken  58   60   56   57
58  to_bottom  halt_closed pressed   a_off    safe      broken  58   59   53   58
59  to_bottom  halt_closed released  a_off    safe      broken  59   58   52   59
60  to_bottom  closing     released  a_off    safe      broken  59   57   55   60
61  past_PoNR  closing     released  a_off    no_abort  broken  62   64   60   61
62  past_PoNR  halt_closed released  a_off    no_abort  broken  62   63   59   62
63  past_PoNR  halt_closed pressed   a_off    safe      broken  63   62   58   63
64  past_PoNR  closing     pressed   a_off    safe      broken  63   61   57   64
65  above_PoNR closing     pressed   a_off    safe      broken  66   68   64   65
66  above_PoNR halt_closed pressed   a_off    safe      broken  66   67   63   66
67  above_PoNR halt_closed released  a_off    no_abort  broken  67   66   62   67
68  above_PoNR closing     released  a_off    no_abort  broken  67   65   61   68
69  at_top     closing     released  a_off    no_abort  broken  70   72   68   69
70  at_top     halt_closed released  a_off    no_abort  broken  70   71   67   70
71  at_top     halt_closed pressed   a_off    safe      broken  71   70   66   71
72  at_top     closing     pressed   a_off    safe      broken  71   69   65   72
73  at_top     ready       pressed   a_on     safe      broken  12   74   73   73
74  at_top     ready       released  a_on     safe      broken  11   73   74   74
----------------------------------------------------------------------------------
```

### Simulation Under PoNR Sensor Low Failure

```
=========================================================
Model check for Industrial Press Control System PoNR_Fail_Low
=========================================================
Number of states = 66
----------------------------------------------------------------------------------
    plunger    control     button    motor    safe      PoNR    C.T  B.T  P.T  F.T
----------------------------------------------------------------------------------
1   at_bottom  opening     released  a_on     safe      ok      1    2    4    41
2   at_bottom  opening     pressed   a_on     safe      ok      2    1    3    40
3   below_PoNR opening     pressed   a_on     safe      ok      3    4    6    35
4   below_PoNR opening     released  a_on     safe      ok      4    3    5    36
5   above_PoNR opening     released  a_on     safe      ok      5    6    18   26
6   above_PoNR opening     pressed   a_on     safe      ok      6    5    7    27
7   at_top     opening     pressed   a_on     safe      ok      8    18   7    28
8   at_top     open        pressed   a_on     safe      ok      9    13   8    29
9   at_top     halt_open   pressed   a_on     safe      ok      9    10   9    12
10  at_top     halt_open   released  a_on     safe      ok      10   9    10   11
11  at_top     halt_open   released  a_on     safe      broken  11   12   11   11
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 12 | at_top | halt_open | pressed | a_on | safe | broken | 12 | 11 | 12 | 12 |
| 13 | at_top | open | released | a_on | safe | ok | 14 | 8 | 13 | 20 |
| 14 | at_top | ready | released | a_on | safe | ok | 14 | 15 | 14 | 21 |
| 15 | at_top | ready | pressed | a_on | safe | ok | 16 | 14 | 15 | 22 |
| 16 | at_top | closing | pressed | a_off | safe | ok | 16 | 17 | 48 | 23 |
| 17 | at_top | closing | released | a_off | no_abort | ok | 18 | 16 | 47 | 24 |
| 18 | at_top | opening | released | a_on | safe | ok | 13 | 7 | 18 | 19 |
| 19 | at_top | opening | released | a_on | safe | broken | 20 | 28 | 19 | 19 |
| 20 | at_top | open | released | a_on | safe | broken | 21 | 29 | 20 | 20 |
| 21 | at_top | ready | released | a_on | safe | broken | 21 | 22 | 21 | 21 |
| 22 | at_top | ready | pressed | a_on | safe | broken | 23 | 21 | 22 | 22 |
| 23 | at_top | closing | pressed | a_off | safe | broken | 23 | 24 | 30 | 23 |
| 24 | at_top | closing | released | a_off | no_abort | broken | 19 | 23 | 25 | 24 |
| 25 | above_PoNR | closing | released | a_off | no_abort | broken | 26 | 30 | 32 | 25 |
| 26 | above_PoNR | opening | released | a_on | safe | broken | 26 | 27 | 19 | 26 |
| 27 | above_PoNR | opening | pressed | a_on | safe | broken | 27 | 26 | 28 | 27 |
| 28 | at_top | opening | pressed | a_on | safe | broken | 29 | 19 | 28 | 28 |
| 29 | at_top | open | pressed | a_on | safe | broken | 12 | 20 | 29 | 29 |
| 30 | above_PoNR | closing | pressed | a_off | safe | broken | 30 | 25 | 31 | 30 |
| 31 | past_PoNR | closing | pressed | a_off | safe | broken | 31 | 32 | 42 | 31 |
| 32 | past_PoNR | closing | released | a_off | no_abort | broken | 33 | 31 | 37 | 32 |
| 33 | past_PoNR | opening | released | a_on | safe | broken | 33 | 34 | 36 | 33 |
| 34 | past_PoNR | opening | pressed | a_on | safe | broken | 34 | 33 | 35 | 34 |
| 35 | below_PoNR | opening | pressed | a_on | safe | broken | 35 | 36 | 27 | 35 |
| 36 | below_PoNR | opening | released | a_on | safe | broken | 36 | 35 | 26 | 36 |
| 37 | to_bottom | closing | released | a_off | safe | broken | 38 | 42 | 46 | 37 |
| 38 | to_bottom | opening | released | a_on | bad_drv | broken | 38 | 39 | 41 | 38 |
| 39 | to_bottom | opening | pressed | a_on | bad_drv | broken | 39 | 38 | 40 | 39 |
| 40 | at_bottom | opening | pressed | a_on | safe | broken | 40 | 41 | 35 | 40 |
| 41 | at_bottom | opening | released | a_on | safe | broken | 41 | 40 | 36 | 41 |
| 42 | to_bottom | closing | pressed | a_off | safe | broken | 42 | 37 | 43 | 42 |
| 43 | at_bottom | closing | pressed | a_off | safe | broken | 44 | 46 | 43 | 43 |
| 44 | at_bottom | halt_closed | pressed | a_off | safe | broken | 44 | 45 | 44 | 44 |
| 45 | at_bottom | halt_closed | released | a_off | safe | broken | 45 | 44 | 45 | 45 |
| 46 | at_bottom | closing | released | a_off | safe | broken | 45 | 43 | 46 | 46 |
| 47 | above_PoNR | closing | released | a_off | no_abort | ok | 5 | 48 | 50 | 25 |
| 48 | above_PoNR | closing | pressed | a_off | safe | ok | 48 | 47 | 49 | 30 |
| 49 | past_PoNR | closing | pressed | a_off | safe | ok | 49 | 50 | 62 | 31 |
| 50 | past_PoNR | closing | released | a_off | no_abort | ok | 51 | 49 | 53 | 32 |
| 51 | past_PoNR | opening | released | a_on | safe | ok | 51 | 52 | 4 | 33 |
| 52 | past_PoNR | opening | pressed | a_on | safe | ok | 52 | 51 | 3 | 34 |
| 53 | to_bottom | closing | released | a_off | safe | ok | 54 | 62 | 66 | 37 |
| 54 | to_bottom | uncond_cls | released | a_off | safe | ok | 54 | 55 | 57 | 61 |
| 55 | to_bottom | uncond_cls | pressed | a_off | safe | ok | 55 | 54 | 56 | 60 |
| 56 | at_bottom | uncond_cls | pressed | a_off | safe | ok | 2 | 57 | 56 | 59 |
| 57 | at_bottom | uncond_cls | released | a_off | safe | ok | 1 | 56 | 57 | 58 |
| 58 | at_bottom | uncond_cls | released | a_off | safe | broken | 41 | 59 | 58 | 58 |
| 59 | at_bottom | uncond_cls | pressed | a_off | safe | broken | 40 | 58 | 59 | 59 |
| 60 | to_bottom | uncond_cls | pressed | a_off | safe | broken | 60 | 61 | 59 | 60 |
| 61 | to_bottom | uncond_cls | released | a_off | safe | broken | 61 | 60 | 58 | 61 |
| 62 | to_bottom | closing | pressed | a_off | safe | ok | 55 | 53 | 63 | 42 |
| 63 | at_bottom | closing | pressed | a_off | safe | ok | 64 | 66 | 63 | 43 |
| 64 | at_bottom | halt_closed | pressed | a_off | safe | ok | 64 | 65 | 64 | 44 |
| 65 | at_bottom | halt_closed | released | a_off | safe | ok | 65 | 64 | 65 | 45 |
| 66 | at_bottom | closing | released | a_off | safe | ok | 65 | 63 | 66 | 46 |

## Simulation Under PoNR Sensor High Failure

```
=========================================================
Model check for Industrial Press Control System PoNR_Fail_High
=========================================================
Number of states = 78
```

| | plunger | control | button | motor | safe | PoNR | C.T | B.T | P.T | F.T |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | at_bottom | opening | released | a_on | safe | ok | 1 | 2 | 4 | 41 |
| 2 | at_bottom | opening | pressed | a_on | safe | ok | 2 | 1 | 3 | 42 |
| 3 | below_PoNR | opening | pressed | a_on | safe | ok | 3 | 4 | 6 | 32 |
| 4 | below_PoNR | opening | released | a_on | safe | ok | 4 | 3 | 5 | 31 |
| 5 | above_PoNR | opening | released | a_on | safe | ok | 5 | 6 | 18 | 34 |
| 6 | above_PoNR | opening | pressed | a_on | safe | ok | 6 | 5 | 7 | 33 |
| 7 | at_top | opening | pressed | a_on | safe | ok | 8 | 18 | 7 | 22 |
| 8 | at_top | open | pressed | a_on | safe | ok | 9 | 13 | 8 | 21 |
| 9 | at_top | halt_open | pressed | a_on | safe | ok | 9 | 10 | 9 | 12 |
| 10 | at_top | halt_open | released | a_on | safe | ok | 10 | 9 | 10 | 11 |

```
11  at_top     halt_open   released  a_on    safe      broken  11   12   11   11
12  at_top     halt_open   pressed   a_on    safe      broken  12   11   12   12
13  at_top     open        released  a_on    safe      ok      14   8    13   20
14  at_top     ready       released  a_on    safe      ok      14   15   14   78
15  at_top     ready       pressed   a_on    safe      ok      16   14   15   77
16  at_top     closing     pressed   a_off   safe      ok      16   17   24   76
17  at_top     closing     released  a_off   no_abort  ok      18   16   23   65
18  at_top     opening     released  a_on    safe      ok      13   7    18   19
19  at_top     opening     released  a_on    safe      broken  20   22   19   19
20  at_top     open        released  a_on    safe      broken  11   21   20   20
21  at_top     open        pressed   a_on    safe      broken  12   20   21   21
22  at_top     opening     pressed   a_on    safe      broken  21   19   22   22
23  above_PoNR closing     released  a_off   no_abort  ok      5    24   26   64
24  above_PoNR closing     pressed   a_off   safe      ok      24   23   25   61
25  past_PoNR  closing     pressed   a_off   safe      ok      25   26   46   60
26  past_PoNR  closing     released  a_off   no_abort  ok      27   25   35   57
27  past_PoNR  opening     released  a_on    safe      ok      27   28   4    30
28  past_PoNR  opening     pressed   a_on    safe      ok      28   27   3    29
29  past_PoNR  opening     pressed   a_on    safe      broken  29   30   32   29
30  past_PoNR  opening     released  a_on    safe      broken  30   29   31   30
31  below_PoNR opening     released  a_on    safe      broken  31   32   34   31
32  below_PoNR opening     pressed   a_on    safe      broken  32   31   33   32
33  above_PoNR opening     pressed   a_on    safe      broken  33   34   22   33
34  above_PoNR opening     released  a_on    safe      broken  34   33   19   34
35  to_bottom  closing     released  a_off   safe      ok      36   46   52   56
36  to_bottom  uncond_cls  released  a_off   safe      ok      36   37   39   45
37  to_bottom  uncond_cls  pressed   a_off   safe      ok      37   36   38   44
38  at_bottom  uncond_cls  pressed   a_off   safe      ok      2    39   38   43
39  at_bottom  uncond_cls  released  a_off   safe      ok      1    38   39   40
40  at_bottom  uncond_cls  released  a_off   safe      broken  41   43   40   40
41  at_bottom  opening     released  a_on    safe      broken  41   42   31   41
42  at_bottom  opening     pressed   a_on    safe      broken  42   41   32   42
43  at_bottom  uncond_cls  pressed   a_off   safe      broken  42   40   43   43
44  to_bottom  uncond_cls  pressed   a_off   safe      broken  44   45   43   44
45  to_bottom  uncond_cls  released  a_off   safe      broken  45   44   40   45
46  to_bottom  closing     pressed   a_off   safe      ok      37   35   47   55
47  at_bottom  closing     pressed   a_off   safe      ok      48   52   47   54
48  at_bottom  halt_closed pressed   a_off   safe      ok      48   49   48   51
49  at_bottom  halt_closed released  a_off   safe      ok      49   48   49   50
50  at_bottom  halt_closed released  a_off   safe      broken  50   51   50   50
51  at_bottom  halt_closed pressed   a_off   safe      broken  51   50   51   51
52  at_bottom  closing     released  a_off   safe      ok      49   47   52   53
53  at_bottom  closing     released  a_off   safe      broken  50   54   53   53
54  at_bottom  closing     pressed   a_off   safe      broken  51   53   54   54
55  to_bottom  closing     pressed   a_off   safe      broken  44   56   54   55
56  to_bottom  closing     released  a_off   safe      broken  45   55   53   56
57  past_PoNR  closing     released  a_off   no_abort  broken  58   60   56   57
58  past_PoNR  uncond_cls  released  a_off   no_abort  broken  58   59   45   58
59  past_PoNR  uncond_cls  pressed   a_off   safe      broken  59   58   44   59
60  past_PoNR  closing     pressed   a_off   safe      broken  59   57   55   60
61  above_PoNR closing     pressed   a_off   safe      broken  62   64   60   61
62  above_PoNR uncond_cls  pressed   a_off   safe      broken  62   63   59   62
63  above_PoNR uncond_cls  released  a_off   no_abort  broken  63   62   58   63
64  above_PoNR closing     released  a_off   no_abort  broken  63   61   57   64
65  at_top     closing     released  a_off   no_abort  broken  66   76   64   65
66  at_top     uncond_cls  released  a_off   no_abort  broken  67   75   63   66
67  at_top     halt_closed released  a_off   no_abort  broken  67   68   70   67
68  at_top     halt_closed pressed   a_off   safe      broken  68   67   69   68
69  above_PoNR halt_closed pressed   a_off   safe      broken  69   70   72   69
70  above_PoNR halt_closed released  a_off   no_abort  broken  70   69   71   70
71  past_PoNR  halt_closed released  a_off   no_abort  broken  71   72   74   71
72  past_PoNR  halt_closed pressed   a_off   safe      broken  72   71   73   72
73  to_bottom  halt_closed pressed   a_off   safe      broken  73   74   51   73
74  to_bottom  halt_closed released  a_off   safe      broken  74   73   50   74
75  at_top     uncond_cls  pressed   a_off   safe      broken  68   66   62   75
76  at_top     closing     pressed   a_off   safe      broken  75   65   61   76
77  at_top     ready       pressed   a_on    safe      broken  12   78   77   77
78  at_top     ready       released  a_on    safe      broken  11   77   78   78
----------------------------------------------------------------------------------
```

### Simulation Under Top Sensor Low Failure

```
=========================================================
Model check for Industrial Press Control System Top_Fail_Low
=========================================================
```

Number of states = 64

| | plunger | control | button | motor | safe | top | C.T | B.T | P.T | F.T |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | at_bottom | opening | released | a_on | safe | ok | 1 | 2 | 4 | 41 |
| 2 | at_bottom | opening | pressed | a_on | safe | ok | 2 | 1 | 3 | 40 |
| 3 | below_PoNR | opening | pressed | a_on | safe | ok | 3 | 4 | 6 | 30 |
| 4 | below_PoNR | opening | released | a_on | safe | ok | 4 | 3 | 5 | 29 |
| 5 | above_PoNR | opening | released | a_on | safe | ok | 5 | 6 | 18 | 32 |
| 6 | above_PoNR | opening | pressed | a_on | safe | ok | 6 | 5 | 7 | 31 |
| 7 | at_top | opening | pressed | a_on | safe | ok | 8 | 18 | 7 | 20 |
| 8 | at_top | open | pressed | a_on | safe | ok | 9 | 13 | 8 | 64 |
| 9 | at_top | halt_open | pressed | a_on | safe | ok | 9 | 10 | 9 | 12 |
| 10 | at_top | halt_open | released | a_on | safe | ok | 10 | 9 | 10 | 11 |
| 11 | at_top | halt_open | released | a_on | safe | broken | 11 | 12 | 11 | 11 |
| 12 | at_top | halt_open | pressed | a_on | safe | broken | 12 | 11 | 12 | 12 |
| 13 | at_top | open | released | a_on | safe | ok | 14 | 8 | 13 | 63 |
| 14 | at_top | ready | released | a_on | safe | ok | 14 | 15 | 14 | 62 |
| 15 | at_top | ready | pressed | a_on | safe | ok | 16 | 14 | 15 | 61 |
| 16 | at_top | closing | pressed | a_off | safe | ok | 16 | 17 | 22 | 60 |
| 17 | at_top | closing | released | a_off | no_abort | ok | 18 | 16 | 21 | 59 |
| 18 | at_top | opening | released | a_on | safe | ok | 13 | 7 | 18 | 19 |
| 19 | at_top | opening | released | a_on | safe | broken | 19 | 20 | 19 | 19 |
| 20 | at_top | opening | pressed | a_on | safe | broken | 20 | 19 | 20 | 20 |
| 21 | above_PoNR | closing | released | a_off | no_abort | ok | 5 | 22 | 24 | 58 |
| 22 | above_PoNR | closing | pressed | a_off | safe | ok | 22 | 21 | 23 | 57 |
| 23 | past_PoNR | closing | pressed | a_off | safe | ok | 23 | 24 | 34 | 56 |
| 24 | past_PoNR | closing | released | a_off | no_abort | ok | 25 | 23 | 33 | 55 |
| 25 | past_PoNR | opening | released | a_on | safe | ok | 25 | 26 | 4 | 28 |
| 26 | past_PoNR | opening | pressed | a_on | safe | ok | 26 | 25 | 3 | 27 |
| 27 | past_PoNR | opening | pressed | a_on | safe | broken | 27 | 28 | 30 | 27 |
| 28 | past_PoNR | opening | released | a_on | safe | broken | 28 | 27 | 29 | 28 |
| 29 | below_PoNR | opening | released | a_on | safe | broken | 29 | 30 | 32 | 29 |
| 30 | below_PoNR | opening | pressed | a_on | safe | broken | 30 | 29 | 31 | 30 |
| 31 | above_PoNR | opening | pressed | a_on | safe | broken | 31 | 32 | 20 | 31 |
| 32 | above_PoNR | opening | released | a_on | safe | broken | 32 | 31 | 19 | 32 |
| 33 | to_bottom | closing | released | a_off | safe | ok | 33 | 34 | 50 | 54 |
| 34 | to_bottom | closing | pressed | a_off | safe | ok | 35 | 33 | 45 | 53 |
| 35 | to_bottom | uncond_cls | pressed | a_off | safe | ok | 35 | 36 | 38 | 44 |
| 36 | to_bottom | uncond_cls | released | a_off | safe | ok | 36 | 35 | 37 | 43 |
| 37 | at_bottom | uncond_cls | released | a_off | safe | ok | 1 | 38 | 37 | 42 |
| 38 | at_bottom | uncond_cls | pressed | a_off | safe | ok | 2 | 37 | 38 | 39 |
| 39 | at_bottom | uncond_cls | pressed | a_off | safe | broken | 40 | 42 | 39 | 39 |
| 40 | at_bottom | opening | pressed | a_on | safe | broken | 40 | 41 | 30 | 40 |
| 41 | at_bottom | opening | released | a_on | safe | broken | 41 | 40 | 29 | 41 |
| 42 | at_bottom | uncond_cls | released | a_off | safe | broken | 41 | 39 | 42 | 42 |
| 43 | to_bottom | uncond_cls | released | a_off | safe | broken | 43 | 44 | 42 | 43 |
| 44 | to_bottom | uncond_cls | pressed | a_off | safe | broken | 44 | 43 | 39 | 44 |
| 45 | at_bottom | closing | pressed | a_off | safe | ok | 46 | 50 | 45 | 52 |
| 46 | at_bottom | halt_closed | pressed | a_off | safe | ok | 46 | 47 | 46 | 49 |
| 47 | at_bottom | halt_closed | released | a_off | safe | ok | 47 | 46 | 47 | 48 |
| 48 | at_bottom | halt_closed | released | a_off | safe | broken | 48 | 49 | 48 | 48 |
| 49 | at_bottom | halt_closed | pressed | a_off | safe | broken | 49 | 48 | 49 | 49 |
| 50 | at_bottom | closing | released | a_off | safe | ok | 47 | 45 | 50 | 51 |
| 51 | at_bottom | closing | released | a_off | safe | broken | 48 | 52 | 51 | 51 |
| 52 | at_bottom | closing | pressed | a_off | safe | broken | 49 | 51 | 52 | 52 |
| 53 | to_bottom | closing | pressed | a_off | safe | broken | 44 | 54 | 52 | 53 |
| 54 | to_bottom | closing | released | a_off | safe | broken | 54 | 53 | 51 | 54 |
| 55 | past_PoNR | closing | released | a_off | no_abort | broken | 28 | 56 | 54 | 55 |
| 56 | past_PoNR | closing | pressed | a_off | safe | broken | 56 | 55 | 53 | 56 |
| 57 | above_PoNR | closing | pressed | a_off | safe | broken | 57 | 58 | 56 | 57 |
| 58 | above_PoNR | closing | released | a_off | no_abort | broken | 32 | 57 | 55 | 58 |
| 59 | at_top | closing | released | a_off | no_abort | broken | 19 | 60 | 58 | 59 |
| 60 | at_top | closing | pressed | a_off | safe | broken | 60 | 59 | 57 | 60 |
| 61 | at_top | ready | pressed | a_on | safe | broken | 60 | 62 | 61 | 61 |
| 62 | at_top | ready | released | a_on | safe | broken | 62 | 61 | 62 | 62 |
| 63 | at_top | open | released | a_on | safe | broken | 62 | 64 | 63 | 63 |
| 64 | at_top | open | pressed | a_on | safe | broken | 12 | 63 | 64 | 64 |

## Simulation Under Top Sensor High Failure

```
=======================================================
Model check for Industrial Press Control System Top_Fail_High
=======================================================
Number of states = 88
```

| | plunger | control | button | motor | safe | top | C.T | B.T | P.T | F.T |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | at_bottom | opening | released | a_on | safe | ok | 1 | 2 | 4 | 88 |
| 2 | at_bottom | opening | pressed | a_on | safe | ok | 2 | 1 | 3 | 83 |
| 3 | below_PoNR | opening | pressed | a_on | safe | ok | 3 | 4 | 6 | 51 |
| 4 | below_PoNR | opening | released | a_on | safe | ok | 4 | 3 | 5 | 52 |
| 5 | above_PoNR | opening | released | a_on | safe | ok | 5 | 6 | 18 | 26 |
| 6 | above_PoNR | opening | pressed | a_on | safe | ok | 6 | 5 | 7 | 53 |
| 7 | at_top | opening | pressed | a_on | safe | ok | 8 | 18 | 7 | 54 |
| 8 | at_top | open | pressed | a_on | safe | ok | 9 | 13 | 8 | 49 |
| 9 | at_top | halt_open | pressed | a_on | safe | ok | 9 | 10 | 9 | 12 |
| 10 | at_top | halt_open | released | a_on | safe | ok | 10 | 9 | 10 | 11 |
| 11 | at_top | halt_open | released | a_on | safe | broken | 11 | 12 | 11 | 11 |
| 12 | at_top | halt_open | pressed | a_on | safe | broken | 12 | 11 | 12 | 12 |
| 13 | at_top | open | released | a_on | safe | ok | 14 | 8 | 13 | 20 |
| 14 | at_top | ready | released | a_on | safe | ok | 14 | 15 | 14 | 21 |
| 15 | at_top | ready | pressed | a_on | safe | ok | 16 | 14 | 15 | 22 |
| 16 | at_top | closing | pressed | a_off | safe | ok | 16 | 17 | 68 | 23 |
| 17 | at_top | closing | released | a_off | no_abort | ok | 18 | 16 | 67 | 24 |
| 18 | at_top | opening | released | a_on | safe | ok | 13 | 7 | 18 | 19 |
| 19 | at_top | opening | released | a_on | safe | broken | 20 | 54 | 19 | 19 |
| 20 | at_top | open | released | a_on | safe | broken | 21 | 49 | 20 | 20 |
| 21 | at_top | ready | released | a_on | safe | broken | 21 | 22 | 21 | 21 |
| 22 | at_top | ready | pressed | a_on | safe | broken | 23 | 21 | 22 | 22 |
| 23 | at_top | closing | pressed | a_off | safe | broken | 23 | 24 | 30 | 23 |
| 24 | at_top | closing | released | a_off | no_abort | broken | 19 | 23 | 25 | 24 |
| 25 | above_PoNR | closing | released | a_off | no_abort | broken | 26 | 30 | 32 | 25 |
| 26 | above_PoNR | opening | released | a_on | safe | broken | 27 | 53 | 19 | 26 |
| 27 | above_PoNR | open | released | a_on | safe | broken | 28 | 48 | 20 | 27 |
| 28 | above_PoNR | ready | released | a_on | safe | broken | 28 | 29 | 21 | 28 |
| 29 | above_PoNR | ready | pressed | a_on | safe | broken | 30 | 28 | 22 | 29 |
| 30 | above_PoNR | closing | pressed | a_off | safe | broken | 30 | 25 | 31 | 30 |
| 31 | past_PoNR | closing | pressed | a_off | safe | broken | 31 | 32 | 64 | 31 |
| 32 | past_PoNR | closing | released | a_off | no_abort | broken | 33 | 31 | 55 | 32 |
| 33 | past_PoNR | opening | released | a_on | safe | broken | 34 | 50 | 52 | 33 |
| 34 | past_PoNR | open | released | a_on | safe | broken | 35 | 43 | 47 | 34 |
| 35 | past_PoNR | ready | released | a_on | safe | broken | 35 | 36 | 42 | 35 |
| 36 | past_PoNR | ready | pressed | a_on | safe | broken | 31 | 35 | 37 | 36 |
| 37 | below_PoNR | ready | pressed | a_on | safe | broken | 38 | 42 | 29 | 37 |
| 38 | below_PoNR | halt_open | pressed | a_on | safe | broken | 38 | 39 | 41 | 38 |
| 39 | below_PoNR | halt_open | released | a_on | safe | broken | 39 | 38 | 40 | 39 |
| 40 | above_PoNR | halt_open | released | a_on | safe | broken | 40 | 41 | 11 | 40 |
| 41 | above_PoNR | halt_open | pressed | a_on | safe | broken | 41 | 40 | 12 | 41 |
| 42 | below_PoNR | ready | released | a_on | safe | broken | 39 | 37 | 28 | 42 |
| 43 | past_PoNR | open | pressed | a_on | safe | broken | 44 | 34 | 46 | 43 |
| 44 | past_PoNR | halt_open | pressed | a_on | safe | broken | 44 | 45 | 38 | 44 |
| 45 | past_PoNR | halt_open | released | a_on | safe | broken | 45 | 44 | 39 | 45 |
| 46 | below_PoNR | open | pressed | a_on | safe | broken | 38 | 47 | 48 | 46 |
| 47 | below_PoNR | open | released | a_on | safe | broken | 39 | 46 | 27 | 47 |
| 48 | above_PoNR | open | pressed | a_on | safe | broken | 41 | 27 | 49 | 48 |
| 49 | at_top | open | pressed | a_on | safe | broken | 12 | 20 | 49 | 49 |
| 50 | past_PoNR | opening | pressed | a_on | safe | broken | 43 | 33 | 51 | 50 |
| 51 | below_PoNR | opening | pressed | a_on | safe | broken | 46 | 52 | 53 | 51 |
| 52 | below_PoNR | opening | released | a_on | safe | broken | 47 | 51 | 26 | 52 |
| 53 | above_PoNR | opening | pressed | a_on | safe | broken | 48 | 26 | 54 | 53 |
| 54 | at_top | opening | pressed | a_on | safe | broken | 49 | 19 | 54 | 54 |
| 55 | to_bottom | closing | released | a_off | safe | broken | 56 | 64 | 66 | 55 |
| 56 | to_bottom | uncond_cls | released | a_off | safe | broken | 57 | 61 | 63 | 56 |
| 57 | to_bottom | halt_closed | released | a_off | safe | broken | 57 | 58 | 60 | 57 |
| 58 | to_bottom | halt_closed | pressed | a_off | safe | broken | 58 | 57 | 59 | 58 |
| 59 | at_bottom | halt_closed | pressed | a_off | safe | broken | 59 | 60 | 59 | 59 |
| 60 | at_bottom | halt_closed | released | a_off | safe | broken | 60 | 59 | 60 | 60 |
| 61 | to_bottom | uncond_cls | pressed | a_off | safe | broken | 58 | 56 | 62 | 61 |
| 62 | at_bottom | uncond_cls | pressed | a_off | safe | broken | 59 | 63 | 62 | 62 |
| 63 | at_bottom | uncond_cls | released | a_off | safe | broken | 60 | 62 | 63 | 63 |
| 64 | to_bottom | closing | pressed | a_off | safe | broken | 61 | 55 | 65 | 64 |
| 65 | at_bottom | closing | pressed | a_off | safe | broken | 59 | 66 | 65 | 65 |
| 66 | at_bottom | closing | released | a_off | safe | broken | 60 | 65 | 66 | 66 |
| 67 | above_PoNR | closing | released | a_off | no_abort | ok | 5 | 68 | 70 | 25 |
| 68 | above_PoNR | closing | pressed | a_off | safe | ok | 68 | 67 | 69 | 30 |
| 69 | past_PoNR | closing | pressed | a_off | safe | ok | 69 | 70 | 78 | 31 |
| 70 | past_PoNR | closing | released | a_off | no_abort | ok | 71 | 69 | 73 | 32 |
| 71 | past_PoNR | opening | released | a_on | safe | ok | 71 | 72 | 4 | 33 |
| 72 | past_PoNR | opening | pressed | a_on | safe | ok | 72 | 71 | 3 | 50 |
| 73 | to_bottom | closing | released | a_off | safe | ok | 74 | 78 | 82 | 55 |
| 74 | to_bottom | uncond_cls | released | a_off | safe | ok | 74 | 75 | 77 | 56 |

```
75  to_bottom  uncond_cls  pressed   a_off    safe      ok      75   74   76   61
76  at_bottom  uncond_cls  pressed   a_off    safe      ok      2    77   76   62
77  at_bottom  uncond_cls  released  a_off    safe      ok      1    76   77   63
78  to_bottom  closing     pressed   a_off    safe      ok      75   73   79   64
79  at_bottom  closing     pressed   a_off    safe      ok      80   82   79   65
80  at_bottom  halt_closed pressed   a_off    safe      ok      80   81   80   59
81  at_bottom  halt_closed released  a_off    safe      ok      81   80   81   60
82  at_bottom  closing     released  a_off    safe      ok      81   79   82   66
83  at_bottom  opening     pressed   a_on     safe      broken  84   88   51   83
84  at_bottom  open        pressed   a_on     safe      broken  85   87   46   84
85  at_bottom  halt_open   pressed   a_on     safe      broken  85   86   38   85
86  at_bottom  halt_open   released  a_on     safe      broken  86   85   39   86
87  at_bottom  open        released  a_on     safe      broken  86   84   47   87
88  at_bottom  opening     released  a_on     safe      broken  87   83   52   88
----------------------------------------------------------------------------------
```

## Simulation Under Button Sensor Low Failure

```
=========================================================
Model check for Industrial Press Control System Button_Fail_Low
=========================================================
Number of states = 64
----------------------------------------------------------------------------------
    plunger    control     button    motor    safe      button  C.T  B.T  P.T  F.T
----------------------------------------------------------------------------------
1   at_bottom  opening     released  a_on     safe      ok      1    2    4    43
2   at_bottom  opening     pressed   a_on     safe      ok      2    1    3    44
3   below_PoNR opening     pressed   a_on     safe      ok      3    4    6    34
4   below_PoNR opening     released  a_on     safe      ok      4    3    5    33
5   above_PoNR opening     released  a_on     safe      ok      5    6    18   36
6   above_PoNR opening     pressed   a_on     safe      ok      6    5    7    35
7   at_top     opening     pressed   a_on     safe      ok      8    18   7    24
8   at_top     open        pressed   a_on     safe      ok      9    13   8    23
9   at_top     halt_open   pressed   a_on     safe      ok      9    10   9    12
10  at_top     halt_open   released  a_on     safe      ok      10   9    10   11
11  at_top     halt_open   released  a_on     safe      broken  11   12   11   11
12  at_top     halt_open   pressed   a_on     safe      broken  12   11   12   12
13  at_top     open        released  a_on     safe      ok      14   8    13   20
14  at_top     ready       released  a_on     safe      ok      14   15   14   21
15  at_top     ready       pressed   a_on     safe      ok      16   14   15   22
16  at_top     closing     pressed   a_off    safe      ok      16   17   26   64
17  at_top     closing     released  a_off    no_abort  ok      18   16   25   63
18  at_top     opening     released  a_on     safe      ok      13   7    18   19
19  at_top     opening     released  a_on     safe      broken  20   24   19   19
20  at_top     open        released  a_on     safe      broken  21   23   20   20
21  at_top     ready       released  a_on     safe      broken  21   22   21   21
22  at_top     ready       pressed   a_on     safe      broken  22   21   22   22
23  at_top     open        pressed   a_on     safe      broken  22   20   23   23
24  at_top     opening     pressed   a_on     safe      broken  23   19   24   24
25  above_PoNR closing     released  a_off    no_abort  ok      5    26   28   62
26  above_PoNR closing     pressed   a_off    safe      ok      26   25   27   61
27  past_PoNR  closing     pressed   a_off    safe      ok      27   28   48   60
28  past_PoNR  closing     released  a_off    no_abort  ok      29   27   37   59
29  past_PoNR  opening     released  a_on     safe      ok      29   30   4    32
30  past_PoNR  opening     pressed   a_on     safe      ok      30   29   3    31
31  past_PoNR  opening     pressed   a_on     safe      broken  31   32   34   31
32  past_PoNR  opening     released  a_on     safe      broken  32   31   33   32
33  below_PoNR opening     released  a_on     safe      broken  33   34   36   33
34  below_PoNR opening     pressed   a_on     safe      broken  34   33   35   34
35  above_PoNR opening     pressed   a_on     safe      broken  35   36   24   35
36  above_PoNR opening     released  a_on     safe      broken  36   35   19   36
37  to_bottom  closing     released  a_off    safe      ok      38   48   54   58
38  to_bottom  uncond_cls  released  a_off    safe      ok      38   39   41   47
39  to_bottom  uncond_cls  pressed   a_off    safe      ok      39   38   40   46
40  at_bottom  uncond_cls  pressed   a_off    safe      ok      2    41   40   45
41  at_bottom  uncond_cls  released  a_off    safe      ok      1    40   41   42
42  at_bottom  uncond_cls  released  a_off    safe      broken  43   45   42   42
43  at_bottom  opening     released  a_on     safe      broken  43   44   33   43
44  at_bottom  opening     pressed   a_on     safe      broken  44   43   34   44
45  at_bottom  uncond_cls  pressed   a_off    safe      broken  44   42   45   45
46  to_bottom  uncond_cls  pressed   a_off    safe      broken  46   47   45   46
47  to_bottom  uncond_cls  released  a_off    safe      broken  47   46   42   47
48  to_bottom  closing     pressed   a_off    safe      ok      39   37   49   57
49  at_bottom  closing     pressed   a_off    safe      ok      50   54   49   56
50  at_bottom  halt_closed pressed   a_off    safe      ok      50   51   50   53
51  at_bottom  halt_closed released  a_off    safe      ok      51   50   51   52
```

```
52  at_bottom  halt_closed released  a_off    safe      broken  52   53   52   52
53  at_bottom  halt_closed pressed   a_off    safe      broken  53   52   53   53
54  at_bottom  closing     released  a_off    safe      ok      51   49   54   55
55  at_bottom  closing     released  a_off    safe      broken  52   56   55   55
56  at_bottom  closing     pressed   a_off    safe      broken  53   55   56   56
57  to_bottom  closing     pressed   a_off    safe      broken  46   58   56   57
58  to_bottom  closing     released  a_off    safe      broken  47   57   55   58
59  past_PoNR  closing     released  a_off    no_abort  broken  32   60   58   59
60  past_PoNR  closing     pressed   a_off    safe      broken  31   59   57   60
61  above_PoNR closing     pressed   a_off    safe      broken  35   62   60   61
62  above_PoNR closing     released  a_off    no_abort  broken  36   61   59   62
63  at_top     closing     released  a_off    no_abort  broken  19   64   62   63
64  at_top     closing     pressed   a_off    safe      broken  24   63   61   64
-----------------------------------------------------------------------------------
```

## Simulate Button Sensor High Failure

```
=========================================================
Model check for Industrial Press Control System Button_Fail_High
=========================================================
Number of states = 64
-----------------------------------------------------------------------------------
    plunger    control     button    motor    safe      button  C.T  B.T  P.T  F.T
-----------------------------------------------------------------------------------
1   at_bottom  opening     released  a_on     safe      ok      1    2    4    41
2   at_bottom  opening     pressed   a_on     safe      ok      2    1    3    42
3   below_PoNR opening     pressed   a_on     safe      ok      3    4    6    32
4   below_PoNR opening     released  a_on     safe      ok      4    3    5    31
5   above_PoNR opening     released  a_on     safe      ok      5    6    18   34
6   above_PoNR opening     pressed   a_on     safe      ok      6    5    7    33
7   at_top     opening     pressed   a_on     safe      ok      8    18   7    22
8   at_top     open        pressed   a_on     safe      ok      9    13   8    21
9   at_top     halt_open   pressed   a_on     safe      ok      9    10   9    12
10  at_top     halt_open   released  a_on     safe      ok      10   9    10   11
11  at_top     halt_open   released  a_on     safe      broken  11   12   11   11
12  at_top     halt_open   pressed   a_on     safe      broken  12   11   12   12
13  at_top     open        released  a_on     safe      ok      14   8    13   20
14  at_top     ready       released  a_on     safe      ok      14   15   14   64
15  at_top     ready       pressed   a_on     safe      ok      16   14   15   63
16  at_top     closing     pressed   a_off    safe      ok      16   17   24   62
17  at_top     closing     released  a_off    no_abort  ok      18   16   23   61
18  at_top     opening     released  a_on     safe      ok      13   7    18   19
19  at_top     opening     released  a_on     safe      broken  20   22   19   19
20  at_top     open        released  a_on     safe      broken  11   21   20   20
21  at_top     open        pressed   a_on     safe      broken  12   20   21   21
22  at_top     opening     pressed   a_on     safe      broken  21   19   22   22
23  above_PoNR closing     released  a_off    no_abort  ok      5    24   26   60
24  above_PoNR closing     pressed   a_off    safe      ok      24   23   25   59
25  past_PoNR  closing     pressed   a_off    safe      ok      25   26   46   58
26  past_PoNR  closing     released  a_off    no_abort  ok      27   25   35   57
27  past_PoNR  opening     released  a_on     safe      ok      27   28   4    30
28  past_PoNR  opening     pressed   a_on     safe      ok      28   27   3    29
29  past_PoNR  opening     pressed   a_on     safe      broken  29   30   32   29
30  past_PoNR  opening     released  a_on     safe      broken  30   29   31   30
31  below_PoNR opening     released  a_on     safe      broken  31   32   34   31
32  below_PoNR opening     pressed   a_on     safe      broken  32   31   33   32
33  above_PoNR opening     pressed   a_on     safe      broken  33   34   22   33
34  above_PoNR opening     released  a_on     safe      broken  34   33   19   34
35  to_bottom  closing     released  a_off    safe      ok      36   46   52   56
36  to_bottom  uncond_cls  released  a_off    safe      ok      36   37   39   45
37  to_bottom  uncond_cls  pressed   a_off    safe      ok      37   36   38   44
38  at_bottom  uncond_cls  pressed   a_off    safe      ok      2    39   38   43
39  at_bottom  uncond_cls  released  a_off    safe      ok      1    38   39   40
40  at_bottom  uncond_cls  released  a_off    safe      broken  41   43   40   40
41  at_bottom  opening     released  a_on     safe      broken  41   42   31   41
42  at_bottom  opening     pressed   a_on     safe      broken  42   41   32   42
43  at_bottom  uncond_cls  pressed   a_off    safe      broken  42   40   43   43
44  to_bottom  uncond_cls  pressed   a_off    safe      broken  44   45   43   44
45  to_bottom  uncond_cls  released  a_off    safe      broken  45   44   40   45
46  to_bottom  closing     pressed   a_off    safe      ok      37   35   47   55
47  at_bottom  closing     pressed   a_off    safe      ok      48   52   47   54
48  at_bottom  halt_closed pressed   a_off    safe      ok      48   49   48   51
49  at_bottom  halt_closed released  a_off    safe      ok      49   48   49   50
50  at_bottom  halt_closed released  a_off    safe      broken  50   51   50   50
51  at_bottom  halt_closed pressed   a_off    safe      broken  51   50   51   51
52  at_bottom  closing     released  a_off    safe      ok      49   47   52   53
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 53 | at_bottom | closing | released | a_off | safe | broken | 50 | 54 | 53 | 53 |
| 54 | at_bottom | closing | pressed | a_off | safe | broken | 51 | 53 | 54 | 54 |
| 55 | to_bottom | closing | pressed | a_off | safe | broken | 44 | 56 | 54 | 55 |
| 56 | to_bottom | closing | released | a_off | safe | broken | 45 | 55 | 53 | 56 |
| 57 | past_PoNR | closing | released | a_off | no_abort | broken | 57 | 58 | 56 | 57 |
| 58 | past_PoNR | closing | pressed | a_off | safe | broken | 58 | 57 | 55 | 58 |
| 59 | above_PoNR | closing | pressed | a_off | safe | broken | 59 | 60 | 58 | 59 |
| 60 | above_PoNR | closing | released | a_off | no_abort | broken | 60 | 59 | 57 | 60 |
| 61 | at_top | closing | released | a_off | no_abort | broken | 61 | 62 | 60 | 61 |
| 62 | at_top | closing | pressed | a_off | safe | broken | 62 | 61 | 59 | 62 |
| 63 | at_top | ready | pressed | a_on | safe | broken | 62 | 64 | 63 | 63 |
| 64 | at_top | ready | released | a_on | safe | broken | 61 | 63 | 64 | 64 |

# Appendix D Spark Ada Code

There are seven Ada files altogether.

1. specification of **package** Sensors†
2. **package body** Sensors
3. specification of **package** Actuator†
4. **package body** Actuator
5. specification of **package** Transitions†
6. **package body** Transitions†
7. main **procedure** Machine†

All seven have been checked for syntax and semantics by the GNAT Ada compiler, and five (marked †) have been analysed by the SPARK Examiner. Those five are included here in the form of a numbered listing generated by SPARK or the original source where no listing is available. The remaining two are implementatiuon dependent and have not been checked by SPARK. Their listing here is the original source text.

1. File `sensors.lst`† - contains a specification of **package** Sensors.

```
          *******************************************************
                           Listing of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 2.5 / 04.97
                           Demonstration Version
          *******************************************************


                       DATE : 12-SEP-1999 12:19:08.03

Line
   1  package Sensors
   2    --# own State_Seq;
   3    --# initializes State_Seq;
   4  is
   5
   6     type In_Sig is (high, low);
   7
   8     type State is record
   9     top   : In_Sig;
  10     PoNR  : In_Sig;
  11     bottom   : In_Sig;
  12     button   : In_Sig;
  13     end record;
  14
  15     procedure Read (Value: out State);
  16     --# global State_Seq;
  17     --# derives Value, State_Seq from State_Seq;
  18
  19  end Sensors;


--End of file--------------------------------------------------
```

2. File `sensors.adb` - contains **package body** Sensors

```
with System.Storage_Elements;

package body Sensors
  --# own State_Seq is Sensor_Register, Sensor_Error;
is
   type Sensor_Value is (lo, fault_1, fault_2, hi);
   for Sensor_Value use (lo => 0, fault_1 => 1, fault_2 => 2, hi => 3);

   type Local_Sensors_State is record
   SV_Top   : Sensor_Value;
   SV_PNR   : Sensor_Value;
   SV_Bottom   : Sensor_Value;
   SV_Button   : Sensor_Value;
```

```
   end record;

   for Local_Sensors_State use record
      SV_Top at 0 range 0..1;
      SV_PNR at 0 range 2..3;
      SV_Bottom at 0 range 4..5;
      SV_Button at 0 range 6..7;
   end record;

   for Local_Sensors_State'Size use 8;    -- fits in a single byte
   for Local_Sensors_State'Alignment use 1;  -- byte aligned
   for Local_Sensors_State'Bit_Order use System.High_Order_First;-- big-endian machine

   Sensor_Register : Local_Sensors_State;
   SR_Address : constant := 16#100001#; -- 16Mb addressable RAM, 000000..ffffff
   for Sensor_Register'Address use System.Storage_Elements.To_Address (SR_Address);
   pragma Volatile (Sensor_Register);
   -- machine has 16Mb addressable RAM, 000000..ffffff
   -- hardware sensor is connected to address 100001
   -- hardware is writing to this location continuously

   Sensor_Error : Boolean := False;
   -- if ever a sensor error is detected,
   -- the read routine delivers a 'high' on all channels forever.
   -- this will cause the press machine to halt within one cycle.

   procedure Read1 (X : in Sensor_Value; Y : out In_Sig) is
   --# global out Sensor_Error
   begin
      if X = hi then Y := high;
      elsif X = lo then Y := low;
      else Sensor_Error := True; Y:= high;
      end if;
   end Read1;

   procedure Read (Value: out State) is
      Sensor_Temp : Local_Sensors_State;
   begin
      if Sensor_Error then
      Value := (high, high, high, high);
         return;
      end if;

      Sensor_Temp := Sensor_Register; -- atomic assignment to ensure a stable value

      Read1 (Sensor_Temp.SV_Top, Value.top);
      Read1 (Sensor_Temp.SV_PNR, Value.PoNR);
      Read1 (Sensor_Temp.SV_Bottom, Value.bottom);
      Read1 (Sensor_Temp.SV_Button, Value.button);
   end Read;

begin
      Sensor_Error := False;
end Sensors;
```

3. File actuator.`lst`[†] - contains specification of **package** Actuator

```
          *******************************************************
                           Listing of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 2.5 / 04.97
                           Demonstration Version
          *******************************************************


                       DATE : 12-SEP-1999 12:19:07.92

Line
   1  package Actuator
   2    --# own State_Seq;
   3    --# initializes State_Seq;
   4  is
   5
   6     type Out_Sig is (a_on, a_off);
   7
```

```
   8     procedure Write   (Value: in Out_Sig);
   9     --# global State_Seq;
  10     --# derives State_Seq from Value, State_Seq;
  11
  12  end Actuator;


--End of file--------------------------------------------------
```

## 4. File actuator.adb - contains **package body** Actuator

```
with System.Storage_Elements;

package body Actuator
  --# own State_Seq is Drive_Register;
is

   type Drive_Value is range 0..(2**8-1);
   for Drive_Value'Size use 8;   -- fits in a single byte
   for Drive_Value'Alignment use 1; -- byte aligned

   Drive_ON : constant Drive_Value := 2#1111_1111#;
   Drive_OFF : constant Drive_Value := 2#0000_0000#;

   Drive_Register : Drive_Value;
   Drive_Address : constant := 16#100011#; -- 16Mb addressable RAM, 000000..ffffff
   for Drive_Register'Address use System.Storage_Elements.To_Address (Drive_Address);
   pragma Volatile (Drive_Register);
   -- machine has 16Mb addressable RAM, 000000..ffffff
   -- hardware motor drive is connected to address 100011
   -- hardware is reading from this location continuously

   procedure Write   (Value: in Out_Sig) is
   begin
      case Value is
      when on =>  Drive_Register := Drive_ON;
      when off => Drive_Register := Drive_OFF;
   end case;
   end Write;

end Actuator;
```

## 5. File transitions.ada† - contains specification of **package** Transitions

```
with Sensors;
use type Sensors.In_Sig;

with Actuator;
--# inherit Sensors, Actuator;

package Transitions is

  type Control_Type is (opening, open, ready, closing, uncond_closing, halt_open,
halt_closed);

  -- abbreviations to make the postconditions more readable

  high   : constant Sensors.In_Sig := Sensors.high;
  low : constant Sensors.In_Sig := Sensors.low;

  a_on   : constant Actuator.Out_Sig := Actuator.a_on;
  a_off: constant Actuator.Out_Sig := Actuator.a_off;


  procedure Init  (control : out Control_Type;
          motor      : out Actuator.Out_Sig);
  --# derives control, motor from ;
  --# post  (control = opening and motor = a_on);


  procedure At_Opening  (sensors_state: in Sensors.State;
          control: in out Control_Type;
          motor   : out Actuator.Out_Sig);
  --# derives control from control, sensors_state
  --#    & motor from ;
```

```
--# pre control = opening;
--# post  (sensors_state.top = high
--#      -> (control = open and motor = a_on))
--# and   (not (sensors_state.top = high)
--#      -> (control = opening and motor = a_on));


procedure At_Open  (sensors_state : in Sensors.State;
        control : in out Control_Type;
        motor      : out Actuator.Out_Sig);
--# derives control from control, sensors_state
--#    & motor from ;

--# pre control = open;
--# post  ((sensors_state.bottom = high or
--#      sensors_state.PoNR = high or
--#      sensors_state.button = high)
--#         -> (control = halt_open and motor = a_on))
--#    and   (not(sensors_state.bottom = high or
--#         sensors_state.PoNR = high or
--#         sensors_state.button = high)
--#      -> (control = ready and motor = a_on));


procedure At_Ready (sensors_state : in Sensors.State;
        control : in out Control_Type;
        motor      : out Actuator.Out_Sig);
--# derives control from control, sensors_state
--#    & motor from sensors_state;

--# pre control = ready;
--# post  ((sensors_state.bottom = high or
--#      sensors_state.PoNR = high)
--#      -> (control = halt_open and motor = a_on))
--#  and  ((not (sensors_state.bottom = high or
--#          sensors_state.PoNR = high)
--#     and (sensors_state.button = high))
--#      -> (control = closing and motor = a_off))
--#  and  ((not (sensors_state.bottom = high or
--#              sensors_state.PoNR = high)
--#     and (not (sensors_state.button = high)))
--#      -> (control = ready and motor = a_on));


procedure At_Closing  (sensors_state : in Sensors.State;
        control : in out Control_Type;
        motor      : out Actuator.Out_Sig);
--# derives control from control, sensors_state
--#    & motor from sensors_state;

--# pre control = closing;
--# post  (sensors_state.bottom = high
--#      -> (control = halt_closed and motor = a_off))
--#  and  ((not (sensors_state.bottom = high)
--#     and (sensors_state.PoNR = high))
--#      -> (control = uncond_closing and motor = a_off))
--#  and  ((not (sensors_state.bottom = high or
--#          sensors_state.PoNR = high)
--#            and (sensors_state.button /= high))
--#      -> (control = opening and motor = a_on))
--#  and  (not (sensors_state.bottom = high or
--#          sensors_state.PoNR = high or
--#            sensors_state.button /=high)
--#      -> (control = closing and motor = a_off));


procedure At_Uncond_Closing (sensors_state : in Sensors.State;
           control : in out Control_Type;
           motor      : out Actuator.Out_Sig);
--# derives control from control, sensors_state
```

```
  --#    & motor from sensors_state;

  --# pre control = uncond_closing;
  --# post  (sensors_state.top = high
  --#      -> (control = halt_closed and motor = a_off))
  --#  and  ((not (sensors_state.top = high)
  --#     and (sensors_state.bottom= high))
  --#      -> (control = opening and motor = a_on))
  --#  and  ((not (sensors_state.top = high or
  --#           sensors_state.bottom = high))
  --#      -> (control = uncond_closing and motor = a_off));

end Transitions;
```

## 6. File transitions.`lst`[†] – contains **package body** Transitions

```
          *******************************************************
                          Listing of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 2.5 / 04.97
                          Demonstration Version
          *******************************************************


                       DATE : 12-SEP-1999 12:19:10.01

Line
   1  package body Transitions
   2  is
   3    procedure Init  (control : out Control_Type;
   4            motor      : out Actuator.Out_Sig) is
   5    begin
   6      control := opening;
   7      motor := a_on;
   8    end Init;

+++        Flow analysis of subprogram Init performed: no
           errors found.

   9
  10
  11    procedure At_Opening  (sensors_state: in Sensors.State;
  12            control: in out Control_Type;
  13            motor   : out Actuator.Out_Sig) is
  14    begin
  15       if sensors_state.top = high then
  16          control := open;
  17       end if;
  18       motor := a_on;
  19    end At_Opening;

+++        Flow analysis of subprogram At_Opening
           performed: no errors found.

  20
  21    procedure At_Open  (sensors_state : in Sensors.State;
  22            control : in out Control_Type;
  23            motor      : out Actuator.Out_Sig) is
  24    begin
  25       if (sensors_state.bottom = high) or
  26          (sensors_state.PoNR = high) or
  27          (sensors_state.button = high) then
  28          control := halt_open;
  29       else
  30          control := ready;
  31       end if;
  32       motor := a_on;
  33    end At_Open;

!!! (  1)  Flow Error        : Importation of the initial value of variable
           control is ineffective.
!!! (  2)  Flow Error        : The imported value of control is not used in the
           derivation of control.
```

```
  34
  35    procedure At_Ready (sensors_state : in Sensors.State;
  36            control : in out Control_Type;
  37            motor      : out Actuator.Out_Sig) is
  38    begin
  39       if (sensors_state.bottom = high) or
  40          (sensors_state.PoNR = high) then
  41          control := halt_open;
  42          motor := a_on;
  43       elsif sensors_state.button = high then
  44          control := closing;
  45          motor := a_off;
  46       else motor := a_on;
  47       end if;
  48    end At_Ready;

+++        Flow analysis of subprogram At_Ready performed:
           no errors found.

  49
  50    procedure At_Closing  (sensors_state : in Sensors.State;
  51            control : in out Control_Type;
  52            motor      : out Actuator.Out_Sig) is
  53    begin
  54       if sensors_state.bottom = high then
  55          control := halt_closed;
  56          motor := a_off;
  57       elsif sensors_state.PoNR = high then
  58          control := uncond_closing;
  59          motor := a_off;
  60       elsif sensors_state.button /= high then
  61          control := opening;
  62          motor := a_on;
  63       else
  64          motor := a_off;
  65       end if;
  66    end At_Closing;

+++        Flow analysis of subprogram At_Closing
           performed: no errors found.

  67
  68    procedure At_Uncond_Closing (sensors_state : in Sensors.State;
  69               control : in out Control_Type;
  70               motor      : out Actuator.Out_Sig) is
  71    begin
  72       if sensors_state.top = high then
  73          control := halt_closed;
  74          motor := a_off;
  75       elsif sensors_state.bottom = high then
  76          control := opening;
  77          motor := a_on;
  78       else motor := a_off;
  79       end if;
  80    end At_Uncond_Closing;

+++        Flow analysis of subprogram At_Uncond_Closing
           performed: no errors found.

  81
  82  end Transitions;


--End of file--------------------------------------------------
```

7. File machine.`lst`[†] - contains main **procedure** Machine

```
       *******************************************************
                         Listing of SPARK Text
    SPARK95 Examiner with VC and RTC Generator Release 2.5 / 04.97
                         Demonstration Version
       *******************************************************


                     DATE : 12-SEP-1999 12:19:09.02
```

```
Line
   1  with Transitions; use type Transitions.Control_Type;
   2  with Sensors;
   3  with Actuator;
   4
   5  --# inherit Transitions, Sensors, Actuator;
   6  --# main_program;
   7
   8  procedure Machine
   9  --# global in out Sensors.State_Seq;
  10  --#        in out Actuator.State_Seq;
  11  --# derives Sensors.State_Seq from *
  12  --#       & Actuator.State_Seq from *, Sensors.State_Seq;
  13  is
  14
  15     control : Transitions.Control_Type;
  16     sensors_state : Sensors.State;
  17     motor : Actuator.Out_Sig;
  18
  19  begin
  20     Transitions.Init (control, motor);
  21     Actuator.Write (motor);
  22
  23     while (control /= Transitions.halt_open)
  24      and then (control /= Transitions.halt_closed)
  25     --# assert true;  -- SPARK demands a loop invariant
  26
  27     loop
  28        Sensors.Read (sensors_state);
  29
  30        case control is
  31           when Transitions.opening =>
  32        Transitions.At_Opening (sensors_state, control, motor);
  33           when Transitions.open =>
  34        Transitions.At_Open (sensors_state, control, motor);
  35           when Transitions.ready =>
  36        Transitions.At_Ready (sensors_state, control, motor);
  37           when Transitions.closing =>
  38        Transitions.At_Closing (sensors_state, control, motor);
  39           when Transitions.uncond_closing =>
  40        Transitions.At_Uncond_Closing (sensors_state, control, motor);
  41     when Transitions.halt_open | Transitions.halt_closed =>
  42        null; --# check false;  -- means that this path can never be taken
  43        end case;
  44
  45        Actuator.Write (motor);
  46     end loop;
  47
  48     --# assert (control = Transitions.halt_open) or (control = Transitions.halt_closed);
  49
  50  end Machine;

+++       Flow analysis of subprogram Machine performed:
          no errors found.


--End of file--------------------------------------------------
```

# Appendix E Spade Simplifier Output

The following SPADE outputs are provided:

1. 7 files of generated VCs

   `init.vcg, at_openi.vcg, at_open.vcg, at_ready.vcg, at_closi.vcg, at_uncon.vcg, machine.vcg`

File `at_closi.vcg` shown here.

```
          *******************************************************
                       Semantic Analysis of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 2.5 / 04.97
                            Demonstration Version
          *******************************************************


                        DATE : 12-SEP-1999 12:19:09.84

                       procedure Transitions.At_Closing




For path(s) from start to finish:

procedure_at_closing_1.
H1:    control = closing .
H2:    fld_bottom(sensors_state) = high .
        ->
C1:    (fld_bottom(sensors_state) = high) -> ((halt_closed =
           halt_closed) and (a_off = a_off)) .
C2:    ((not (fld_bottom(sensors_state) = high)) and (fld_ponr(
           sensors_state) = high)) -> ((halt_closed =
           uncond_closing) and (a_off = a_off)) .
C3:    ((not ((fld_bottom(sensors_state) = high) or (fld_ponr(
           sensors_state) = high))) and (fld_button(
           sensors_state) <> high)) -> ((halt_closed =
           opening) and (a_off = a_on)) .
C4:    (not ((fld_bottom(sensors_state) = high) or ((fld_ponr(
           sensors_state) = high) or (fld_button(
           sensors_state) <> high)))) -> ((halt_closed =
           closing) and (a_off = a_off)) .


procedure_at_closing_2.
H1:    control = closing .
H2:    not (fld_bottom(sensors_state) = high) .
H3:    fld_ponr(sensors_state) = high .
        ->
C1:    (fld_bottom(sensors_state) = high) -> ((
           uncond_closing = halt_closed) and (a_off = a_off)) .
C2:    ((not (fld_bottom(sensors_state) = high)) and (fld_ponr(
           sensors_state) = high)) -> ((uncond_closing =
           uncond_closing) and (a_off = a_off)) .
C3:    ((not ((fld_bottom(sensors_state) = high) or (fld_ponr(
           sensors_state) = high))) and (fld_button(
           sensors_state) <> high)) -> ((uncond_closing =
           opening) and (a_off = a_on)) .
C4:    (not ((fld_bottom(sensors_state) = high) or ((fld_ponr(
           sensors_state) = high) or (fld_button(
           sensors_state) <> high)))) -> ((uncond_closing =
           closing) and (a_off = a_off)) .


procedure_at_closing_3.
H1:    control = closing .
H2:    not (fld_bottom(sensors_state) = high) .
H3:    not (fld_ponr(sensors_state) = high) .
H4:    fld_button(sensors_state) <> high .
        ->
C1:    (fld_bottom(sensors_state) = high) -> ((opening =
           halt_closed) and (a_on = a_off)) .
C2:    ((not (fld_bottom(sensors_state) = high)) and (fld_ponr(
           sensors_state) = high)) -> ((opening =
```

```
          uncond_closing) and (a_on = a_off)) .
C3:    ((not ((fld_bottom(sensors_state) = high) or (fld_ponr(
          sensors_state) = high))) and (fld_button(
          sensors_state) <> high)) -> ((opening = opening) and (
          a_on = a_on)) .
C4:    (not ((fld_bottom(sensors_state) = high) or ((fld_ponr(
          sensors_state) = high) or (fld_button(
          sensors_state) <> high)))) -> ((opening =
          closing) and (a_on = a_off)) .


procedure_at_closing_4.
H1:    control = closing .
H2:    not (fld_bottom(sensors_state) = high) .
H3:    not (fld_ponr(sensors_state) = high) .
H4:    not (fld_button(sensors_state) <> high) .
        ->
C1:    (fld_bottom(sensors_state) = high) -> ((control =
          halt_closed) and (a_off = a_off)) .
C2:    ((not (fld_bottom(sensors_state) = high)) and (fld_ponr(
          sensors_state) = high)) -> ((control =
          uncond_closing) and (a_off = a_off)) .
C3:    ((not ((fld_bottom(sensors_state) = high) or (fld_ponr(
          sensors_state) = high))) and (fld_button(
          sensors_state) <> high)) -> ((control = opening) and (
          a_off = a_on)) .
C4:    (not ((fld_bottom(sensors_state) = high) or ((fld_ponr(
          sensors_state) = high) or (fld_button(
          sensors_state) <> high)))) -> ((control =
          closing) and (a_off = a_off)) .
```

## 2. 7 files of SPADE proofs

init.slg, at_openi.slg, at_open.slg, at_ready.slg, at_closi.slg, at_uncon.slg, machine.slg

Proof of At_Closing_1 in file at_closi.slg shown here.

```
              ****************************************************
              LOG OF SIMPLIFICATIONS PERFORMED BY SPADE SIMPLIFIER
                        PVL SPADE TOOL VERSION : 1.4
            Copyright (C) 1986-97 Praxis Critical Systems, Bath, UK
              ****************************************************

                     DATE : 12-SEP-1999 TIME : 12:20:07



@@@@@@@@@@  VC: procedure_at_closing_1.  @@@@@@@@@@
%%%  Simplified C1 on reading formula in, to give:
     %%%  C1:  true
%%%  Simplified C2 on reading formula in, to give:
     %%%  C2:  not (not fld_bottom(sensors_state) = high and fld_ponr(sensors_state) = high)
%%%  Simplified C3 on reading formula in, to give:
     %%%  C3:  not (not (fld_bottom(sensors_state) = high or fld_ponr(sensors_state) = high)
and fld_button(sensors_state) <> high)
%%%  Simplified C4 on reading formula in, to give:
     %%%  C4:  fld_bottom(sensors_state) = high or (fld_ponr(sensors_state) = high or
fld_button(sensors_state) <> high)
-S-  Applied substitution rule at_closing_rules(3).
     This was achieved by replacing all occurrences of high by:
          sensors__high.
<S>  New H2:  fld_bottom(sensors_state) = sensors__high
<S>  New C2:  not (not fld_bottom(sensors_state) = sensors__high and fld_ponr(sensors_state) =
sensors__high)
<S>  New C3:  not (not (fld_bottom(sensors_state) = sensors__high or fld_ponr(sensors_state) =
sensors__high) and fld_button(sensors_state) <> sensors__high)
<S>  New C4:  fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)
```

```
***  Proved C1:  true
***  Proved C2:  not (not fld_bottom(sensors_state) = sensors__high and
fld_ponr(sensors_state) = sensors__high)
     using hypothesis H2.
***  Proved C3:  not (not (fld_bottom(sensors_state) = sensors__high or
fld_ponr(sensors_state) = sensors__high) and fld_button(sensors_state) <> sensors__high)
     using hypothesis H2.
***  Proved C4:  fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)
     using hypothesis H2.
***  PROVED VC.


@@@@@@@@@@  VC: procedure_at_closing_2.  @@@@@@@@@@
%%%  Simplified C1 on reading formula in, to give:
     %%%  C1:  not fld_bottom(sensors_state) = high
%%%  Simplified C2 on reading formula in, to give:
     %%%  C2:  true
%%%  Simplified C3 on reading formula in, to give:
     %%%  C3:  not (not (fld_bottom(sensors_state) = high or fld_ponr(sensors_state) = high)
and fld_button(sensors_state) <> high)
%%%  Simplified C4 on reading formula in, to give:
     %%%  C4:  fld_bottom(sensors_state) = high or (fld_ponr(sensors_state) = high or
fld_button(sensors_state) <> high)
>>>  Restructured hypothesis H2 into:
     >>>  H2:  fld_bottom(sensors_state) <> high
-S-  Applied substitution rule at_closing_rules(3).
     This was achieved by replacing all occurrences of high by:
          sensors__high.
<S>  New H2:  fld_bottom(sensors_state) <> sensors__high
<S>  New H3:  fld_ponr(sensors_state) = sensors__high
<S>  New C1:  not fld_bottom(sensors_state) = sensors__high
<S>  New C3:  not (not (fld_bottom(sensors_state) = sensors__high or fld_ponr(sensors_state) =
sensors__high) and fld_button(sensors_state) <> sensors__high)
<S>  New C4:  fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)
***  Proved C2:  true
***  Proved C1:  not fld_bottom(sensors_state) = sensors__high
     using hypothesis H2.
***  Proved C3:  not (not (fld_bottom(sensors_state) = sensors__high or
fld_ponr(sensors_state) = sensors__high) and fld_button(sensors_state) <> sensors__high)
     using hypothesis H3.
***  Proved C4:  fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)
     using hypothesis H3.
***  PROVED VC.


@@@@@@@@@@  VC: procedure_at_closing_3.  @@@@@@@@@@
%%%  Simplified C1 on reading formula in, to give:
     %%%  C1:  not fld_bottom(sensors_state) = high
%%%  Simplified C2 on reading formula in, to give:
     %%%  C2:  not (not fld_bottom(sensors_state) = high and fld_ponr(sensors_state) = high)
%%%  Simplified C3 on reading formula in, to give:
     %%%  C3:  true
%%%  Simplified C4 on reading formula in, to give:
     %%%  C4:  fld_bottom(sensors_state) = high or (fld_ponr(sensors_state) = high or
fld_button(sensors_state) <> high)
>>>  Restructured hypothesis H2 into:
     >>>  H2:  fld_bottom(sensors_state) <> high
>>>  Restructured hypothesis H3 into:
     >>>  H3:  fld_ponr(sensors_state) <> high
-S-  Applied substitution rule at_closing_rules(3).
     This was achieved by replacing all occurrences of high by:
          sensors__high.
<S>  New H2:  fld_bottom(sensors_state) <> sensors__high
<S>  New H3:  fld_ponr(sensors_state) <> sensors__high
<S>  New H4:  fld_button(sensors_state) <> sensors__high
<S>  New C1:  not fld_bottom(sensors_state) = sensors__high
<S>  New C2:  not (not fld_bottom(sensors_state) = sensors__high and fld_ponr(sensors_state) =
sensors__high)
<S>  New C4:  fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)
***  Proved C3:  true
***  Proved C1:  not fld_bottom(sensors_state) = sensors__high
     using hypothesis H2.
```

```
***  Proved C2:  not (not fld_bottom(sensors_state) = sensors__high and
fld_ponr(sensors_state) = sensors__high)
     using hypothesis H3.
***  Proved C4:  fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)
     using hypothesis H4.
***  PROVED VC.


@@@@@@@@@@  VC: procedure_at_closing_4.  @@@@@@@@@@
%%%  Simplified C1 on reading formula in, to give:
     %%%  C1:  fld_bottom(sensors_state) = high -> control = halt_closed
%%%  Simplified C2 on reading formula in, to give:
     %%%  C2:  not fld_bottom(sensors_state) = high and fld_ponr(sensors_state) = high ->
control = uncond_closing
%%%  Simplified C4 on reading formula in, to give:
     %%%  C4:  not (fld_bottom(sensors_state) = high or (fld_ponr(sensors_state) = high or
fld_button(sensors_state) <> high)) -> control = closing
>>>  Restructured hypothesis H2 into:
     >>>  H2:  fld_bottom(sensors_state) <> high
>>>  Restructured hypothesis H3 into:
     >>>  H3:  fld_ponr(sensors_state) <> high
>>>  Restructured hypothesis H4 into:
     >>>  H4:  fld_button(sensors_state) = high
-S-  Applied substitution rule at_closing_rules(1).
     This was achieved by replacing all occurrences of a_on by:
          actuator__a_on.
<S>  New C3:  not (fld_bottom(sensors_state) = high or fld_ponr(sensors_state) = high) and
fld_button(sensors_state) <> high -> control = opening and a_off = actuator__a_on
-S-  Applied substitution rule at_closing_rules(2).
     This was achieved by replacing all occurrences of a_off by:
          actuator__a_off.
<S>  New C3:  not (not (fld_bottom(sensors_state) = high or fld_ponr(sensors_state) = high)
and fld_button(sensors_state) <> high)
-S-  Applied substitution rule at_closing_rules(3).
     This was achieved by replacing all occurrences of high by:
          sensors__high.
<S>  New H2:  fld_bottom(sensors_state) <> sensors__high
<S>  New H3:  fld_ponr(sensors_state) <> sensors__high
<S>  New H4:  fld_button(sensors_state) = sensors__high
<S>  New C1:  fld_bottom(sensors_state) = sensors__high -> control = halt_closed
<S>  New C2:  not fld_bottom(sensors_state) = sensors__high and fld_ponr(sensors_state) =
sensors__high -> control = uncond_closing
<S>  New C4:  not (fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)) -> control = closing
<S>  New C3:  not (not (fld_bottom(sensors_state) = sensors__high or fld_ponr(sensors_state) =
sensors__high) and fld_button(sensors_state) <> sensors__high)
***  Proved C1:  fld_bottom(sensors_state) = sensors__high -> control = halt_closed
     using hypothesis H2.
***  Proved C2:  not fld_bottom(sensors_state) = sensors__high and fld_ponr(sensors_state) =
sensors__high -> control = uncond_closing
     using hypothesis H3.
***  Proved C4:  not (fld_bottom(sensors_state) = sensors__high or (fld_ponr(sensors_state) =
sensors__high or fld_button(sensors_state) <> sensors__high)) -> control = closing
     using hypothesis H1.
***  Proved C3:  not (not (fld_bottom(sensors_state) = sensors__high or
fld_ponr(sensors_state) = sensors__high) and fld_button(sensors_state) <> sensors__high)
     using hypothesis H4.
***  PROVED VC.
```

3. 7 files of summary theorems

`init.siv, at_openi.siv, at_open.siv, at_ready.siv, at_closi.siv, at_uncon.siv, machine.siv`

File `at_closi.siv` shown here

```
          *******************************************************
                      Semantic Analysis of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 2.5 / 04.97
                           Demonstration Version
          *******************************************************


      CREATED 12-SEP-1999, 12:19:09  SIMPLIFIED 12-SEP-1999, 12:20:07
               (Simplified by SPADE Simplifier, Version 1.4)
```

```
procedure Transitions.At_Closing


For path(s) from start to finish:

procedure_at_closing_1.
*** true .          /* all conclusions proved */


procedure_at_closing_2.
*** true .          /* all conclusions proved */


procedure_at_closing_3.
*** true .          /* all conclusions proved */


procedure_at_closing_4.
*** true .          /* all conclusions proved */
```