AUTOMATED VALIDATION FOR SYNCHRONOUS REACTIVE EMBEDDED SYSTEMS

Honors Research Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Science with Honors Research Distinction
at The Ohio State University

By

Vahid Rajabian Schwart

* * * * *

Department of Electrical and Computer Engineering
The Ohio State University

2012

Thesis Committee:

Paolo Sivilotti, Co-Adviser

Ashok Krishnamurthy, Co-Adviser

Approved by

Co-Adviser

Co-Adviser

© Copyright by

Vahid Rajabian Schwart

2012

ABSTRACT

Proper functionality is a necessity for systems used in safety-critical applications; consequently, software in these systems is often subject to rigorous validation and formal verification that aims at ensuring expected behavior. To aid in the design of these systems, several synchronous programming languages exist for describing deterministic system models suitable for formal verification and validation. Examples of such synchronous languages include SIGNAL, Lustre, MRICDF, and Esterel. Common application domains for synchronous programs include avionics, automotive control, process control, and defense systems. In many cases, rigorous formal verification of these systems is unfeasible because the methods, such as theorem proving and model checking, are too expensive. A theorem proving approach requires a great deal of user involvement and expertise, and a model checking approach may not be feasible on systems of substantial complexity due to computation constraints. This thesis presents the design, implementation, and evaluation of SAGA, a prototype tool for the automated validation of synchronous reactive embedded systems. SAGA shifts the testing effort associated with critical systems from creating individual test cases manually to reasoning about the safety and environment properties of a system. The approach SAGA takes is to generate relevant inputs to the system-under-test from a user-specified environment description, and to validate the resulting system behavior against user-specified safety properties. This overview of SAGA includes a thorough user's guide and important implementation details. Additionally, the validation process with SAGA is qualitatively assessed. The assessment is done through a case study involving the celebrated steam boiler control specification problem. Results from this case study reveal the utility of SAGA in exposing non-trivial system errors.

This thesis is dedicated to my brother, Nabil.

ACKNOWLEDGMENTS

I would like to start by thanking my advisors, Paul Sivilotti and Ashok Krishnamurthy; Ashok for readily providing the means to making this thesis possible, and Paul for his diligent effort in working with me. Paul's constant feedback throughout this demanding process and dedicated commitment to my thesis have been invaluable. From day one, he has trusted in my ability to meet the challenges at hand and has provided me with the needed guidance to do so.

The work presented in this thesis has come to fruition from my internship experience at the Air Force Research Laboratory (AFRL). I would therefore like to thank William McKeever and Steven Drager, who served as my mentors at AFRL and provided me with the initial direction of my project. Without their support and guidance, this thesis would not have been possible. I would also like to thank my co-workers at AFRL, Matthew Anderson and Thomas Levy, for their assistance in this project and for the many technical discussions which have played a key part in my work.

Finally, I would like to thank the participants in my research group seminar at The Ohio State University (OSU): Ted Pavlic, Diego Zaccai, Derek Bronish, Yan Xu, Sanket Tavarageri, Aditi Tagore, and Jason Kirschenbaum. They have provided me with valuable commentary on my thesis and have made my undergraduate research experience at OSU an engaging and gratifying endeavor.

TABLE OF CONTENTS

Chapters:

Appendices:

LIST OF TABLES

Table **Page**

LIST OF FIGURES

LISTINGS

CHAPTER 1

INTRODUCTION

1.1 Problem Statement and Motivation

Proper functionality is a must for systems used in safety-critical applications; consequently, software in these systems is subject to rigorous validation and formal verification that aims at ensuring expected behavior [21, 1, 16]. The rise in autonomy in modern-day embedded systems has been accompanied by a rise in software complexity, which has made the verification and validation of these systems an increasing challenge [24, 19]. The failure of such systems, particularly in safety-critical applications, can result in serious injury, loss of life, environmental damage, or negative economic repercussions [15, 9]. According to the National Institute of Standards and Technology, in 2002 the cost to the US economy due to software errors was estimated to be $59.5 billion [27].

The problems associated with current-day verification and validation (V&V) methods also concern national security. In Technical Horizons, a report establishing the vision for U.S. Air Force key science and technology focus areas for 2010-2030, V&V is highlighted as a main research focus area [24]. The relative ease with which modern autonomous systems can be developed poses an advantage to adversaries who may choose to field such systems without the burden of ensuring them trustworthy [24].

In many cases, rigorous formal verification is unfeasible because the methods, such as theorem proving and model checking, are costly [23, 10]. The type of costs associated with each method varies based on the type of formal verification being performed. A theorem proving approach requires a great deal of user involvement and expertise [10]. A model checking approach may not be feasible on systems of substantial complexity due to computation constraints [10]. Figure 1.1 illustrates the state space explosion problem associated with model checking. The approach taken in model checking is to conduct an exhaustive exploration of the program state space; therefore when given an automaton A with M number of states and n Boolean inputs, the state space S of A that must be explored grows exponentially as the number of inputs n increases as follows: $S(A) = M * 2^n$.

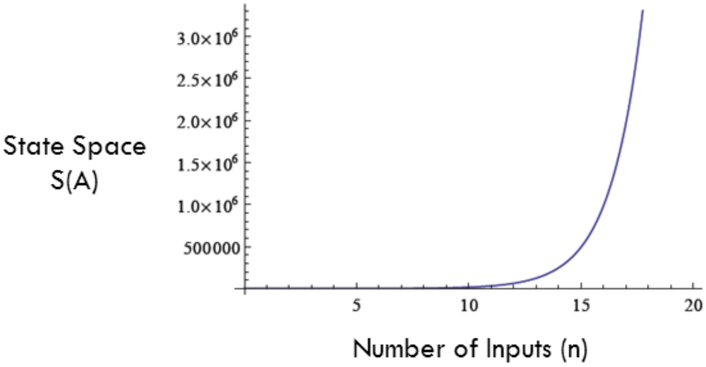


Figure 1.1: Exponential Growth of Automaton State Space

1.2 Contribution

The primary contribution of this thesis is SAGA, the SIGNAL Auto-Generated Assayer. SAGA is a prototype tool designed and implemented for the automated validation of synchronous reactive embedded systems. It provides a framework which shifts the testing effort associated with critical systems from manually creating individual test cases to reasoning about the safety and environment properties of a system. The purpose behind the development of SAGA is to provide a means towards increasing the effectiveness and lessening the effort associated with the validation process of modern-day embedded systems.

The operation of SAGA is fully defined in this thesis. A detailed overview describes the validation process in SAGA, from system environment simulation to checking system safety properties. A thorough guide to using SAGA is also included. In the guide, a cruise control system is used to illustrate a complete working example.

Finally, a qualitative assessment of SAGA is done through a case study. The case study consists of the application of SAGA to the celebrated steam boiler controller specification problem. For the purpose of this case study, an implementation of the steam boiler controller was developed using the SIGNAL language. Using safety properties derived directly from the specification, the system is validated in two scenarios: (i) a realistic runtime environment, where the typical operation of the system can be inspected and (ii) a stress scenario in which threshold values are exceeded and boundary operating cases are observed.

1.3 Related Work

The motivating purpose behind the development of synchronous languages has been to provide a suitable means for correctly implementing safety-critical systems, hence their formal verification and validation has been a research subject of great attention. Of these languages, Lustre has received the most interest from industry and academia due to its successful commercialization [4]. Many of the automatic testing tools developed for reactive systems are therefore for Lustre programs. Some of these tools include Lutess, Lurette, and GATeL [4].

An overarching theme in automated validation is the use of synchronous observers as established in [18]. This approach lends itself adequate for synchronous programs since communication among synchronous processes is an ordered procedure that avoids the problem of non-determinism in the interleaving of asynchronous processes. Many of the tools for Lustre take a monoformalistic approach, where the test generator is specified using the same language as the system programming language. One such approach is done in [17]. Test generation in [17] is done by specifying a set of Lustre invariants which impose constraints on the possible input vectors that a random generator may produce, and analyzing the runtime behavior with safety properties (also specified in Lustre). This approach has the obvious advantage that a user won't need to learn an additional specification language, but may in turn limit the overall capabilities of a tool.

Both Lutess and Lurette use synchronous observers and a monoformalistic test specification with language extensions. The testing methods they adopt are random, property-guided, operational profiled-based, and pattern-based testing [4, 25]. The tool GATeL takes a different approach. Given a specified property, GATeL provides a sequence of input vectors which will arrive the program from an initial configuration to a state which will test the property. GATeL therefore requires the full specification of the system-under-test in order to find such a sequence of input vectors [4]. Currently, Sigali is the only verification tool available for SIGNAL programs, and is included in the Polychrony Toolset. Sigali, however, does not include automatic testing capabilities [7]. No other tools for SIGNAL currently exist in the domain of this work.

1.4 Thesis Organization

The work presented in this Thesis is organized as follows:

- Chapter 2 gives an introduction to the synchronous languages and the synchronous approach to validation. A brief overview of the SIGNAL language is given and a cruise control system is presented as an example SIGNAL program.
- Chapter 3 provides an overview of SAGA and defines its approach to environment simulation and safety validation.
- Chapter 4 serves as a guide for using SAGA by providing a definition of its syntax and required file formats. A complete example of a validation session on the cruise control system is included.
- Chapter 5 presents a case study on the celebrated steam boiler controller specification problem as a qualitative assessment of SAGA.
- Chapter 6 describes important design decisions and implementation details of SAGA.

CHAPTER 2

BACKGROUND AND THEORY

2.1 Synchronous Programming Languages

The synchronous programming languages provide a favorable approach to describing reactive embedded systems [13, 11, 5]. A reactive system is one that interacts with its environment by producing a response to every event. In particular, the system must properly respond to the environment before a second event acts upon it; therefore, reactive systems are typically subject to strict timing constraints, and require concurrent determinism [22, 13]. The synchronous languages provide means for ensuring concurrent determinism and abstracting timing requirements from the software specification of a system [13, 5]. The synchronous deterministic system models are suitable for formal verification and validation, and are therefore commonly used in specifying safety-critical applications [5, 8]. Some synchronous programming languages include Esterel, SIGNAL, Lustre, Argos, and MRICDF [13, 8]. Common application domains for synchronous programs include signal processing systems, avionics, automotive control, nuclear power control, and defense systems [28, 26].

2.1.1 Synchronous Hypothesis

Synchronous languages vary in programming style, but are all based on the same computation model. This computation model is founded on the synchronous hypothesis. The synchronous hypothesis states that computation and communications are instantaneous from the point of view of logical time [5, 12]. Figure 2.1 illustrates the concept of synchronous computation.

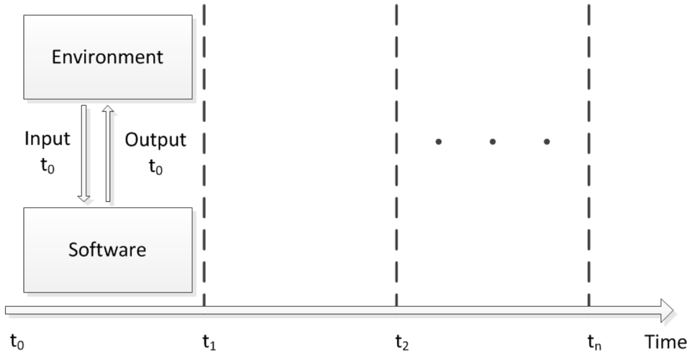


Figure 2.1: Synchronous Software Operation

The synchronous model is fundamentally a time abstraction that assumes the hardware timing constraints are met by the system [11]. A *step* in execution denotes a logical instant in time. Each *step* in a synchronous system is an ordered sequence consisting of the reception of inputs, the internal program computation, and the generation of output values. The execution of a synchronous program consists of a sequence of steps.

2.1.2 Synchronous Validation

The synchronous paradigm is suitable for automated validation due its inherent determinism. An overarching theme in automated validation of synchronous programs is the use of synchronous observers as established in [18]. By observer, it is meant a second program that monitors the runtime behavior of the program under test [14]. This approach is adequate for synchronous programs since communication among synchronous processes is a strict sequence of atomic steps that avoids the problem of non-determinism in the interleaving of asynchronous processes [14].

More specifically with reactive systems, the synchronous observers are an environment property observer and a safety property observer that work in conjunction with a test generator. The purpose of an environment observer is to ensure that only relevant test cases are produced by the test case generator. In practice, this is typically implemented by specifying environment test profiles. Given a sequence of test cases, the purpose of a safety observer is to ensure the system behavior meets specified safety properties; therefore, rather than explicitly providing a test suite, a collection of invariant safety properties must be specified.

2.2 SIGNAL

SIGNAL is a *polychronous* (i.e. synchronous, multi-clock) dataflow specification language. Polychronous components allow multiple clock rates, and are therefore suitable for describing distributed systems [13, 12]. In a dataflow description of a program, each concurrent statement can effectively have a different clock by having dependencies on different signals. This concept is often found in hardware description languages, where a statement is updated only when an event, such as a rising-edge,

Construct Type	**Syntax**
Parallel Composition	(| P | Q |)
Restriction	P `where x`

Assignment	`y := x`
	`y := F(`x_1`,`x_2`,...,`x_n`)`

Delay Assignment	`y := x $ init c`
Sampling Assignment	`y := x when z`
Merging Assignment	`y := x default z`

Table 2.1: Assignment Statements in SIGNAL

occurs on one of its signals. At each step in the execution of a SIGNAL program, each signal can be either *present* or *absent* [11].

SIGNAL was developed in Rennes, France at the Research Institute in Computer Science and Random Systems (IRISA) [11]. A SIGNAL compiler is included in the Polychrony toolset which is freely distributed by the ESPRESSO team at IRISA [28]. A commercial implementation of Polychrony, called RT-Builder, is supplied by the company GeenSoft for industrial scale projects [20, 11].

2.2.1 Primitive Language Constructs

The SIGNAL primitive language constructs are briefly introduced here, and an example of the SIGNAL specification for a cruise control system is provided in the following section. For a complete guide to the SIGNAL language, the reader is referred to [6]. The primitive language constructs of SIGNAL are summarized in Table 2.1. The individual statements that make up a program in SIGNAL are added through parallel composition. The parallel composition of P and Q means that both statements execute concurrently. In a SIGNAL program, a subprocess declaration is done by using a restriction statement. The restriction statement declares `x` as being contained in P.

A concurrent assignment statement will assign the current step value of the signal `x` to the signal `y`. A delay assignment is much like a concurrent assignment, but with the inclusion of a delay operation. The delay operator " `$` " is used to assign the previous-step value of `x` to `y`. The sampling assignment can be used to assign a value to `y` when a given signal `x` is *present* and *true* or an expression is logically true. A merge assignment can be used to assign the value of `x` to `y` when `x` is present, or the value of `z` to `y` when `x` is absent and `z` is present.

2.2.2 Example: A Cruise Control System

A cruise control system is incrementally introduced in this section to provide an example of a SIGNAL program. The full program listing is provided in Appendix A. The cruise control system is further used in chapter 4 as the system-under-test for an example of a validation session in SAGA. For sake of clarity, this example provides a simple abstraction of an actual cruise control. The system is simplified by using the output signals `throttle` and `brake` as Boolean assertions to control the vehicle speed differential. Furthermore, the system is restricted to a single-clocked specification, where all signals share the same clock. A diagram of the cruise control system is presented in Figure 2.2.

The diagram illustrates the system inputs and outputs which constitute the cruise control interface. Listing 2.1 shows the corresponding interface specification in SIGNAL. These statements are the preamble to the body description of the system. The

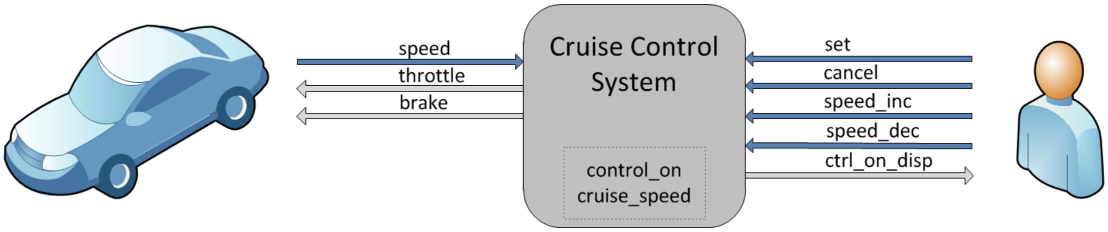


Figure 2.2: Cruise Control System Diagram

`set` input is used to activate cruise control, and the `cancel` input is used to deactivate cruise control. The `speed_inc` and `speed_dec` inputs are used to increment and decrement, respectively, the cruise control speed. The input `speed` is an integer value provided to the cruise control system which represents the current reading from a speed sensor in the vehicle. The outputs `throttle` and `brake` are used by the cruise control system to increase or decrease, respectively, the speed of the vehicle. Finally, the `ctrl_on_disp` output is used to indicate whether cruise control is active or inactive.

```
process cruiseControl =
(       % inputs %
      ? integer speed;         % speed sensor %
        boolean set,           % turn ON cruise control %
                cancel,        % turn OFF cruise control %
                speed_inc,     % increase cruise speed %
                speed_dec;     % decrease cruise speed %
        % outputs %
      ! boolean throttle,      % throttle control %
                brake,         % brake control %
          ctrl_on_disp;        % control indicator %
)
```

Listing 2.1: Cruise Control: Input and Output Interface

The internal variables `control_on` and `cruise_speed` are declared in Listing 2.2. The `control_on` variable is used to store the current status of the system. A value of *true* for `control_on` indicates the cruise control system is active. The `cruise_speed` variable is used to store the speed at which the cruise control system must maintain the vehicle.

```
where % local variables %
    boolean control_on;      % cruise control state %
    integer cruise_speed;    % cruise control speed %
end % cruiseControl %
```

Listing 2.2: Cruise Control: Internal Variables

The body of the program is presented in Listing 2.3. It is important to keep in mind that assignment statements are executed concurrently within a step. The statement in line 1 assigns a value to the internal variable `control_on` of *true* when `set` is *true*, else *false* when `cancel` is *true*; otherwise, it maintains its previous-step value `control_on$`. The output signal `ctrl_on_disp` in line 5 reflects the value of the internal variable `control_on`.

```
1  ( | control_on := ( true when set )
2      default ( false when cancel )
3      default ( control_on$ init false )
4
5    | ctrl_on_disp := control_on
6
7    | cruise_speed := ( speed when set ) default ( (cruise_speed$ init
         0)+1 when speed_inc )
8      default ( (cruise_speed$ init 0)-1 when speed_dec )
9      default ( cruise_speed$ init 0 )
10
11   | throttle := ( false when ( (speed >= cruise_speed) and
        control_on) )
12     default ( true when ((speed < cruise_speed) and control_on) )
13     default false
14
15   | brake := ( false when ((speed <= cruise_speed) and control_on) )
16     default ( true when ( (speed > cruise_speed) and control_on) )
17     default false
18   | )
```

Listing 2.3: Cruise Control: Signal Assignments

Line 7 provides that when `set` is asserted, the current value of `speed` is saved in the internal variable `cruise_speed`; else if `speed_inc` or `speed_dec` are asserted, the value of `cruise_speed` is incremented or decremented, respectively, by one unit. The default value of `cruise_speed` is otherwise its previous-step value `cruise_speed$`.

For cruise control to effect a speed increase, the output signal `throttle` is used. Line 11 states `throttle` will only be *true* when `control_on` is active and the value of input signal `speed`, the current vehicle speed, is below `cruise_speed`. The output signal `brake` is also only *true* when `control_on` is active, but instead requires the `speed` to be greater than `cruise_speed`.

CHAPTER 3

AN OVERVIEW OF SAGA

3.1 SAGA Defined

The SIGNAL Auto-Generated Assayer (SAGA) is an automated validation tool for synchronous reactive embedded systems. It is built to work with SIGNAL programs as part of an investigation of the automatic testing of synchronous reactive systems. The main purpose of SAGA is to shift the testing effort associated with critical systems from the onerous procedure of manually creating individual test cases to reasoning about the safety and environment properties of a system.

SAGA provides a framework with which, given the current-day computational resources and a properly specified environment, a system may be simulated through multiple lifetimes of operation under varying scenarios; hence, rather than using an exhaustive exploration, SAGA provides the means for conducting a smart search in the practically infinite state space of a modern non-trivial application.

3.2 Approach

When a SIGNAL program is compiled, source code is generated in either C, C++ or Java code by the Polychrony Environment [11]. The generated code can be used for simulation or deployment purposes. SAGA works with the simulation executable program compiled from C code. The diagram in Figure 3.1 illustrates the general architecture of SAGA. Given the executable program of the system-under-test, only information about the system interface concerning the inputs and outputs must be furnished to SAGA. This allows for black-box testing where no knowledge is required of the system code, which may be proprietary or confidential. SAGA reads the interface information from a user-provided initialization file.

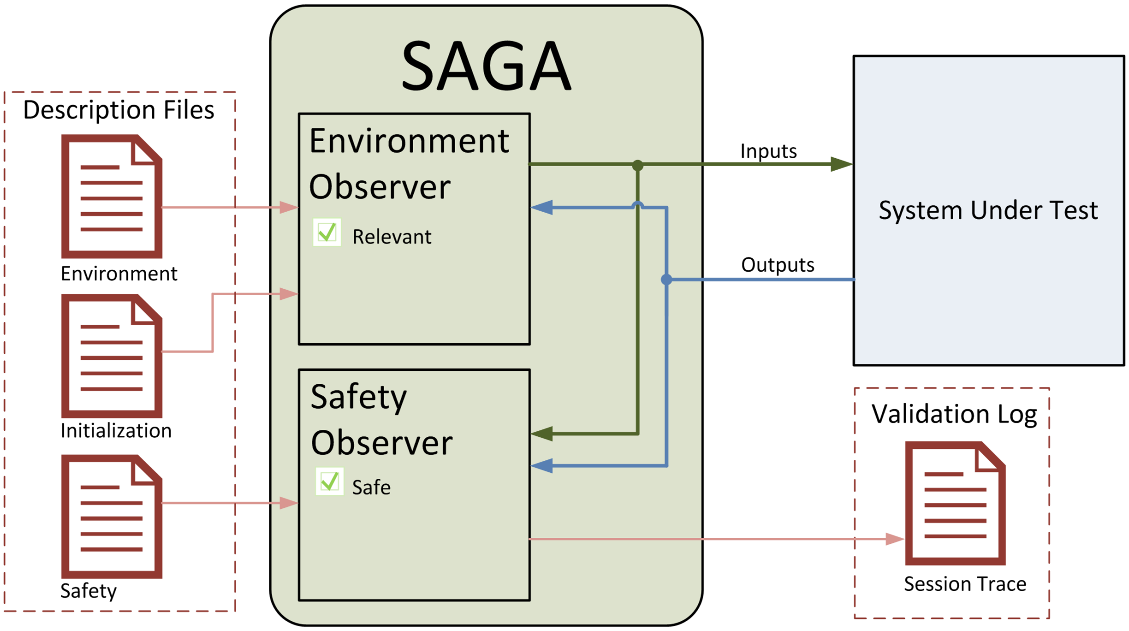


Figure 3.1: Architecture Diagram of SAGA

In addition to the initialization file, SAGA makes use of an environment description file and a safety description file containing the environment and safety properties, respectively, specified by a user for a system. The properties in these files are written with SAGA-specific syntax. Complete with these files, SAGA can run a validation session of specified length. The length in a session is defined by the number of steps (discrete logical instants) through which the system must be elapsed. During each step, SAGA generates a set of inputs to the system-under-test in accordance with the environment properties and checks the corresponding system behavior against the safety properties.

Upon the completion of each session, a log is generated of the entire simulation. This log contains the values of the provided inputs and observed outputs at each step. If a safety violation was detected during the test, a warning is displayed on the standard output and the corresponding step at which the problem occurred is annotated in the log. The entire data set is formatted in a comma-separated values (CSV) file for ease of data manipulation in parsing or generation of visual graphs.

3.3 Environment Simulation

The SAGA simulated environment is generated in accordance to the user-provided environment description. The environment is reactive with the system-under-test and therefore also takes into account the past inputs and outputs. For this reason, the user may specify initial conditions to the system (initial I/O values). For data-direction clarity, the convention is used of referring to signals from the environment to the system-under-test as inputs, and signals from the system-under-test to the environment as outputs. SAGA currently supports integer and Boolean data-types as input and output for environment simulation. For integer input generation, the user may specify a value range by setting max and min values. The default value range is the compiler-specific signed integer range. A linear constraint may also be set on the integer inputs, where a value may only increment or decrement by a single unit during each step.

It is important to note that since outputs are the response of the system-under-test to the simulated environment, only the generation of inputs to the system-under-test is controlled by SAGA. The overall behavior of the environment, i.e generated inputs, is obtained from the collection of individually specified environment properties. To specify environment properties, the following four mechanisms which are incorporated into SAGA may be used: explicit constraint, probability-based constraint, operational-profile, and pattern profile. These mechanisms make use of a variety of logical and relational operators, such as greater-than, less-than-or-equal-to, not-equal-to, etc. The mechanisms available, thoroughly detailed in Chapter 4, are introduced here:

- *Explicit Constraint*: a constraint between either a) an input and an output signal, b) two input signals, or c) an input signal and a value. The type of constraint is defined by the logical or relational operator used. The explicit constraint is satisfied by SAGA at every step in the simulation, meaning appropriate values are assigned to the input signal(s) so as to make the constraint true. For example, when an explicit constraint relating an input to an output is declared, such as $\mathbf{input}_1 < \mathbf{output}_1$, SAGA satisfies this condition by assigning a value to $\mathbf{input}_1$ that is less than the value of $\mathbf{output}_1$.

- *Probability-based Constraint*: a constraint defined similarly to an *explicit constraint*, but rather than being satisfied at every step, is only satisfied on a specified probability bias. The specified probability bias P, a real number contained in $[0, 1]$, corresponds to the probability of an associated constraint C being satisfied during the current step in simulation. It holds therefore, that the logical negation of the associated constraint, $\neg C$ is satisfied on a $1-P$ basis.

For example, a probability-based constraint such as $\mathbf{P}\ (\ \mathbf{input}_1 < \mathbf{output}_1\ )$ $= \mathbf{0.75}$ states that during any given step, there is a 75% probability of input1 being less than output1 and hence a 25% probability of $\mathbf{input}_1$ being greater than or equal to $\mathbf{output}_1$.

- *Operational Profile*: a reaction to a specified system event. The system event is denoted as an enter-condition, which is a propositional statement about the values of the previous-step signals. Given the enter-condition to an operational profile is true, then an associated constraint is satisfied on a specified probability bias. An operational profile is a probability-based constraint which becomes active if its enter-condition is true. The associated constraint condition is satisfied in the current step. For example, $\mathbf{OP\ input}_1 = \mathbf{true} - > \mathbf{P(\ input}_1$ $= \mathbf{true}\ ) = \mathbf{0.85\ END\_OP}$ means that if the value of $\mathbf{input}_1$ was true in the previous step (i.e. the enter-condition is true), then current step value of $\mathbf{input}_1$ has an 85% probability of being true. If the enter-condition is false, no assertion is made about the current value of $\mathbf{input}_1$. It is of special importance to note, as this example demonstrated, that an enter-condition refers to the previous step I/O values, and the constraint which must be satisfied pertains to the current step values.

- *Pattern Profile*: a sequence of reactions to a specified system event. Given the enter-condition of a pattern profile property evaluates to *true*, the first constraint in a sequence of probability constraints is satisfied on a specified probability bias. On the following execution step, the next probability constraint is likewise satisfied; therefore, each constraint in a pattern has a its own specified probability bias. The pattern exits once a constraint is not satisfied (i.e. when a probability evaluates to false). After exiting, the enter-condition must once again be true for the pattern to be re-entered. An example pattern profile is as follows,

PATTERN $\mathbf{output}_3 = \mathbf{100} -> \mathbf{P}(\ \mathbf{input}_2 = \mathbf{2}\ ) = \mathbf{0.95} -> \mathbf{P}(\ \mathbf{input}_2 = \mathbf{4}\ ) = \mathbf{1} -> \mathbf{P}(\ \mathbf{input}_2 = \mathbf{8}\ ) = \mathbf{0.9}$ **END_PATTERN**. Given $\mathbf{output}_3$ had a value of 100 in the previous step, there is a 95% chance $\mathbf{input}_2$ will be given a value of 2 in the current step. Subsequently, $\mathbf{input}_2$ is guaranteed (100% probability) to be assigned the value 4 in the following step and has a 90% chance of having a value of 8 two steps ahead. The probabilities assigned in a pattern profile are viewed as independent events relating to the current step of execution, consequently the actual probability of satisfying the last constraint in a sequence is in reality a compound probability (trivially, the product of all preceding probabilities).

Altogether, the validation methods in conjunction with relational operators allow for a wide-range of specifications which may be tailored to different types of systems and testing goals. A proper environment description in which a combination of common patterns of operation that occasionally deviate from typical behavior allows for an extensive exploration of the relevant state space of a system. Furthermore, the environment properties may be written so as to conduct specialized tests to search around system thresholds and boundary conditions in order to increase robustness.

3.4 Safety Validation

During each step in a simulation, the safety validation phase occurs after the system-under-test has reacted to the SAGA generated inputs. The validation process in SAGA consists of checking the execution of the system against the user specified safety properties. Similar to the environment description, the safety description is a collection of individual properties that together encompass the overall safe behavior of a system. The safety properties must therefore be written so as to express the conservative set of states in which nothing bad happens.

In SAGA, safety properties are written as invariant propositional statements on the inputs and outputs of a system with the use of logical and mathematical relational operators. Each property must be true throughout the entire execution for a system to be safe. The safe behavior of a system is therefore contained in the intersection of the set of states described by each safety property, as is illustrated in Figure 3.2.

Although, the use of temporal logic is not fully incorporated into SAGA validation, a delay operator \$ allows invariant properties to be set between current and previous-step signals. The delay operator \$, inherited from the Signal language, allows one to refer to the previous-step value of a signal. It can be used by appending it to an input or output signal as such: $\textbf{input}_3\textbf{\$}$. This operator can therefore be used to include temporal aspects into a safety property.

Individual safety properties may be specified as single or multiple nested propositional statements by using logical operators and parenthesis. For example, using Boolean data-type signals $\textbf{output}_1$, and $\textbf{output}_2$ a property could be written using logical operators as follows, **SAFE** $\textbf{output}_1$ **OR** $\textbf{output}_2$ **END_SAFE**. The

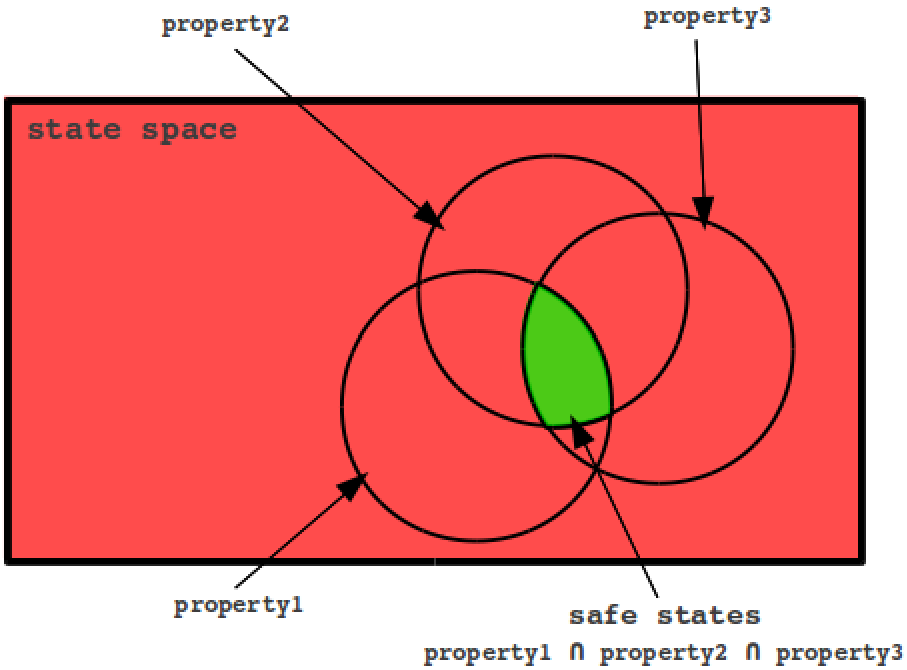


Figure 3.2: Safety Properties and the System State Space

property states that either **output**$_1$ or **output**$_2$ must always be true. A second property could be specified using a integer data-type signal **output**$_3$, a relational operator, and the delay operator, in the following manner, **SAFE output**$_3$ $>$ **output**$_3$**$** **END_SAFE**. This property checks that the current value of **output**$_3$ is always greater than its previous step value **output**$_3$**$**. With proper use of parenthesis and logical connectives, individual properties may be as extensive as necessary. By nesting statements and using basic operators, a more complex property may be written as follows, **SAFE (output**$_3$ $>$ **100) AND ((output**$_1$ $=$ **output**$_1$**$) OR (input**$_1$ $\geq$ **output**$_3$ **)) END_SAFE**. The requirements this property imposes on a system, is that **output**$_3$ must be greater than 100 and either the value of **output**$_1$ is equal to its past value or **input**$_1$ is greater-than-or-equal to **output**$_3$. Generally, a safety violation is detected by SAGA during the validation phase when any one of the properties, such as the ones presented here, in a safety description evaluate to false during any given step in a simulation.

The aim of a well-written safety description must be to conservatively restrict the system to trusted behavior. It may be challenging to produce safety properties at times because it is not always evident what kind of system behavior can cause problems; however, even when SAGA does not detect an error condition due to lack of a thorough specification, an inspection of the simulation log may in many cases reveal program bugs. Furthermore, engagement in the activity of creating safety properties by the system designer can in itself lead to the discovery of system design errors or bugs.

CHAPTER 4

USING SAGA

4.1 Compiling and Running SAGA

SAGA is a console application that accepts the required specification files and optional flags as parameters to run a simulation. In order to install SAGA, it must first be compiled from source code. To compile the latest version of SAGA, a plain "make" command can be used on the makefile provided in the base directory of the source code folder. The makefile requires the GNU Compiler Collection (GCC). Upon compilation, an executable "SAGA" is generated in the base directory. This executable may be moved to the desired installation directory. SAGA may then be executed from its install directory as a console application using a UNIX shell. Upon running SAGA without providing parameters, the following help menu is displayed which explains the use of individual flags:

```
Usage:
   SAGA [flags] ... [length] [init_file] [env_file] [safe_file]

Description:
   Run a validation session of length [length] for a specified
      system, whose I/O interface is contained in [init_file]. The
      environment is generated with respect to the properties
      specified in [env_file] and the system is validated with
      respect to the properties specified in [safe_file]. The output
       of the session is logged in a comma separated variable file
      "[system name]_trace_[TIMESTAMP].csv".

Flags:
   -v
       Verbose mode. Provides detailed information about the
          simulation. It is recommended to pipe the output of SAGA
          to a text file when using verbose mode. An example
          execution would be of the format:
           SAGA -v 100 init.txt env.txt safe.txt > log.txt
   -i
       Manual initialization. A prompt is provided to enter
          information about the System (rather than providing [
          initial_file] ).
   -help init
       Provides information about the contents and format of the
          initialization file as required by SAGA.
   -help env
       Provides information about the contents and format of the
          environment description file as required by SAGA.
   -help safe
       Provides information about the contents and format of the
          safety description file as required by SAGA.
```

Listing 4.1: SAGA Help Menu

The file parameters `[init_file]`, `[env_file]`, `[safe_file]` above refer to the directory location of the initialization, environment, and safety files, respectively. The test length parameter, `[length]`, specifies the number of *steps* (discrete logical instants) in the simulation. As an example, the configuration of SAGA for a validation session of the cruise control system introduced in Section 2.2.2 is used in this and following sections. To execute SAGA for the purpose of running a validation session on the cruise control system, the following command is used:

```
SAGA -v 50 cruiseControl_init.txt cruiseControl_env.txt cruiseControl_safe.txt
```

In the above command, the flag “`-v`” provides a verbose output to the standard output regarding the status of the simulation. The files `cruiseControl_init.txt`, `cruiseControl_env.txt`, and `cruiseControl_safe.txt` represent the corresponding initialization, environment, and safety descriptions located in the local SAGA directory and used for the cruise control system. Each file is presented in the following sections.

4.2 Specifying Initialization

In order to run a simulation, SAGA requires interface information about the system-under-test. The interface information can be provided to SAGA through an initialization file or manually through a prompt by using the `-i` flag. Since the prompt for manually entering information is self-explanatory, only the initialization file format is presented here. The initialization file requires information about the inputs and outputs of the system-under-test, and allows the user to specify system initial conditions and set options regarding environment control.

4.2.1 File Format

The initialization file must contain the complete interface information of the system-under-test regarding the name of the executable program and its input and output signals. In addition, an initial value can be specified for any I/O signal in the file to start a simulation from a predetermined state. Integer input signals must be specified with additional attributes including min and max values, as well as whether or not they should be linearly constrained. For *linearly constrained* integer signals, the environment will only generate values within single-unit increments or decrements. The information is parsed from the file in a specific order and therefore the file must be formatted properly.

The required information and ordering is as follows:

1. The name of the executable program for the system-under-test.
2. The number of Boolean input signals.
3. The number of Boolean output signals.
4. The number of integer input signals.
5. The number of integer output signals.
6. A list of the Boolean input names and initial values, for example:

   ```
   Boolean_input 0
   ```

7. A list of the Boolean output names and initial values, for example:

   ```
   Boolean_output  1
   ```

8. A list of the integer input names, their corresponding initial, min, and max values, and whether or not (1 or 0) they should be linearly constrained, for example:

   ```
   integer_input 35 0 400 1
   ```

9. A list of the integer output names and their corresponding initial values, for example:

   ```
   integer_output 5
   ```

In the initialization file, tab and newline characters are effectively treated the same as space characters and can be used as delimiters for organization. The data provided must be consistent throughout; therefore the number of signals of each data-type specified, must match an equivalent count of signal names. If there any inconsistencies in the file are detected, SAGA will produce a relevant error and quit. For any value in the file, the place holder “`x`” can be used to generate default values. By default, integer input signals have min and max values equivalent to the corresponding compiler-specific min and max signed integer values, and are not linearly constrained. For any signal, its default initial condition is randomly chosen. By random, it is meant a uniform probability of its possible values.

4.2.2 Example: Cruise Control Initialization

A sample configuration for the cruise control system with the interface information provided in Section 2.2.2 is as follows:

```
% -----------------------------
% --------cruiseControl-------
% -----------------------------

% Name of system-under-test
cruiseControl

% Number of Boolean input and output signals
% [No. input] [No. output]
4             3

% Number of Integer input and output signals
% [No. input] [No. output]
1             0

% Names of Boolean signals (must match SUT signal name) and initial values
% [name]     [init_value]
set          0                % inputs
cancel       0
speed_inc    0
speed_dec    0
throttle     0                % outputs
brake        0
```

```
ctrl_on_disp 0

% Names of Integer signals (must match SUT signal name), initial value
% minimum value, maximum value, and linear constraint
% [name]    [init_value]   [min_value] [max_value] [linear]
speed        30            0           300        1       % input
```

Listing 4.2: Contents of “cruiseControl_init.txt”

The initialization file presented above follows the formatting specified in Section 4.2.1. The Boolean values `true` and `false` are represented using 1 and 0 respectively. Furthermore, comments are included to help organize the file. The percent character % is universally used by SAGA to denote a comment; therefore, any characters following the percent character in a line are ignored.

4.3 Specifying the Environment

The environment description is a collection of properties which are used to describe the behavior of the system environment. The different types of properties which can be used are described in Section 3.3. In this section, the syntax of each type of property is detailed and examples of properties as applied to the environment description of the cruise control system are provided.

4.3.1 Syntax

The syntax is formally introduced in Backus-Naur Form (BNF). The four mechanisms for describing the environment properties are defined as follows:

Explicit Constraint

explicit_constraint ::= *in_signal operator* (*signal* | *value*) *signal* ::= *in_signal* | *out_signal* | *prev_signal*

out_signal ::= output signal

in_signal ::= input signal

prev_signal ::= *in_signal*\$ | *out_signal*\$

value ::= Boolean | integer

operator ::= logical | relational

Probability-based Constraint

probability_constraint ::= `P(` *explicit_constraint* `) =` *probability*

probability ::= $0 \leq real \leq 1$

Operational Profile

operational_profile ::= `OP` *enter_condition* `->` *probability_constraint* `END_OP`

enter_condition ::= *prev_signal operator* (*prev_signal* | *value*)

Pattern Profile

pattern ::= `PATTERN` *enter_cond* `->` *prob_const_lst* `END_PATTERN`

prob_const_lst ::= *probability_constraint* | *probability_constraint* `->` *prob_const_lst*

The available operators are provided in Table 4.1. In general, relational operators can be used on integer data-type signals, while logical operators can be used on Boolean data-type signals or with expressions which evaluate to a Boolean value. The "+" and "−" are special operators used increment or decrement an integer signal by

Comments	**Operator**	**Type**
Equality	=	Relational/Logical
Difference	!=	Relational/Logical
Disjunction	OR/||	Logical
Conjunction	AND/&&	Logical
Greater Than	>,>=	Relational
Less Than	<,<=	Relational
Increment	+	Arithmetic
Decrement	-	Arithmetic

Table 4.1: SAGA Operators

a specified value. They effectively add or subtract the value of the right expression to/from the signal on the left side.

When generating the environment during each step, SAGA ensures that every input signal is assigned a value. Given an environment description is composed of one or more properties, individual signals can be used in more than one property. Priority is given to satisfying signals in properties in the following order: pattern profile, operational profile, probability-based constraint, and explicit constraint. When a signal is used in two properties of the same type, the first property in top-down order in the description file is used to generate a value for the signal. When a signal which has already been assigned a value is used in a property that relates it to a different signal which has not been assigned a value, the property is satisfied by only assigning a new value to the second signal, or ignored if the constraint is not satisfiable. Furthermore, when a signal is not used in any one property, by default it is assigned a random value within its defined range.

4.3.2 Example: Cruise Control Environment

The following excerpt provides properties used to describe a realistic environment for the cruise control system.

```
1     % Speed up when throttle is activated
2     OP throttle = 1 -> P( speed > speed$ ) = 1 END_OP
3
4     % Driver increases speed more often when in cruise control
5     P( speed_inc = 1 ) = 0.60
6     P( speed_dec = 1 ) = 0.20
7
8     % If user decreases speed, more likely to continue decreasing
9     OP speed_dec$ = 1 -> P( speed_dec = 1 ) = 0.7
10
11    % Speed sensor fault
12    P( speed > 9000 ) = 0.01
```

Listing 4.3: Excerpt from “cruiseControl_env.txt”

In line 2 of Listing 4.3, we observe a property of the environment which states that when the Boolean output `throttle` is true, the integer input `speed` will increase in value i.e. have a greater value than `speed$`, its previous-cycle value. The environment models the theoretical driver as someone who tends to increase their speed more often when they are using the cruise control feature of the automobile, as is illustrated in lines 5 and 6, by assigning a higher probability to the `speed_inc` input signal; however, the property in line 9, accounts for the fact that if a driver decreases speed once, they will more than likely continually do so. Finally, the property in line 12 states there is a 1% probability that the value of `speed` is over 9000, which effectively simulates a sensor glitch with a discontinuous jump to a large, unrealistic value.

4.4 Specifying Safety Validation

The safety description is a collection of properties that specify the safe behavior of a system. These properties are checked at every step in a simulation. A violation of safety in the system is detected when any of the properties evaluates to false. This concept is more thoroughly described in Section 3.4.

4.4.1 Syntax

A safety property can be defined by a single expression or by multiple expressions connected by well-formed parentheses and logical operators as follows:

safety_property ::= `SAFE` *first_expression* `END_SAFE`
first_expression ::= *expression operator expression*
expression ::= *signal* | *value* | “ `(` ” *expression operator expression* “ `)` ”
signal ::= *input_signal* | *output_signal* | *prev_signal*
out_signal ::= output signal
in_signal ::= input signal
prev_signal ::= *in_signal*$ | *out_signal*$
value ::= Boolean | integer
operator ::= logical | relational

This format allows for basic propositional statements, as well as more complex statements containing several nested parenthesis. A current limitation on the syntax is that each nested expression must consist of three arguments, where the the middle argument must be a *operator* and the leftmost and rightmost arguments can be a *signal*, *value*, or a subsequent nested *expression*.

4.4.2 Example: Cruise Control Safety

The following Listing provides properties taken from the safety description used with the cruise control system.

```
1 % Throttle or brake control should not be active if cruise control is OFF
2 SAFE ( ( throttle or brake ) and ( ctrl_on_disp = 0 ) ) = 0 END_SAFE
3
4 % Brake and throttle should not be activated simultaneously
5 SAFE ( brake = 0 ) or ( throttle = 0 ) END_SAFE
6
7 % When cancel is true , cruise , throttle , and brake control should be OFF
8 SAFE ( ( cancel = 1 ) and ( ( ( ctrl_on_disp = 0) and ( throttle = 0 ) ) and ( brake = 0 ) ) ) or ( cancel = 0 ) END_SAFE
```

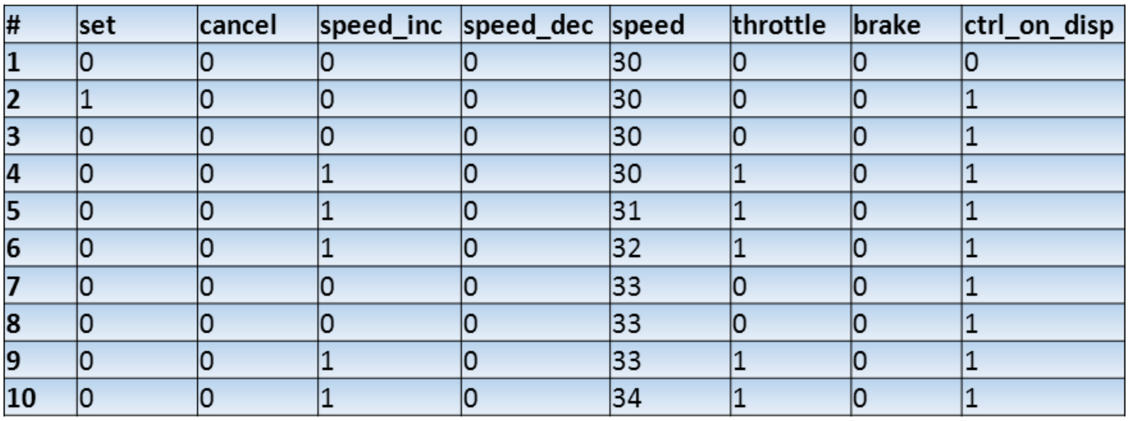
Listing 4.4: Excerpt from “cruiseControl_safe.txt”

A very basic expectation of a cruise control system is declared in line 2 of Listing 4.4, which states that neither `throttle` or `brake` should be asserted while the cruise control is OFF, as is indicated by `ctrl_on_disp`, the control state display. Furthermore, the property in line 5 stipulates that `brake` and `throttle` should not be asserted simultaneously. The last property in line 8 states that when `cancel` is asserted, meaning the driver wishes to turn OFF cruise control, the system responds accordingly.

4.5 Interpreting Results

The output of every validation session in SAGA is logged in a CSV file named `[system_name]_trace_[time_stamp].csv`. This file contains the values generated and received at every step. When a safety violation is detected, a warning is displayed on the standard output and the log is annotated on the corresponding step.

Given the initialization, environment, and safety descriptions provided for the example cruise control system, the results of the first ten steps are displayed in Figure 4.1. These values show that the set signal was asserted at logical time two and effectively turned cruise control ON. Throughout the following steps, the `speed_inc` signal is asserted, which in turn activates `throttle` and effects increases in `speed`.



#	set	cancel	speed_inc	speed_dec	speed	throttle	brake	ctrl_on_disp
1	0	0	0	0	30	0	0	0
2	1	0	0	0	30	0	0	1
3	0	0	0	0	30	0	0	1
4	0	0	1	0	30	1	0	1
5	0	0	1	0	31	1	0	1
6	0	0	1	0	32	1	0	1
7	0	0	0	0	33	0	0	1
8	0	0	0	0	33	0	0	1
9	0	0	1	0	33	1	0	1
10	0	0	1	0	34	1	0	1

Figure 4.1: Cruise Control Simulation Log

By plotting the values generated by SAGA for the `speed` signal, a visual representation of the system behavior is obtained in Figure 4.2. The figure is annotated with key events from the simulation.

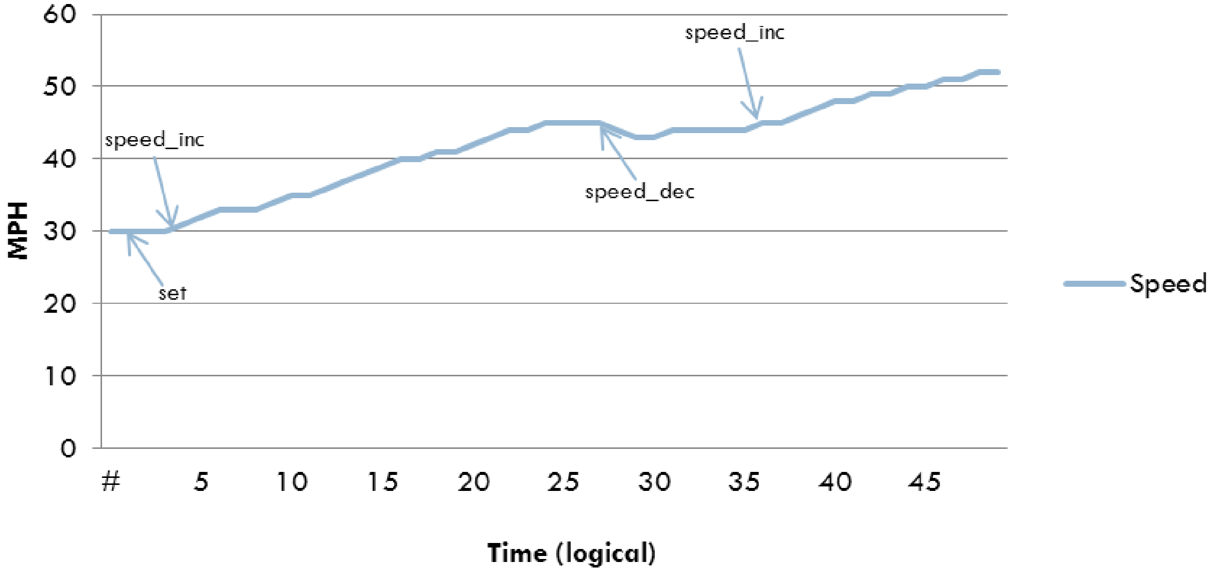
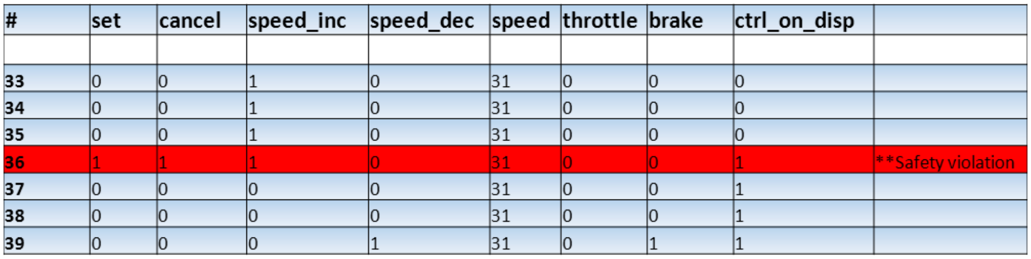


Figure 4.2: Speed Signal in Cruise Control

The visual representation of the data shows typical, expected operation of the cruise control; however, a warning on the standard display and an inspection of the simulation log yield a surprise on cycle #36 displayed in Figure 4.3. The detected error is apparent upon examination of the input and output signals. The problem, evident by the status display `ctrl_on_disp`, is that the system turned ON and remained ON, even though the `cancel` input was asserted.

Upon examination of the cruise control system Signal specification introduced in Section 2.2.2 the error points to the implementation of the local variable `control_on`



#	set	cancel	speed_inc	speed_dec	speed	throttle	brake	ctrl_on_disp	
33	0	0	1	0	31	0	0	0	
34	0	0	1	0	31	0	0	0	
35	0	0	1	0	31	0	0	0	
36	1	1	1	0	31	0	0	1	**Safety violation
37	0	0	0	0	31	0	0	1	
38	0	0	0	0	31	0	0	1	
39	0	0	0	1	31	0	1	1	

Figure 4.3: A Safety Violation

used to process the state of the system. The faulty code snippet is provided in Listing 4.5.

```
| control_on := ( true when set )
        default ( false when cancel )
        default ( control_on$ init false )
```

Listing 4.5: Cruise Control: A Bug Exposed

The test case exposed that when `set` and `cancel` where simultaneously asserted, `cancel`was ignored. Therefore, if a driver operating this system would have attempted to turn OFF cruise control and accidentally pressed both `set` and `cancel`, the system would have remained ON. This implementation is not correct with the desired safe specification of the system. Semantically, the root of the problem is the improper use of the default operator in the program. In Signal, the `default` operator gives priority to its left argument, and only if the left expression evaluates to false is the right expression evaluated. An appropriate correction in this example is to give `cancel` priority over `set` as is done in the altered code snippet in Listing 4.6.

```
| control_on := ( false when cancel )
  default ( true when set )
  default ( control_on$ init false )
```

Listing 4.6: Cruise Control: Altered Code

CHAPTER 5

CASE STUDY: STEAM BOILER

5.1 Introduction

The purpose of this chapter is to demonstrate the effectiveness of the simulation and validation capabilities of SAGA as applied to a system of substantial complexity. To that end, a well-known verification problem, the steam boiler control specification [2], was chosen. This specification has been often used for the purpose of demonstrating the application of formal methods to a real-life industrial application in order to unify scientific progress in the field with practices in industry [3]. The steam boiler problem, though most frequently used for examples of verification, was selected since its full specification is readily available and many in the target audience are familiar with it. The specification is used to develop an initial implementation of a steam boiler system with the SIGNAL programming language. Given the purpose of testing is to expose the presence of errors, not prove their absence, the goal in implementing the steam boiler is to provide a working example with which the effectiveness of the validation process in SAGA of exposing errors can be qualitatively assessed.

5.2 Steam Boiler Specification

A steam boiler program is a control application that moderates the level of water inside a steam boiler by turning water pumps ON and OFF. The control program must function properly; otherwise a quantity of water too high or too low could damage the steam boiler or the driven turbine. An informal specification for such a system constitutes the *steam boiler specification problem* presented in [2]. The case study program is implemented directly from the aforementioned text. A brief overview of the specification is provided here. For the complete specification details, the reader should refer to the original text. The SAGA files and the full program listing of the SIGNAL implementation of the steam boiler developed for this case study are provided in Appendix B. For ease of readability, the input and output signals in the implementation are consistent with the naming of the sent and received *messages* of the original specification.

5.2.1 Overview

A diagram of the steam boiler environment is depicted in Figure 5.1. This diagram demonstrates the constraints of the control application, which are namely the water level limits and the available physical devices. The implemented values associated with each water level limit are presented in Table 5.1.

The steam boiler physical environment effectively consists of four pump units, a water release valve, a water level sensor, and a steam sensor. Each pump unit consists of a pump, which can be turned ON and OFF by the program, and a pump controller that provides information to the program regarding whether or not water is flowing through the pump. The program is informed of the current amount of water in the

Parameter	**Comment**	**Value**	**Unit**
C	Maximal Capacity	150	liter
M_1	Minimal Limit	20	liter
M_2	Maximal Limit	130	liter
N_1	Minimal Normal	60	liter
N_2	Maximal Normal	90	liter

Table 5.1: Implementation Values of Physical Constants

steam boiler and the amount of steam being produced, by a water level sensor and a steam sensor, respectively. The water release valve is used by the program only during start-up in order to lower the amount of water to the desired initial level.

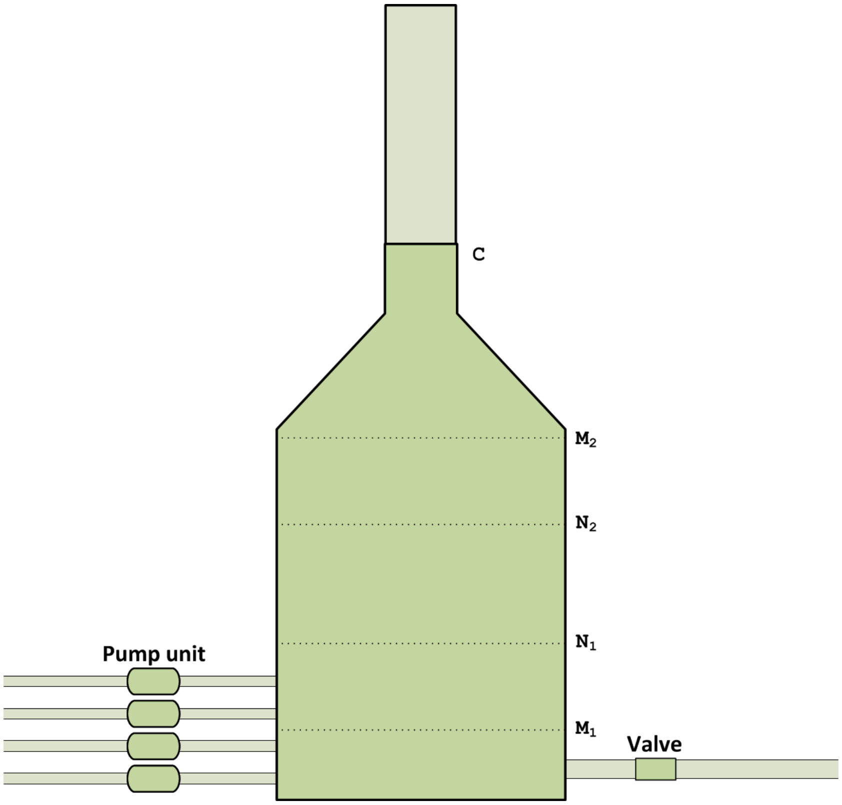


Figure 5.1: Steam Boiler Physical Diagram

The steam boiler program can be modeled as a synchronous system since it must follow a five second cycle, denoted as a *step*, in which the reception of messages, analysis of information received, and response to messages must happen, in that order. A simplified diagram of the steam boiler program interface is presented in Figure 5.2. The input and output signals constitute the *messages* between the program and the physical units.

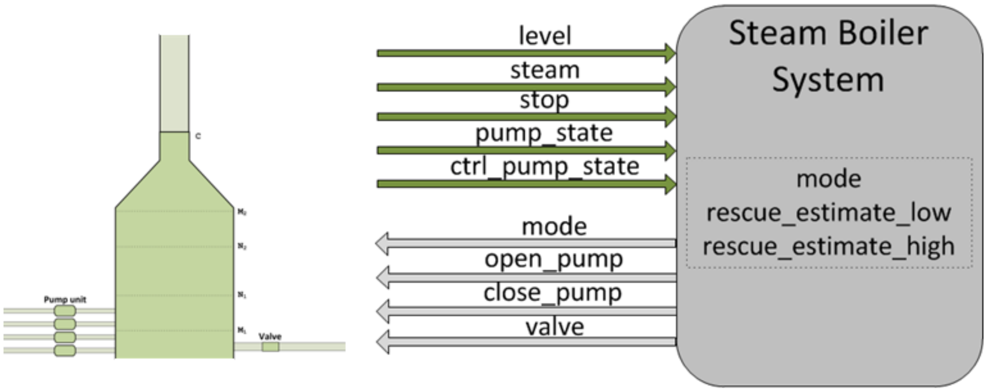


Figure 5.2: Steam Boiler Program Simplified Interface

The behavior of the steam boiler program is primarily determined by its current mode. The five possible modes of operation are the following:

- *Initialization*: The program begins in this mode from start-up. It awaits until the steam boiler is ready, and ensures the water level is initially within $N1$ and $N2$ by opening the water release valve or activating a pump if necessary before starting.
- *Normal*: While in this mode, the program tries to maintain the water level closely within $N1$ and $N2$. The program remains in this mode while no faults

are detected in the physical units and the water level is not risking reaching $M1$ or $M2$.

- *Degraded*: In this mode, the program attempts to maintain a satisfactory water level given the presence of a fault on one of the physical units other than the water level sensor.

- *Rescue*: When the water level sensor has a fault, the program enters this mode. In this mode, the program attempts to maintain a satisfactory water level by taking into account the maximum physical dynamics of the system.

- *Emergency stop*: The program enters this mode when the water level is risking reaching $M1$ or $M2$, the external `stop` signal was asserted, or a message transmission error is detected. In this mode, the program stops and the physical environment is responsible for taking appropriate actions.

5.2.2 Implementation Limitations

The steam boiler program was implemented with some assumptions and limitations. These choices where made in order to provide a simplistic model for the physical evolution of the system, and to accommodate the fact that SAGA currently only supports integer type numbers. The limitations are the following:

- Quantity measurements of liters are in whole units and are therefore represented by integers. These include the steam output and water level measurements.

- A working pump will deliver water at its constant rate of 0.2 ltrs/sec (i.e. 1 liter every 5 seconds) where 5 seconds represent one *step*. This represents the nominal capacity of each pump.

Program Mode	`mode` **Value**
Initialization	1
Normal	2
Degraded	3
Rescue	4
Emergency Stop	5

Table 5.2: Program Mode Corresponding to `mode` Values

- The system produces steam in discrete values (ltrs/step) directly related to the water level, as shown in Listing 5.4.
- The `stop` signal must only be asserted once, rather than three times in a row, for the program to go into *emergency stop* mode.

5.3 Validation

Given a complete system specification, the validation process in SAGA is a straight forward activity. The safety properties of a system can be directly implemented from the specification, and thus be done concurrently with system design. The aforementioned approach is taken in this case study. A large number of properties, each relating to a functional requirement or set of requirements, may be written to produce a thorough validation. For brevity, only a handful of safety and environment properties are presented here. A following discussion on some of the nontrivial errors detected in the implementation is then provided.

5.3.1 From Specification to Safety Description

A set of safety properties that correspond to the specification are presented in Listing 5.1. As a reference, Table 5.2 provides the values corresponding to each program mode for the signal `mode` in the implementation. In line 1, the property declares that either the program is in *emergency stop* mode or the `stop` signal is not true. This effectively states that the program should not be in a mode other than *emergency stop* if the `stop` signal is asserted. The following property in line 3, provides that the program cannot be in *normal* mode if the water level has reached or surpassed the maximal or minimal limits. When a pump and pump controller do not have matching states, it constitutes an equipment failure and the program must respond accordingly; therefore, the property in line 5 states that if `pump_state1` and `pump_ctrl_state1` do not have the same value, the program must be in *degraded*, *rescue*, or *emergency stop* mode. This property is also applied to pumps 2,3, and 4 (not shown here). Finally, the properties declared starting at line 7 check for transmission failures by ensuring that the program acknowledges every message received that requires an acknowledgment.

```
1 SAFE ( mode = 5 ) or ( stop != 1 ) END_SAFE
2
3 SAFE ( mode != 2 ) or ( ( level <= 130 ) and (level >= 20 ) )
    END_SAFE
4
5 SAFE ( pump_state1 = pump_ctrl_state1 ) or ( ( ( mode = 3 ) or (
    mode = 4 ) ) or ( mode = 5) )  END_SAFE
6
7 SAFE ( ( pump_repaired1$ == 1 ) and ( pump_repaired_ack1 == 1 ) ) or
     ( pump_repaired1$ == 0 ) END_SAFE
8 SAFE ( ( pump_ctrl_repaired1$ == 1 ) and ( pump_ctrl_repaired_ack1
    == 1 ) ) or ( pump_ctrl_repaired1$ == 0 )  END_SAFE
9 SAFE ( ( level_repaired$ == 1 ) and ( level_repaired_ack == 1 ) ) or
     ( level_repaired$ == 0 ) END_SAFE
```

```
SAFE ( ( steam_repaired$ == 1 ) and ( steam_repaired_ack == 1 ) ) or
    ( steam_repaired$ == 0 ) END_SAFE
```

Listing 5.1: Steam Boiler Safety Properties

5.3.2 Environment Simulation

Two approaches are taken for simulating a test environment in this case study: (i) describing a realistic runtime environment, where the typical operation of the system under usual conditions can be inspected and (ii) describing a stress scenario in which threshold values are exceeded and boundary operating cases are observed. The two approaches are illustrated using portions of the steam boiler environment description and their associated result.

Typical Operation

For the purpose of simulating the typical operation of the system, the environment must be described so as to be correctly responsive to the system-under-test. This is achieved by ensuring that the output of the program effects an appropriate change in the environment. The properties in Listing 5.2 demonstrate this by assigning a high probability to providing the correct response when a pump is opened or closed. The response is the pump and pump controller state change according to the program action.

```
OP close_pump1 = 1 -> P( pump_state1 == 0 ) = 0.95 END_OP
OP close_pump2 = 1 -> P( pump_state2 == 0 ) = 0.95 END_OP
OP close_pump3 = 1 -> P( pump_state3 == 0 ) = 0.95 END_OP
OP close_pump4 = 1 -> P( pump_state4 == 0 ) = 0.95 END_OP

OP open_pump1 = 1 -> P( pump_state1 == 1 ) = 0.95 END_OP
OP open_pump2 = 1 -> P( pump_state2 == 1 ) = 0.95 END_OP
OP open_pump3 = 1 -> P( pump_state3 == 1 ) = 0.95 END_OP
OP open_pump4 = 1 -> P( pump_state4 == 1 ) = 0.95 END_OP

OP close_pump1 = 1 -> P( pump_ctrl_state1 == 0 ) = 0.95 END_OP
OP close_pump2 = 1 -> P( pump_ctrl_state2 == 0 ) = 0.95 END_OP
OP close_pump3 = 1 -> P( pump_ctrl_state3 == 0 ) = 0.95 END_OP
OP close_pump4 = 1 -> P( pump_ctrl_state4 == 0 ) = 0.95 END_OP

OP open_pump1 = 1 -> P( pump_ctrl_state1 == 1 ) = 0.95 END_OP
OP open_pump2 = 1 -> P( pump_ctrl_state2 == 1 ) = 0.95 END_OP
OP open_pump3 = 1 -> P( pump_ctrl_state3 == 1 ) = 0.95 END_OP
OP open_pump4 = 1 -> P( pump_ctrl_state4 == 1 ) = 0.95 END_OP
```

Listing 5.2: Following Program Orders

Furthermore, the water level is set to increase according the the number of active pumps in Listing 5.3. Since each pump has a throughput of 1 ltr/step, the water level must increase by 1 liter for each active pump at every step. The water level of the steam boiler is also affected by the quantity of exiting steam. The amount of water leaving as steam is subtracted from the water level in line 6. Finally, in line 8 when the water release valve is on, the water level is decreased at a constant rate of 4 liters/step.

```
1 OP pump_state1 = 1 -> P( level + 1 ) = 0.99 END_OP
2 OP pump_state2 = 1 -> P( level + 1 ) = 0.99 END_OP
3 OP pump_state3 = 1 -> P( level + 1 ) = 0.99 END_OP
4 OP pump_state4 = 1 -> P( level + 1 ) = 0.99 END_OP
5
6 P( level - steam ) = 1
7
8 OP valve = 1  -> P( level - 4 ) = 1 END_OP
```

Listing 5.3: Water Level

For sake of simplicity, the amount of steam leaving the boiler is modeled based on the current amount of water. Therefore the properties in Listing 5.4 provide a direct relationship between the water level and steam output.

```
OP level <= 150 -> P( steam == 1 ) = 1 END_OP
OP level <= 80 -> P( steam == 2 ) = 1 END_OP
OP level <= 65 -> P( steam == 3 ) = 1 END_OP
OP level <= 20 -> P( steam == 4 ) = 1 END_OP
OP level <= 10 -> P( steam == 5 ) = 1 END_OP
```

Listing 5.4: Steam Production

An unsolicited signal indicating a unit fault is acknowledged or has been repaired when the program has not sent a fault detected signal is considered an erroneous transmission and will force the program to *emergency stop* mode; therefore, to keep the environment running with minimal faults the signals by default are set false as is shown in Listing 5.5. Solicited acknowledgments however, are given a high probability of occurring. The “unit repaired” messages are treated similarly (not shown here).

```
pump_fault_ack1 = 0
pump_fault_ack2 = 0
pump_fault_ack3 = 0
pump_fault_ack4 = 0

pump_ctrl_fault_ack1 = 0
pump_ctrl_fault_ack2 = 0
pump_ctrl_fault_ack3 = 0
pump_ctrl_fault_ack4 = 0

level_fault_ack = 0
steam_outcome_fault_ack = 0

OP pump_fault_detected1$ == 1 -> P( pump_fault_ack1 = 1 ) = 1 END_OP
OP pump_fault_detected2$ == 1 -> P( pump_fault_ack2 = 1 ) = 1 END_OP
OP pump_fault_detected3$ == 1 -> P( pump_fault_ack3 = 1 ) = 1 END_OP
OP pump_fault_detected4$ == 1 -> P( pump_fault_ack4 = 1 ) = 1 END_OP

OP pump_ctrl_fault_detected1$ == 1 -> P( pump_ctrl_fault_ack1 = 1 )
    = 1 END_OP
OP pump_ctrl_fault_detected2$ == 1 -> P( pump_ctrl_fault_ack2 = 1 )
    = 1 END_OP
OP pump_ctrl_fault_detected3$ == 1 -> P( pump_ctrl_fault_ack3 = 1 )
    = 1 END_OP
OP pump_ctrl_fault_detected4$ == 1 -> P( pump_ctrl_fault_ack4 = 1 )
    = 1 END_OP

OP level_fault_detected$ == 1 -> P( level_fault_ack = 1 ) = 1 END_OP
OP steam_fault_detected$ == 1 -> P( steam_outcome_fault_ack = 1 ) =
    1 END_OP
```

Listing 5.5: Unsolicited and Solicited Messages

In the environment description presented, many actions that corresponds to the expected behavior are given a high, though not absolute, probability of occurring. This maintains that the system may still deviate from the typical operation and therefore does not restrict the behavior in its entirety. The purpose of such a specification is to create interesting observable events that may help expose bugs which are not directly being searched.

With the specified properties, a simulation with test length of 150 steps is conducted. The safety properties are temporarily ignored and discussed in the following section. The SAGA initialization file is written to provide a cold start of the steam boiler with an initial water level of 110. The resulting behavior is best observed by plotting the interesting data. The values for the `level` signal, which represents the water level, are observed in Figure 5.3 and the steam output is observed in Figure 5.4 by plotting the values for the `steam` signal.

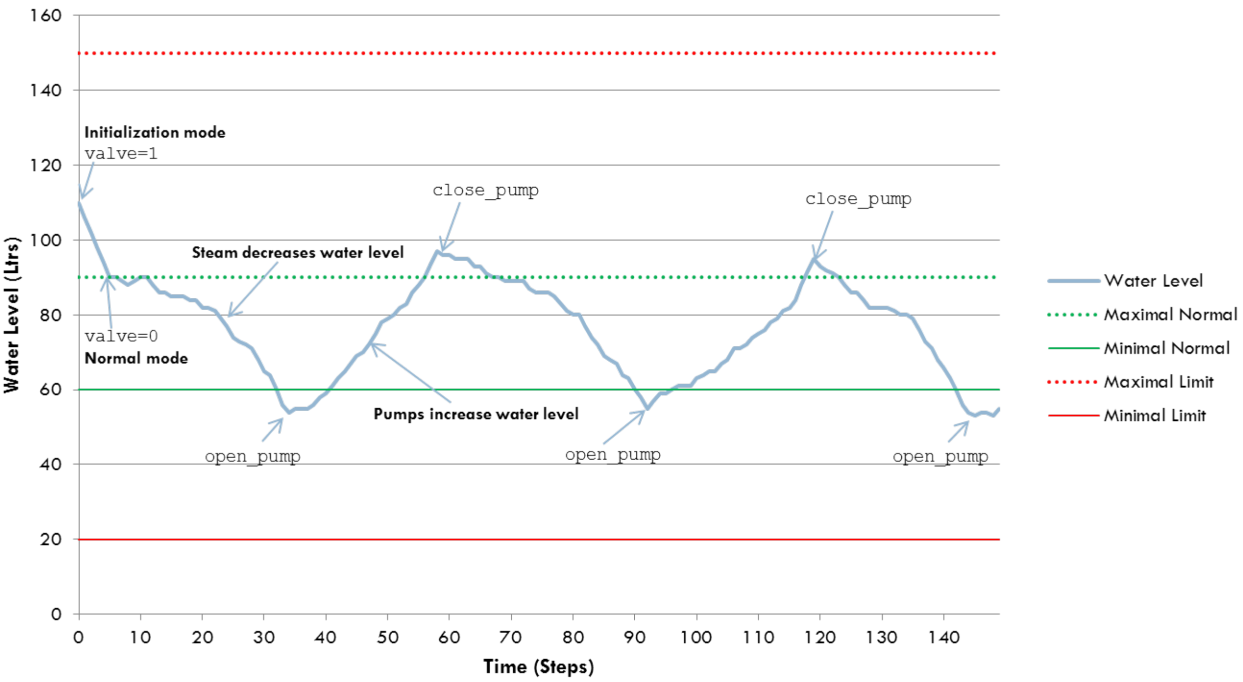


Figure 5.3: Steam Boiler Typical Operation: Water Level

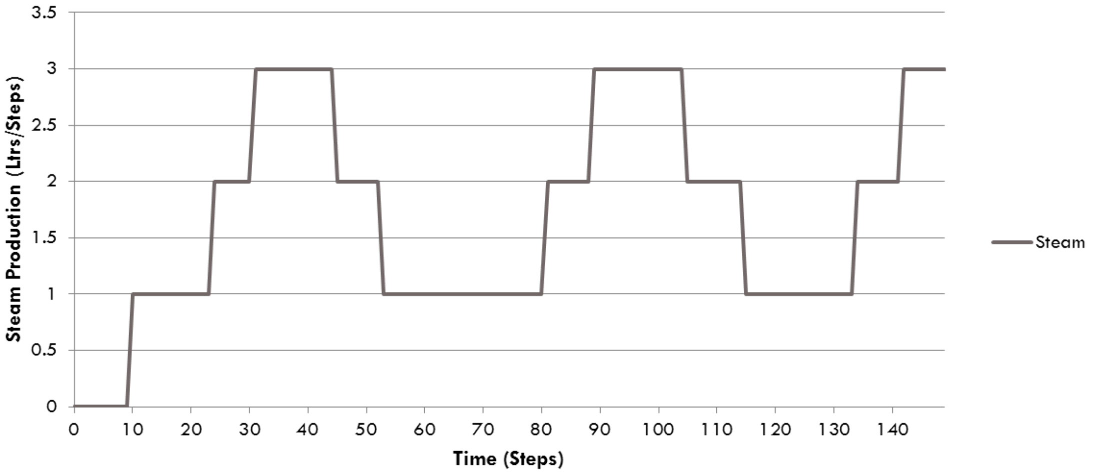


Figure 5.4: Steam Boiler Typical Operation: Steam Output

Given the initial water level value is above the *maximal normal* capacity, the steam boiler opens the water release valve until the level is between *minimal normal* and *maximal normal*. The program then issues the `program_ready` signal and enters *normal* mode. When in *normal* mode, the program effectively maintains the water level in the normal operating range by opening and closing pumps accordingly.

Stress Operation

In order to observe the operation of the steam boiler program under a chaotic scenario, an environment description that asserts the occurrence of a water leak is used. Using the previous description from Section 5.3.2 to initially maintain the typical system behavior, a single property, presented in Listing 5.6, can be added to force the desired behavior. The water leak in the steam boiler is effectively simulated using a *patter profile* whose *enter-condition* is satisfied when the current water level is at a specific value (in this case 65 liters). The pattern ensures that `level` is continually decreased until it reaches a very low value. Once the pattern profile is entered, the constraints that sequentially follow are satisfied in subsequent steps. This behavior is guaranteed because each constraint in the pattern has a 100% probability of being satisfied.

```
PATTERN
level = 65
  -> P( level - 4 ) = 1
  -> P( level - 7 ) = 1
  -> P( level - 9 ) = 1
  -> P( level - 11 ) = 1
  -> P( level - 13 ) = 1
END_PATTERN
```

Listing 5.6: Simulating a water leak

The result which the pattern profile effects in the simulation is shown in Figure 5.5, where the values of the `level` signal are plotted.

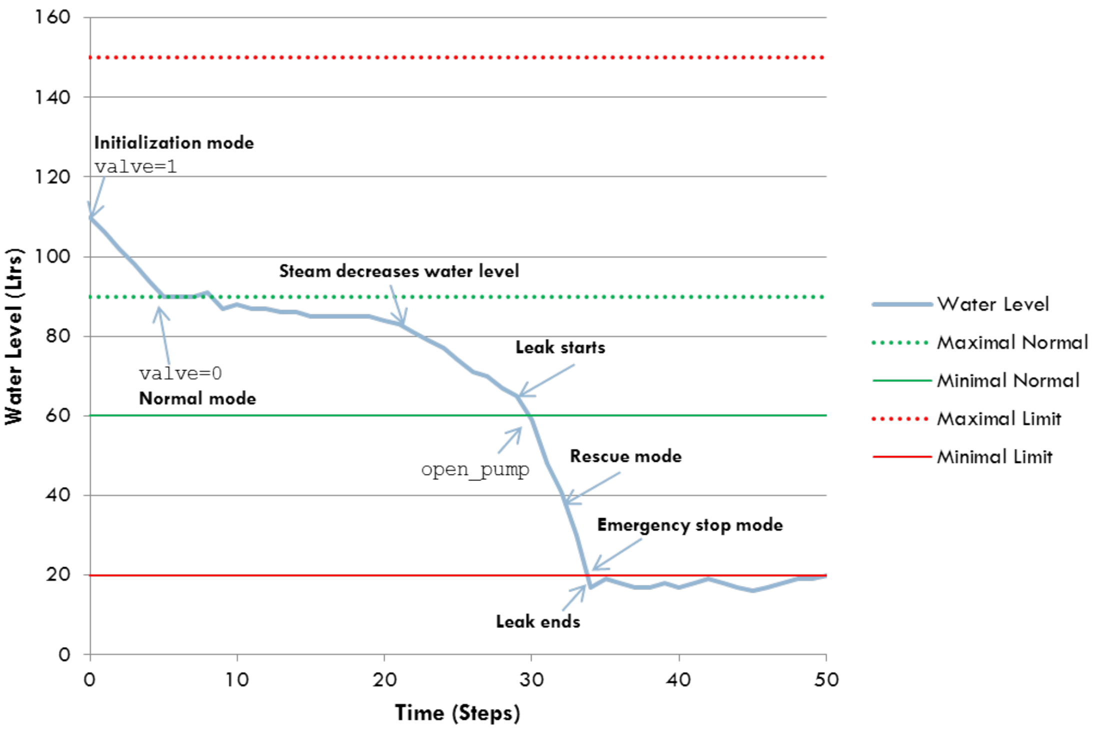


Figure 5.5: Steam Boiler Stress Operation: Water Leak

5.3.3 Nontrivial Bugs

The validation process for the program reveals some safety violations with the implementation contained in Appendix B. The nontrivial problems that were exposed are described. The problems associated with each violation were determined after inspection of the execution trace in the annotated simulation log. They are as follows:

- The program remains in normal mode when the pump controller only momentarily changes to a state which does not match the corresponding pump state. The proper behavior is for the program to enter *degraded* mode, given the equipment failure is detected. The program should have then asserted the `pump_ctrl_fault_detected` signal, in which case it must wait for an acknowledgment `pump_ctrl_fault_ack` and a subsequent repair message `pump_ctrl_repaired` in order to return to *normal* mode. This error was detected due to a violation of the safety property introduced in line 5 of Listing 5.1.

- When in *rescue* mode, the program may erroneously assert `open_pump` and `close_pump` back-to-back repeatedly as high and low water level estimates reach thresholds. The program enters *rescue* mode when a fault is detected in the water level measuring unit, at which point it must calculate the water level based on the amount of steam being produced and the throughput of the active pumps. The `open_pump` signal is asserted when the low estimate reaches the *minimal normal* limit and the current state of the pump is closed. The `close_pump` signal is asserted when the high estimate reaches the *maximal normal* limit and the current state of the pump is open. Therefore, when both the high and low estimates reach their limit, the program generates the problematic behavior. This problematic behavior was detected during an inspection of the system runtime activity in a simulation log.

CHAPTER 6

DESIGN AND IMPLEMENTATION DETAILS

This chapter presents information about some of the important design decisions and implementation details of SAGA. More specifically, it discusses why SIGNAL was chosen as the target language, how a testing harness is built for a simulation, how numerical probabilities are evaluated, and how environment generation and safety validation are handled. The information presented here is not meant to serve as an exhaustive design overview, but rather as a high-level description of some of the important technical details of SAGA.

6.1 Why SIGNAL?

SAGA was developed as part of an investigatory project in automatic test case generation from formal specifications. It has initially been designed to work with synchronous programs specified in the SIGNAL language. SIGNAL was chosen as the target synchronous language for several practical reasons. The SIGNAL compiler is incorporated into the Polychrony tool set whose binaries and source are freely distributed under the GPLv2 license [28]. Also, the language is well documented since it has accumulated an extensive amount of technical literature over the years. Although SIGNAL has mostly been used in academic research rather than industry, it has been gaining the recent attention from industry due to its multi-rate (polychronous) nature that makes it suitable for specifying parallel and distributed systems[12]. This aspect of SIGNAL makes it suitable for a future direction of this project in the validation of parallel and distributed systems.

6.2 Testing Harness

A black-box model of the system under test is used in SAGA for validation. In order to achieve dynamic synchronous control of the program being tested, SAGA creates a testing harness by making use of simulation executables generated from the Polychrony environment. The simulation executables are SIGNAL programs that have been compiled from code generated by the Polychrony environment. In order to run a simulation with the executables, a user must create a file for each input and output signal of the program. In the file, the user must enter the values for each signal at every step [11]. SAGA takes advantage of this feature by using the files as buffers in order to control the system under test. Figure 6.1 illustrates this concept.

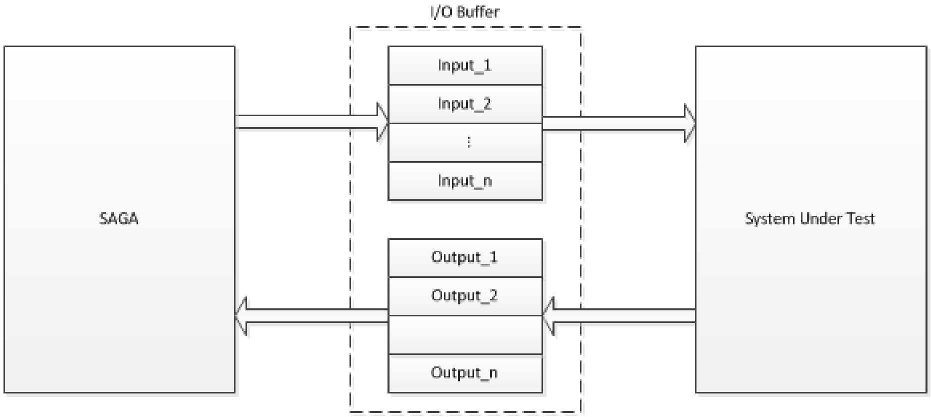


Figure 6.1: Testing Harness for Dynamic Synchronous Control

The intermediary files are created as named pipes in order to enable interprocess communication between SAGA and the system under test. A named pipe is a file type which allows multiple processes to simultaneously read and write from it. This mechanism allows SAGA to generate values for all input signal and read values from all output signals at every step, and therefore concurrently simulate a runtime environment.

6.3 Evaluating Probabilities

The use of probabilities in SAGA gives great flexibility to attributing realistic behaviors to an environment. It is important to understand on a quantitative basis the meaning of such an operation in order to understand its capabilities and limitations. The method of how probabilities are evaluated in SAGA is therefore presented here.

Given that a decision must be made with P_x, a probability basis for property x, then R_{xy}, a random number for property x during simulation step y, is generated by SAGA. The following corresponding inequality is then evaluated:

$$\frac{(R_{xy} \bmod 100) + 1}{100} \leq P_x$$

where $P_x \in \mathbb{R}$, $0 \leq P_x \leq 1$, and $R_{xy} \in \mathbb{Z}^+$.

If the inequality is true, property x is satisfied in simulation step y. The random number R_{xy} is generated using the pseudo-random sequence generation function $rand()$ from the standard C++ library. In order to provide distinct values for different simulations using the same initialization and environment description files, the random number generator is seeded with the current system time at the start of each simulation.

6.4 Handling Properties

With speed being a crucial aspect of simulation tools, many design decisions were made in consideration of running time. Generating the environment and checking safety from properties composes the bulk of the computation work. The work done by SAGA at each iteration is effectively bounded by the number of properties that must be satisfied or checked, and the total number of system input and output signals. In most cases, this number is relatively small compared to the total number of steps. Furthermore, since each property is initially parsed from its description file and stored in program memory, a file read operation bottle neck is avoided. The overall execution time of a simulation in SAGA is therefore primarily dependent on the total number of simulated steps (i.e. the user-specified test length).

Generating an environment consists of providing a value for every input signal to the system at every simulation step. The values selected must be consistent with the specified environment properties. At each step, every property in the list is evaluated in the manner described in Section 3.3. This effectively assigns values to all the signals which have properties associated with them. The set of possible values for the remaining signals is only restricted by the signal data type and specified initialization minimum and maximum values. These signals are therefore assigned a value randomly chosen from their possible set.

Checking safety in a simulation entails verifying the specified properties hold at every step in execution. In SAGA, each safety property is stored in a binary tree data structure when it is first parsed from the description file. The binary tree allows for an intuitive and organized way of checking the truth value of logical statements. For example, the property in Listing 6.1, taken from the steam boiler case study, is stored in SAGA as illustrated in Figure 6.2. The tree arrangement is described using the BNF symbols introduced in Section 4.4.1.

```
SAFE ( mode != 2 ) or ( ( level <= 130 ) and (level >= 20 ) )
   END_SAFE
```

Listing 6.1: A Safety Property

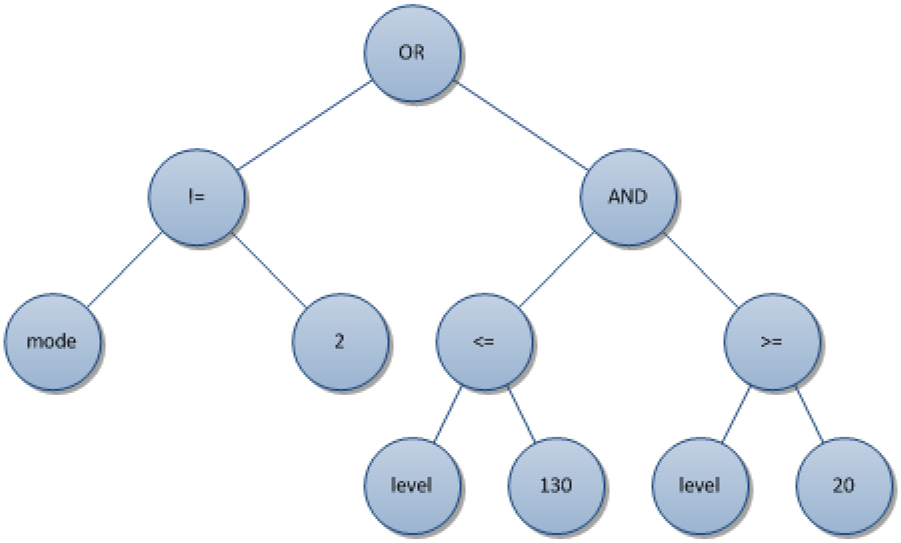


Figure 6.2: Binary Tree Representation of a Safety Property

When a safety property is stored in SAGA, the *operator* of the *first_expression* always comprises the root node of the tree. Every node in the tree corresponds to the *operator* in the local *expression*; therefore, each node must be composed of either two child nodes, two leaves, or a node and a leaf. The leaves in the tree can be either a *signal* or *value*. The tree structure is therefore always a full binary tree, where every node must have two children, but not a perfect binary tree because not all leaves will have the same depth level.

In order to check the truth value of a property, a recursive function is used to traverse the tree from the root node to the bottom-most nodes. Each node then returns a value determined by evaluating with its operator the values of its child nodes or leaves. The truth value of each node is computed as the function returns back to the root node. Once both child nodes of the root note have returned a value, the truth value of the property is determined.

CHAPTER 7

CONCLUSION

7.1 Summary and Conclusion

The main contribution of this thesis is the design, implementation, and qualitative assessment of SAGA, a prototype utility for the automated validation of SIGNAL programs. SAGA provides a framework which shifts the testing effort associated with critical systems from manually creating individual test cases to reasoning about the safety and environment properties of a system. An overview of the methodology and a detailed user's guide of SAGA is presented. In the user's guide, a thorough example of SAGA as applied to the validation of a cruise control system is included for completeness.

A qualitative assessment of the utility of SAGA is presented. The assessment consists of a case study of SAGA as applied to the validation of the celebrated steam boiler control specification problem, a system of substantial complexity. For this case study, an implementation of the steam boiler program is developed in the SIGNAL language. Results from the validation of the steam boiler program show that SAGA is able to reveal non-trivial errors in the design of the system-under-test.

7.1.1 Lessons from the Case Study

From the case study, the following important observations were made about the validation process in SAGA:

- Given an initial specification of a system, the validation process in SAGA can start concurrently with the design and development phases of the software development cycle. This promotes, in general, a good practice for the development of safe software systems.

- The SAGA environment description requires sufficient detail to expose interesting system behavior, but should not be overly constraining. Given a loosely specified environment where signal inputs are mostly generated randomly, a system will seldom evolve into more interesting states. Conversely, if signals are generated from an overly rigid specification, the system will always be tested against the same input routines. The appropriate level of environment specification must therefore be carefully considered and will be unique to each system test agenda.

- Writing safety properties need not be an entirely creative process. Many of the safety properties of a system can be systematically inferred from the system specification itself.

7.2 Future Work

Future work of interest concerning the development of SAGA involves primarily making technical improvements and additions. Three such improvements are key:

1. Full incorporation of temporal logic for the description of safety properties. SAGA currently handles statements about the immediate previous state through use of a delay operator, but does not support temporal propositions which make use of other notions of time such as *until* or *since*. Incorporating temporal logic for describing safety would provide more flexibility in the types of properties that may be specified.

2. Support for polychronous systems. The SIGNAL language allows for the specification of polychronous systems, where each input signal has its own activation clock. Currently, validation in SAGA is limited to single-clocked systems, where all signals share the same activation clock (i.e. a value is generated for all signals during every *step*). Adding support for the validation of polychronous systems would require the incorporation of environment description mechanisms which take into consideration the presence or absence (i.e. clock) of each signal. The polychronous model provides suitable means for describing distributed systems, a domain in which the application of automated validation is of interest.

3. Compatibility with other synchronous languages. SAGA has been initially developed to work with SIGNAL programs. Since SAGA performs black box testing on compiled executables, the methodology for validation effectively remains the same for all synchronous programs. The only accommodation required would be a suitable testing harness for each program type. Of the synchronous languages, Lustre seems the most appealing due to its wider audience and commercial success.

APPENDIX A

CRUISE CONTROL PROGRAM LISTING

```
% Cruise Control System
% Author: Vahid Rajabian Schwart

process cruiseControl = % ----- Cruise Control System ----- %
(     % inputs %
      ? integer speed;          % speed sensor %
        boolean set,            % turn ON cruise control %
                cancel,         % turn OFF cruise control %
                speed_inc,      % increase cruise speed %
                speed_dec;      % decrease cruise speed %
      % outputs %
      ! boolean throttle,       % throttle control %
                brake,          % brake control %
          ctrl_on_disp;) % control indicator %
(   % system specification %
  % clock restrictions %
  | throttle ^= brake ^= ctrl_on_disp ^= speed
  | speed ^= set ^= cancel ^= speed_inc ^= speed_dec

  | control_on := ( true when set )
    default ( false when cancel )
    default ( control_on$ init false )

  | ctrl_on_disp := control_on

  | cruise_speed := ( speed when set ) default ( (cruise_speed$ init 0)+1 when speed_inc )
    default ( (cruise_speed$ init 0)-1 when speed_dec )
    default ( cruise_speed$ init 0 )

  | throttle := ( false when ( (speed >= cruise_speed) and control_on) )
    default ( true when ((speed < cruise_speed) and control_on) )
    default false

  | brake := ( false when ((speed <= cruise_speed) and control_on) )
    default ( true when ( (speed > cruise_speed) and control_on) )
    default false

  | ) where % local variables %
    boolean control_on;      % cruise control state %
    integer cruise_speed;    % cruise control speed %
end % cruiseControl %
```

APPENDIX B

STEAM BOILER FILES

B.1 Initialization

```
%
% Steam Boiler Initialization
%

% Name of system under test
steamBoiler

% Number of Boolean input and output signals
% [No. input] [No. output]
31         30

% Number of Integer input and output signals
% [No. input] [No. output]
2       1

% Names of Boolean signals (must match SUT signal name) and initial value
% [name]  [init_value]
stop      0   % inputs
steam_boiler_waiting  0
physical_units_ready  0
pump_state1   0
pump_state2   0
pump_state3   0
pump_state4   0
pump_ctrl_state1  0
pump_ctrl_state2  0
pump_ctrl_state3  0
pump_ctrl_state4  0
pump_repaired1    0
```

```
pump_repaired2    0
pump_repaired3    0
pump_repaired4    0
pump_ctrl_repaired1 0
pump_ctrl_repaired2 0
pump_ctrl_repaired3 0
pump_ctrl_repaired4 0
level_repaired    0
steam_repaired    0
pump_fault_ack1   0
pump_fault_ack2   0
pump_fault_ack3   0
pump_fault_ack4   0
pump_ctrl_fault_ack1  0
pump_ctrl_fault_ack2  0
pump_ctrl_fault_ack3  0
pump_ctrl_fault_ack4  0
level_fault_ack   0
steam_outcome_fault_ack 0

program_ready 0   % outputs
valve     0
open_pump1    0
open_pump2    0
open_pump3    0
open_pump4    0
close_pump1   0
close_pump2   0
close_pump3   0
close_pump4   0
pump_fault_detected1  0
pump_fault_detected2  0
pump_fault_detected3  0
pump_fault_detected4  0
pump_ctrl_fault_detected1 0
pump_ctrl_fault_detected2 0
pump_ctrl_fault_detected3 0
pump_ctrl_fault_detected4 0
level_fault_detected  0
steam_fault_detected  0
pump_repaired_ack1  0
pump_repaired_ack2  0
pump_repaired_ack3  0
pump_repaired_ack4  0
pump_ctrl_repaired_ack1 0
pump_ctrl_repaired_ack2 0
pump_ctrl_repaired_ack3 0
pump_ctrl_repaired_ack4 0
level_repaired_ack  0
steam_repaired_ack  0

```

```
% Names of Integer signals (must match SUT signal name), initial value
% minimum value, maximum value, and linear constraint
% [name] [init_value]   [min_value]   [max_value] [linear]
level 110 0 900 1
steam 0 0 900 1
mode 1
```

B.2 Safety Description

```
%
% Safety Description
%

% Rescue or emergency stop mode if level sensor is faulty
SAFE ( ( level <= 150 ) and ( level >= 0 ) ) or ( ( mode = 4) or ( mode = 5 ) ) END_SAFE

% Emergency mode if stop is asserted
SAFE ( mode = 5 ) or ( stop != 1 ) END_SAFE

% Cannot be in normal mode unless water level is near max and min normal
SAFE ( mode != 2 ) or ( ( level <= 110 ) and (level >= 40 ) ) END_SAFE

% Valve must only be used in initialization
SAFE ( valve = 0 ) or ( mode = 1 ) END_SAFE

% Pumps are working, or the program must be in degraded, rescue or emergency stop mode
SAFE ( pump_state1 = pump_ctrl_state1 ) or ( ( ( mode = 3 ) or ( mode = 4 ) ) or ( mode = 5) )  END_SAFE
SAFE ( pump_state2 = pump_ctrl_state2 ) or ( ( ( mode = 3 ) or ( mode = 4 ) ) or ( mode = 5) )  END_SAFE
SAFE ( pump_state3 = pump_ctrl_state3 ) or ( ( ( mode = 3 ) or ( mode = 4 ) ) or ( mode = 5) )  END_SAFE
SAFE ( pump_state4 = pump_ctrl_state4 ) or ( ( ( mode = 3 ) or ( mode = 4 ) ) or ( mode = 5) )  END_SAFE

% Program must acknowledge messages
SAFE ( ( pump_repaired1$ == 1 ) and ( pump_repaired_ack1 == 1 ) ) or ( pump_repaired1$ == 0 ) END_SAFE
SAFE ( ( pump_repaired2$ == 1 ) and ( pump_repaired_ack2 == 1 ) ) or ( pump_repaired2$ == 0 ) END_SAFE
SAFE ( ( pump_repaired3$ == 1 ) and ( pump_repaired_ack3 == 1 ) ) or
     ( pump_repaired3$ == 0 ) END_SAFE
SAFE ( ( pump_repaired4$ == 1 ) and ( pump_repaired_ack4 == 1 ) ) or
     ( pump_repaired4$ == 0 ) END_SAFE

SAFE ( ( pump_ctrl_repaired1$ == 1 ) and ( pump_ctrl_repaired_ack1
    == 1 ) ) or ( pump_ctrl_repaired1$ == 0 )  END_SAFE
SAFE ( ( pump_ctrl_repaired2$ == 1 ) and ( pump_ctrl_repaired_ack2
    == 1 ) ) or ( pump_ctrl_repaired2$ == 0 )  END_SAFE
SAFE ( ( pump_ctrl_repaired3$ == 1 ) and ( pump_ctrl_repaired_ack3
    == 1 ) ) or ( pump_ctrl_repaired3$ == 0 )  END_SAFE
SAFE ( ( pump_ctrl_repaired4$ == 1 ) and ( pump_ctrl_repaired_ack4
    == 1 ) ) or ( pump_ctrl_repaired4$ == 0 )  END_SAFE

SAFE ( ( level_repaired$ == 1 ) and ( level_repaired_ack == 1 ) ) or
     ( level_repaired$ == 0 ) END_SAFE
SAFE ( ( steam_repaired$ == 1 ) and ( steam_repaired_ack == 1 ) ) or
     ( steam_repaired$ == 0 ) END_SAFE

% Program should not open and close the same pump
SAFE ( open_pump1 = 0 ) or ( close_pump1 = 0 ) END_SAFE
SAFE ( open_pump2 = 0 ) or ( close_pump2 = 0 ) END_SAFE
SAFE ( open_pump3 = 0 ) or ( close_pump3 = 0 ) END_SAFE
SAFE ( open_pump4 = 0 ) or ( close_pump4 = 0 ) END_SAFE
```

B.3 Environment Description

```
%
% Steam Boiler Environment
%

% Respond to program ready
OP program_ready = 1 -> P(  physical_units_ready = 1 ) = 0.99 END_OP

% Infrequent assertion of stop
P( stop = 1 ) = 0.01

% No steam output during initialization
OP mode = 1 -> P( steam = 0 ) = 1 END_OP

% Decrease water level when valve is open
OP valve = 1  -> P( level - 4 ) = 0.98 END_OP

% No unsolicited messages
 steam_boiler_waiting = 0
 physical_units_ready = 0
 pump_repaired1 = 0
 pump_repaired2 = 0
 pump_repaired3 = 0
 pump_repaired4 = 0
 pump_ctrl_repaired1 = 0
 pump_ctrl_repaired2 = 0
 pump_ctrl_repaired3 = 0
 pump_ctrl_repaired4 = 0
 level_repaired = 0
steam_repaired = 0
pump_fault_ack1 = 0
pump_fault_ack2 = 0
pump_fault_ack3 = 0
pump_fault_ack4 = 0
pump_ctrl_fault_ack1 = 0
pump_ctrl_fault_ack2 = 0
pump_ctrl_fault_ack3 = 0
pump_ctrl_fault_ack4 = 0
level_fault_ack = 0
steam_outcome_fault_ack = 0

% Close/open pumps according to program orders
OP close_pump1 = 1 -> P( pump_state1 == 0 ) = 0.95 END_OP
OP close_pump2 = 1 -> P( pump_state2 == 0 ) = 0.95 END_OP
OP close_pump3 = 1 -> P( pump_state3 == 0 ) = 0.95 END_OP
OP close_pump4 = 1 -> P( pump_state4 == 0 ) = 0.95 END_OP
OP open_pump1 = 1 -> P( pump_state1 == 1 ) = 0.95 END_OP
OP open_pump2 = 1 -> P( pump_state2 == 1 ) = 0.95 END_OP
OP open_pump3 = 1 -> P( pump_state3 == 1 ) = 0.95 END_OP
OP open_pump4 = 1 -> P( pump_state4 == 1 ) = 0.95 END_OP

OP close_pump1 = 1 -> P( pump_ctrl_state1 == 0 ) = 0.95 END_OP
OP close_pump2 = 1 -> P( pump_ctrl_state2 == 0 ) = 0.95 END_OP
OP close_pump3 = 1 -> P( pump_ctrl_state3 == 0 ) = 0.95 END_OP
OP close_pump4 = 1 -> P( pump_ctrl_state4 == 0 ) = 0.95 END_OP
OP open_pump1 = 1 -> P( pump_ctrl_state1 == 1 ) = 0.95 END_OP
OP open_pump2 = 1 -> P( pump_ctrl_state2 == 1 ) = 0.95 END_OP
OP open_pump3 = 1 -> P( pump_ctrl_state3 == 1 ) = 0.95 END_OP
OP open_pump4 = 1 -> P( pump_ctrl_state4 == 1 ) = 0.95 END_OP

% Keep pump state constant unless changed by system
P( pump_state1 == pump_state1$ ) = 0.99
P( pump_state1 == pump_state2$ ) = 0.99
P( pump_state1 == pump_state3$ ) = 0.99
P( pump_state1 == pump_state4$ ) = 0.99

% Keep pump control state constant unless changed by system
P( pump_ctrl_state1 == pump_ctrl_state1$ ) = 0.99
P( pump_ctrl_state1 == pump_ctrl_state2$ ) = 0.99
P( pump_ctrl_state1 == pump_ctrl_state3$ ) = 0.99
```

```
P( pump_ctrl_state1 == pump_ctrl_state4$ ) = 0.99

% Increase water level with each pump
OP pump_state1 = 1 -> P( level + 1 ) = 0.99 END_OP
OP pump_state2 = 1 -> P( level + 1 ) = 0.99 END_OP
OP pump_state3 = 1 -> P( level + 1 ) = 0.99 END_OP
OP pump_state4 = 1 -> P( level + 1 ) = 0.99 END_OP

% Decrease water level according to steam production %
OP steam != 0 -> P( level - steam ) = 1 END_OP % delete below lines

% Steam output increases as level decreases
OP level <= 150 -> P( steam == 1 ) = 1 END_OP
OP level <= 80 -> P( steam == 2 ) = 1 END_OP
OP level <= 65 -> P( steam == 3 ) = 1 END_OP
OP level <= 20 -> P( steam == 4 ) = 1 END_OP
OP level <= 10 -> P( steam == 5 ) = 1 END_OP

% Transmission responses
OP pump_fault_detected1$ == 1 -> P( pump_fault_ack1 = 1 ) = 1 END_OP
OP pump_fault_detected2$ == 1 -> P( pump_fault_ack2 = 1 ) = 1 END_OP
OP pump_fault_detected3$ == 1 -> P( pump_fault_ack3 = 1 ) = 1 END_OP
OP pump_fault_detected4$ == 1 -> P( pump_fault_ack4 = 1 ) = 1 END_OP

OP pump_ctrl_fault_detected1$ == 1 -> P( pump_ctrl_fault_ack1 = 1 ) = 1 END_OP
OP pump_ctrl_fault_detected2$ == 1 -> P( pump_ctrl_fault_ack2 = 1 ) = 1 END_OP
OP pump_ctrl_fault_detected3$ == 1 -> P( pump_ctrl_fault_ack3 = 1 ) = 1 END_OP
OP pump_ctrl_fault_detected4$ == 1 -> P( pump_ctrl_fault_ack4 = 1 ) = 1 END_OP

OP level_fault_detected$ == 1 -> P( level_fault_ack = 1 ) = 1 END_OP
OP steam_fault_detected$ == 1 -> P( steam_outcome_fault_ack = 1 ) = 1 END_OP
```

B.4 Program Listing

```
% Steam boiler controller initial implementation
% Author: Vahid Rajabian Schwart
% Implemented from the specification presented
% in: Jean-Raymond Abrial, Egon B rger, and Hans Langmaack.
%     Formal Methods for Industrial Applications:
%     Specifying and Programming the Steam Boiler Control.
%     LNCS 1165, Springer-Verlag, October 1996.

process steamBoiler =
( % inputs %
  ? integer level,
    steam;

    boolean stop,
    steam_boiler_waiting,
    physical_units_ready,
    pump_state1, pump_state2,
    pump_state3, pump_state4,
    pump_ctrl_state1, pump_ctrl_state2,
    pump_ctrl_state3, pump_ctrl_state4,
    pump_repaired1, pump_repaired2,
    pump_repaired3, pump_repaired4,
    pump_ctrl_repaired1, pump_ctrl_repaired2,
    pump_ctrl_repaired3, pump_ctrl_repaired4,
    level_repaired,
    steam_repaired,
    pump_fault_ack1, pump_fault_ack2,
    pump_fault_ack3, pump_fault_ack4,
    pump_ctrl_fault_ack1, pump_ctrl_fault_ack2,
    pump_ctrl_fault_ack3, pump_ctrl_fault_ack4,
    level_fault_ack,
    steam_outcome_fault_ack;

  % outputs %
  ! integer mode;

    boolean program_ready,
    valve,
    open_pump1, open_pump2, open_pump3, open_pump4,
    close_pump1, close_pump2, close_pump3, close_pump4,
    pump_fault_detected1, pump_fault_detected2,
    pump_fault_detected3, pump_fault_detected4,
    pump_ctrl_fault_detected1, pump_ctrl_fault_detected2,
    pump_ctrl_fault_detected3, pump_ctrl_fault_detected4,
    level_fault_detected,
    steam_fault_detected,
    pump_repaired_ack1, pump_repaired_ack2,
    pump_repaired_ack3, pump_repaired_ack4,
    pump_ctrl_repaired_ack1, pump_ctrl_repaired_ack2,
    pump_ctrl_repaired_ack3, pump_ctrl_repaired_ack4,
    level_repaired_ack,
    steam_repaired_ack;

)
  % clock equations: model as single-clocked system %
(
 | mode ^= program_ready ^= valve ^= open_pump1 ^= open_pump2 ^= open_pump3 ^=
  open_pump4 ^= close_pump1 ^= close_pump2 ^= close_pump3 ^= close_pump4 ^=
     pump_fault_detected1 ^= pump_fault_detected2 ^= pump_fault_detected3 ^=
  pump_fault_detected4 ^= pump_ctrl_fault_detected1 ^=
  pump_ctrl_fault_detected2 ^= pump_ctrl_fault_detected3 ^=
  pump_ctrl_fault_detected4 ^=  level_fault_detected ^=
  steam_fault_detected ^= pump_repaired_ack1 ^= pump_repaired_ack2 ^=
  pump_repaired_ack3 ^= pump_repaired_ack4 ^= pump_ctrl_repaired_ack1 ^=
     pump_ctrl_repaired_ack2 ^= pump_ctrl_repaired_ack3 ^=
  pump_ctrl_repaired_ack4 ^= level_repaired_ack ^= steam_repaired_ack ^=
  level ^= steam ^= stop ^= steam_boiler_waiting ^= physical_units_ready ^=
  pump_state1 ^= pump_state2 ^= pump_state3 ^= pump_state4 ^=
  pump_ctrl_state1 ^= pump_ctrl_state2 ^= pump_ctrl_state3 ^=
  pump_ctrl_state4 ^= pump_repaired1 ^= pump_repaired2 ^= pump_repaired3 ^=
     pump_repaired4 ^= pump_ctrl_repaired1 ^= pump_ctrl_repaired2 ^=
  pump_ctrl_repaired3 ^= pump_ctrl_repaired4 ^= level_repaired ^=
  steam_repaired ^= pump_fault_ack1 ^= pump_fault_ack2 ^= pump_fault_ack3 ^=
  pump_fault_ack4 ^= level_fault_ack ^= steam_outcome_fault_ack ^=
  pump_fault1 ^= pump_fault2 ^= pump_fault3 ^= pump_fault4 ^=
  pump_ctrl_fault1 ^= pump_ctrl_fault2 ^= pump_ctrl_fault3 ^=
  pump_ctrl_fault4 ^= water_level_measuring_unit_fault ^=
  steam_level_measuring_unit_fault ^= transmission_fault ^=
  steam_boiler_waiting_received ^= physical_units_ready_received ^=
  rescue_estimate_high ^= rescue_estimate_low ^= pump_count ^= pmp1 ^=
  pmp2 ^= pmp3 ^= pmp4 ^= pump_fault_count ^= pmpflt1 ^= pmpflt2 ^=
  pmpflt3 ^= pmpflt4 ^= steam_delta ^= level_delta

  % program mode %
 | mode := EMERGENCY_MODE when ( stop
    or steam_level_measuring_unit_fault
    or transmission_fault
    or ( level >= (M2-10) ) or ( level <= (M1+10) )
    or ((mode$ init INIT_MODE = INIT_MODE) and (steam/=0) )
    or ( water_level_measuring_unit_fault and ((mode$ init INIT_MODE) = INIT_MODE) )
    or ( (pump_fault1 or pump_ctrl_fault1)
      and (pump_fault2 or pump_ctrl_fault2)
      and (pump_fault3 or pump_ctrl_fault3)
      and (pump_fault4 or pump_ctrl_fault4) ) ) default
  RESCUE_MODE when ( physical_units_ready_received$ init false
    and water_level_measuring_unit_fault ) default
```

```
  DEGRADED_MODE when ( physical_units_ready_received$ init false
    and not(water_level_measuring_unit_fault)
    and (pump_fault1 or pump_fault2 or pump_fault3 or pump_fault4
        or pump_ctrl_fault1 or pump_ctrl_fault2
        or pump_ctrl_fault3 or pump_ctrl_fault4)  ) default
  NORMAL_MODE when ( physical_units_ready_received$ init false )
  default mode$ init INIT_MODE

  % control  %
 | valve := ( true when ( level > N2 ) default false ) when ( mode = INIT_MODE )
    default false

| open_pump1 := ( true  when ((level < N1) and (pump_state1=false)) default false )
      when ( mode=NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_low < N1) and (pump_state1=false)) default false ) when ( mode = RESCUE_MODE )
    default false
| open_pump2 :=  ( true  when ((level < N1) and (pump_state2=false)) default false )
      when ( mode=NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_low < N1) and (pump_state2=false)) default false ) when ( mode = RESCUE_MODE )
    default false
| open_pump3 :=  ( true  when ((level < N1) and (pump_state3=false)) default false )
      when ( mode=NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_low < N1) and (pump_state3=false)) default false ) when ( mode = RESCUE_MODE )
    default false
| open_pump4 :=  ( true  when ((level < N1) and (pump_state4=false)) default false )
      when ( mode=NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_low < N1) and (pump_state4=false)) default false ) when ( mode = RESCUE_MODE )
    default false

| close_pump1 := ( true  when ((level > N2) and (pump_state1=true)) default false )
      when ( mode=NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_high > N2 and (pump_state1=true)) default false ) when ( mode = RESCUE_MODE )
    default false
| close_pump2 := ( true  when ((level > N2) and (pump_state2=true)) default false )
```

```
      when ( mode=NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_high > N2 and (pump_state2=true)) default false ) when ( mode = RESCUE_MODE )
    default false
| close_pump3 := ( true  when ((level > N2) and (pump_state3=true)) default false )
      when ( mode = NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_high > N2 and (pump_state3=true)) default false ) when ( mode = RESCUE_MODE )
    default false
| close_pump4 := ( true  when ((level > N2) and (pump_state4=true)) default false )
      when ( mode=NORMAL_MODE or mode=DEGRADED_MODE or mode=INIT_MODE )
    default ( true when ((rescue_estimate_high > N2 and (pump_state4=true)) default false ) when ( mode = RESCUE_MODE )
    default false

  % rescue mode calculations %
    % estimate water level %
    % high estimate - assume faulty pumps are open %
 | rescue_estimate_high := (level$ init 0) when ( water_level_measuring_unit_fault
    and not(water_level_measuring_unit_fault$ init false))
  default ( (rescue_estimate_high$ init 0) - steam + pump_count + pump_fault_count )
    when ( water_level_measuring_unit_fault )
  default level
    % low estimate - assume faulty pumps are closed %
 | rescue_estimate_low := (level$ init 0) when ( water_level_measuring_unit_fault
    and not(water_level_measuring_unit_fault$ init false))
  default ( (rescue_estimate_low$ init 0) - steam + pump_count)
    when ( water_level_measuring_unit_fault )
  default level

    % count working pumps %
 | pmp1 := 0 when ( (pump_fault1 or pump_ctrl_fault1) and not(pump_state1$ init false) ) default 1
 | pmp2 := 0 when ( (pump_fault2 or pump_ctrl_fault2) and not(pump_state2$ init false) ) default 1
 | pmp3 := 0 when ( (pump_fault3 or pump_ctrl_fault3) and not(pump_state3$ init false) ) default 1
 | pmp4 := 0 when ( (pump_fault4 or pump_ctrl_fault4) and not(pump_state4$ init false) ) default 1
 | pump_count := ( pmp1 + pmp2 + pmp3 + pmp4 )

    % count faulty pump units ( controller or pump )  %
 | pmpflt1 := 1 when ( pump_fault1 or pump_ctrl_fault1) default 0
 | pmpflt2 := 1 when ( pump_fault2 or pump_ctrl_fault2) default 0
 | pmpflt3 := 1 when ( pump_fault3 or pump_ctrl_fault3) default 0
 | pmpflt4 := 1 when ( pump_fault4 or pump_ctrl_fault4) default 0
 | pump_fault_count := ( pmpflt1 + pmpflt2 + pmpflt3 + pmpflt4 )

  % communications %
 | program_ready := ( true when ( steam_boiler_waiting_received and (steam = 0 )
      and ( level < N2 ) and ( level > N1 )  ) default false)
      when ( mode = INIT_MODE )
    default false

    % Acknowledge messages %
| pump_repaired_ack1 := true when ( pump_repaired1$ init false ) default false
| pump_repaired_ack2 := true when ( pump_repaired2$ init false ) default false
| pump_repaired_ack3 := true when ( pump_repaired3$ init false ) default false
| pump_repaired_ack4 := true when ( pump_repaired4$ init false ) default false

| pump_ctrl_repaired_ack1 := true when ( pump_ctrl_repaired1$ init false ) default false
| pump_ctrl_repaired_ack2 := true when ( pump_ctrl_repaired2$ init false ) default false
| pump_ctrl_repaired_ack3 := true when ( pump_ctrl_repaired3$ init false ) default false
| pump_ctrl_repaired_ack4 := true when ( pump_ctrl_repaired4$ init false ) default false

| level_repaired_ack := true when ( level_repaired$ init false ) default false
| steam_repaired_ack  := true when ( steam_repaired$ init false ) default false

  % fault detection %
    % pump faults %
 | pump_fault1 := true when pump_fault_detected1
    default false when pump_repaired1
    default pump_fault1$ init false
 | pump_fault2 := true when pump_fault_detected2
    default false when pump_repaired2
    default pump_fault2$ init false
 | pump_fault3 := true when pump_fault_detected3
    default false when pump_repaired3
    default pump_fault3$ init false
 | pump_fault4 := true when pump_fault_detected4
    default false when pump_repaired4
    default pump_fault4$ init false

    % pump controller faults %
 | pump_ctrl_fault1 := true when pump_ctrl_fault_detected1
    default false when pump_ctrl_repaired1
    default pump_ctrl_fault1$ init false
 | pump_ctrl_fault2 := true when pump_ctrl_fault_detected2
    default false when pump_ctrl_repaired2
    default pump_ctrl_fault2$ init false
 | pump_ctrl_fault3 := true when pump_ctrl_fault_detected3
    default false when pump_ctrl_repaired3
    default pump_ctrl_fault3$ init false
 | pump_ctrl_fault4 := true when pump_ctrl_fault_detected4
    default false when pump_ctrl_repaired4
    default pump_ctrl_fault4$ init false

    % pump unit faults 1) non-responsiveness 2) spontaneous change %
| pump_ctrl_fault_detected1 := true when ( ((open_pump1$ init false) and not(pump_ctrl_state1))
    or ((close_pump1$ init false) and pump_ctrl_state1) )
  default true when ( (open_pump1$ init false or close_pump1$ init false)
    and (pump_ctrl_state1 = pump_ctrl_state1$ init false) )
  default false
| pump_ctrl_fault_detected2 := true when ( ((open_pump2$ init false) and not(pump_ctrl_state2))
    or ((close_pump2$ init false) and pump_ctrl_state2) )
  default true when ( (open_pump2$ init false or close_pump2$ init false)
    and (pump_ctrl_state2 = pump_ctrl_state2$ init false) )
  default false
| pump_ctrl_fault_detected3 := true when ( ((open_pump3$ init false) and not(pump_ctrl_state3))
    or ((close_pump3$ init false) and pump_ctrl_state3) )
  default true when ( (open_pump3$ init false or close_pump3$ init false)
    and (pump_ctrl_state3 = pump_ctrl_state3$ init false) )
  default false
| pump_ctrl_fault_detected4 := true when ( ((open_pump4$ init false) and not(pump_ctrl_state4))
    or ((close_pump4$ init false) and pump_ctrl_state4) )
  default true when ( (open_pump4$ init false or close_pump4$ init false)
    and (pump_ctrl_state4 = pump_ctrl_state4$ init false) )
  default false

| pump_fault_detected1 := false when ( pump_ctrl_fault1 )
  default true when ( ((open_pump1$ init false) and not(pump_state1))
    or ((close_pump1$ init false) and pump_state1) )
  default false
| pump_fault_detected2 := false when ( pump_ctrl_fault2 )
```

```
  default true when ( ((open_pump2$ init false) and not(pump_state2) )
    or ((close_pump2$ init false) and pump_state2) )
  default false
| pump_fault_detected3 :=  false when ( pump_ctrl_fault3 )
  default true when ( ((open_pump3$ init false) and not(pump_state3) )
    or ((close_pump3$ init false) and pump_state3) )
  default false
| pump_fault_detected4 := false when ( pump_ctrl_fault4 )
  default true when ( ((open_pump4$ init false) and not(pump_state4) )
    or ((close_pump4$ init false) and pump_state4) )
  default false

    % measure faults 1) known system capacity exceed %
    % 2) delta change inconsistent with physical system %
| level_fault_detected := true when ( (level < 0) or (level > C)
  or (level_delta$ init 0) > 4 ) default false
| level_delta := (level - (level$ init 110)) when (level > (level$ init 110)) %Must provide init value%
  default ( (level$ init 110) - level) when (level < (level$ init 110))
  default 0

| steam_fault_detected := true when ( (steam$ init 0 < 0) or (steam$ init 0 > C)
  or (steam_delta$ init 0) > 4 ) default false
| steam_delta := (steam - (steam$ init 0)) when (steam > (steam$ init 0)) %Must provide init value%
  default ((steam$ init 0) - steam) when (steam < (steam$ init 0))
  default 0


    % water level measuring unit fault %
 | water_level_measuring_unit_fault := true when level_fault_detected
    default false when level_repaired
    default water_level_measuring_unit_fault$ init false

    % steam level measuring unit fault %
 | steam_level_measuring_unit_fault := true when steam_fault_detected
    default false when steam_repaired
    default steam_level_measuring_unit_fault$ init false

    % transmission fault: No response %
 | transmission_fault := true when ( ((pump_fault_detected1$ init false) and (not pump_fault_ack1))
  or ((pump_fault_detected2$ init false) and (not pump_fault_ack2))
  or ((pump_fault_detected3$ init false) and (not pump_fault_ack3))
  or ((pump_fault_detected4$ init false) and (not pump_fault_ack4))
  or ((pump_ctrl_fault_detected1$ init false) and (not pump_ctrl_fault_ack1))
  or ((pump_ctrl_fault_detected2$ init false) and (not pump_ctrl_fault_ack2))
  or ((pump_ctrl_fault_detected3$ init false) and (not pump_ctrl_fault_ack3))
  or ((pump_ctrl_fault_detected4$ init false) and (not pump_ctrl_fault_ack4))
  or ((level_fault_detected$ init false) and (not level_fault_ack))
  or ((steam_fault_detected$ init false) and (not steam_outcome_fault_ack))
    % transmission fault: Unsolicited response %
  or ((not pump_fault1) and pump_fault_ack1)
  or ((not pump_fault2) and pump_fault_ack2)
  or ((not pump_fault3) and pump_fault_ack3)
  or ((not pump_fault4) and pump_fault_ack4)
  or ((not pump_ctrl_fault1) and pump_ctrl_fault_ack1)
  or ((not pump_ctrl_fault2) and pump_ctrl_fault_ack2)
  or ((not pump_ctrl_fault3) and pump_ctrl_fault_ack3)
  or ((not pump_ctrl_fault4) and pump_ctrl_fault_ack4)
  or ((not water_level_measuring_unit_fault) and level_fault_ack)
  or ((not steam_level_measuring_unit_fault) and steam_outcome_fault_ack) )
  default false

  % track message information %
 | steam_boiler_waiting_received := true when steam_boiler_waiting
  default steam_boiler_waiting_received$ init false

 | physical_units_ready_received := true when physical_units_ready
  default physical_units_ready_received$ init false


|) where
  % physical constants %
  constant integer C = 150;   % maximal boiler capacity %
  constant integer M1 = 20; % minimal limit %
  constant integer M2 = 130;  % maximal limit %
  constant integer N1 = 60; % minimal normal %
  constant integer N2 = 90; % maximal normal %

  % program states %
  constant integer INIT_MODE = 1;
  constant integer NORMAL_MODE = 2;
  constant integer DEGRADED_MODE = 3;
  constant integer RESCUE_MODE = 4;
  constant integer EMERGENCY_MODE = 5;

  % local variables %
    % system faults %
  boolean pump_fault1, pump_fault2,
    pump_fault3, pump_fault4,
    pump_ctrl_fault1, pump_ctrl_fault2,
    pump_ctrl_fault3, pump_ctrl_fault4,
    water_level_measuring_unit_fault,
    steam_level_measuring_unit_fault,
    transmission_fault,
    % transmission data %
    steam_boiler_waiting_received init false,
    physical_units_ready_received init false;

    % physical data %
  integer level_delta init 0,
    steam_delta init 0,
    rescue_estimate_high init 0,
    rescue_estimate_low init 0,
    pmp1, pmp2, pmp3, pmp4,
    pump_count init 0,
    pmpflt1, pmpflt2, pmpflt3, pmpflt4,
    pump_fault_count init 0;

end; % steamBoiler %
```

BIBLIOGRAPHY

[1] System design and analysis. Technical report, Federal Aviation Administration, June 21 1988. 1

[2] J.-R. Abrial. Steam-boiler control specification problem. *Technical report, F-75014*, August 1994. 36, 37

[3] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grown out of a Dagstuhl Seminar, June 1995)*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996. 36

[4] A. Belinfante, L. Frantzen, and C. Schallhart. *Tools for Test Case Generation*, chapter 14, pages 64–83. Springer Verlag, 2005. 3, 4

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. LeGuernic, and R. De Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 2003. 6, 7

[6] L. Besnard, T. Gautier, and P. Le Guernic. Signal v4—inria version: Reference, January 2012. 9

[7] L. Besnard, H. Marchand, and E. Rutten. The sigali tool box environment. In *Discrete Event Systems, 2006 8th International Workshop on*, pages 465 –466, July 2006. 4

[8] J. Bijoy and S. Shukla. An alternative polychronous model and synthesis methodology for model-driven. *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, 2010. 6

[9] J. P. Bowen. The ethics of safety-critical systems. *Communications of the ACM*, 43:91–97, 2000. 1

[10] S. Das. *Predicate Abstraction.* PhD thesis, Stanford University, 2003. 2

[11] A. Gamatie. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification.* Springer, 2010. 6, 7, 8, 9, 14, 51

[12] A. Gamatie and T. Gautier. The signal synchronous multiclock approach to the design of distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, May 2010. Vol. 21 No. 5. 7, 8, 51

[13] N. Halbwachs. Synchronous programming of reactive systems—a tutorial and commented bibliography. In *In Tenth International Conference on Computer-Aided Verification, CAV98, Vancouver (B.C.), LNCS 1427*, pages 1–16. Springer Verlag, 1998. 6, 8

[14] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente*. Springer Verlag, 1993. 8

[15] D. Jackson, M. Thomas, and Committee on Certifiably Dependable Software Systems National Research Council L. I. Millett, Editors. *Software for Dependable Systems: Sufficient Evidence?* The National Academies Press, 2007. 1

[16] B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Commun. ACM*, 36:69–80, November 1993. 1

[17] F. Ouabdesselam and I. Parissis. Testing techniques for data-flow synchronous programs. In *Proceedings of the Second International Workshop on Automated and Algorithmic Debugging*, 1995. 4

[18] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. *Proceedings 19th IEEE Real-Time Systems Symposium*, 1998. 4, 7

[19] M. Rovatsos and G. Weib. *Handbook of Software Engineering and Knowledge Engineering*, volume Volume 3: Recent Advances, chapter Autonomous Software, page 6384. World Scientific Publishing, River Edge, NJ, 2005. 1

[20] RT-Builder. Geensoft. `http://geensys.com/rt-builder-2/`, February 2012. 9

[21] J. Rushby. Software verification and system assurance, 2009. 1

[22] K. Schneider. *Verification of reactive systems: formal methods and algorithms.* Texts in theoretical computer science. Springer, 2004. 6

[23] S. Schwoon. *Model-Checking Pushdown Systems.* Ph.D. Thesis, Technische Universität München, June 2002. 2

[24] Chief Scientist. Report on technology horizons: A vision for air force science & technology during 2010-2030. Technical report, United States Air Force, 2010. 1

[25] B. Seljimi and I. Parissis. Automatic generation of test data generators for synchronous programs: Lutess v2. In *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, DOSTA '07, pages 8–12, New York, NY, USA, 2007. ACM. 4

[26] SCADE Suite. Esterel technologies. `http://www.esterel-technologies.com/index.html`, February 2012. 6

[27] G. Tassey. Methods for assessing the economic impacts of government r&d. Technical report, National Institute of Standards and Technology, 2003. 1

[28] The Polychrony Toolset. Espresso team, irisa. `http://www.irisa.fr/espresso/Polychrony/index.php`, 2012. 6, 9, 50