Debugging PVS Specifications of Control Logics via Event-driven Simulation

Cinzia Bernardeschi*, Luca Cassano*, Andrea Domenici* and Paolo Masci*†
*Department of Information Engineering, University of Pisa, Italy
†Information Science and Technologies Institute, National Research Council, Pisa, Italy
Email: {Cinzia.Bernardeschi,Luca.Cassano,Andrea.Domenici,Paolo.Masci}@ing.unipi.it

***Abstract*—In this paper, we present a framework aimed at simulating control logics specified in the higher-order logic of the *Prototype Verification System*. The framework offers a library of predefined modules, a method for the composition of more complex modules, and an event-driven simulation engine. A developer simulates the specified system by providing its input waveforms as functions from time to logic levels. Once the simulation experiments give sufficient confidence in the correctness of the specification, the model can serve as a basis for the formal verification of desired properties of interest. A simple case study from a nuclear power plant application is shown. This paper is a contribution to research aimed at improving the development process of safety-critical systems by integrating simulation and formal specification methods.**

***Index Terms*—PVS; simulation; formal specification; validation**

I. Introduction

Control systems are an important field of application for formal methods and rigorous engineering practices, since they combine real-time requirements and non-trivial control tasks whose failure may compromise safety. Subtle design faults, which are often difficult to avoid and tolerate, and the possibility of failures caused by the occurrence of non-obvious combinations of events, make such systems hard to certify with respect to safety requirements.

In this paper, we present a methodology aimed at simulating control logics specified in the higher-order logic of the *Prototype Verification System (PVS)* [1]. We have developed a library of (purely logic) specifications for typical control logic components, a methodology to combine them into more complex systems, and a simulation engine capable of animating the formal specifications with the PVS ground evaluator.

Section II exposes the motivations for this work. We introduce the PVS system in Section III, then we describe the theories for the logical specification of control components (Section IV) and the theory defining the simulator (Section V). In Section VI we describe a simple case study from the field of control logics for nuclear power plants (NPPs), and finally the conclusion and related work are found in Section VII.

II. Motivation

The use of formal methods is increasingly being required by international standards for the development of safety critical digital control systems (e.g., [2], [3]), but, in industrial practice, verification and validation of such systems relies heavily on simulation and testing. A rigorous development process would benefit from the combined application of formal verification, simulation, and testing. In particular, simulation would be a means to validate specifications against requirements. However, verification tools (such as theorem provers and model checkers) and simulation tools use different languages, and few designers are versed in the use of both kinds of tools.

This work is a first step in a research activity whose expected outcome is a toolset that translates specifications from an application-oriented language into a high-order logic theory that guides the execution of the simulator described in this paper. When the simulation results make developers confident that the specifications are correct, a more detailed and formal analysis may be done by theorem proving. The theorem proving approach was chosen as it may be expected to avoid the problem of state space explosion that model checking tools face in the analysis of complex real-time systems.

III. PVS and PVSio

The PVS [1] specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, `integer`, `real`, among others, and the function type constructor (e.g., type `[A -> B]` is the set of functions from set `A` to set `B`). Predicates are functions with range type `bool`. The type system of PVS also includes record types, dependent types, and abstract data types.

PVS specifications are packaged as *theories* that can be parametric in types and constants. A collection of built-in (*prelude*) theories and loadable libraries provide standard specifications and proved facts for a large number of theories. A theory can use the definitions and theorems of another theory by *importing* it.

PVS has an automated theorem prover. A less frequently used component is its *ground evaluator* [4], used to animate functional specifications by translating executable PVS constructs into efficient *Lisp* code. The *PVSio* package [5] extends the ground evaluator with a library of imperative programming language features such as side effects, unbounded loops, and input/output operations. Thus, PVS specifications can be conveniently animated within the *read-eval-print* loop of the ground evaluator that reads PVS expressions from the user interface and returns the result of their evaluation.

IV. Modeling Control Logics

In this section we describe the PVS theories developed to formally model control logics. We start with the PVS theories that model time, logic levels, signals, and basic operations on signals. Then, we introduce samples of the library for the basic digital modules of a system, such as logic gates and timers. Finally, we show how to build complex components out of basic elements. The developed theories are executable: definitions always use interpreted types and quantification is always performed over bounded types. In the following sections, only the `time_th` theory will be shown in a syntactically complete form; to save space, only fragments of PVS code will be shown in the rest of the paper for the other theories.

A. Time and Logic Levels

Theory `time_th` (shown below) contains the type definition of time (modeled as ranging over the continuous domain of real numbers) and time interval.

```
time_th: THEORY
 BEGIN
  time: TYPE = real
  interval: TYPE = {t: time | t >= 0}
 END time_th
```

Besides the `zero` and `one` logical values, modeling hardware circuits requires additional levels for *unknown* values and *high impedance*. Unknown values are useful to model the logic level when the digital circuit is powered up, while high impedance represents open circuits (designed or faulty).

Theory `logic_levels_th` provides the definitions of the logic levels and of the basic logical operators over the four-valued logic (`lAND`, `lOR`, `lNOT`). In the following fragment we show the first definitions of the theory.

```
logic_level: TYPE  = below(4)
zero: logic_level = 0;
one: logic_level  = 1;
Z: logic_level    = 2;  %-- high impedance
U: logic_level    = 3;  %-- unknown value
lAND(v1, v2: logic_level): logic_level =
      IF one?(v1) AND one?(v2) THEN one
      ELSIF zero?(v1) OR zero?(v2) THEN zero
      ELSE U ENDIF
```

B. Signals

A signal describes the variation of a logic level over time, and we represent signals as functions from the domain of time to logic levels. Theory `signals_th` contains, besides the definition of `signal`, the symbolic constant for time resolution, `tres`, which models the minimum time between two observable variations of a signal, and the definition of a utility function to build periodic signals (`make_periodic`).

Basic signals provided in the theory are: `constval`, a constant logical level; `step`, a signal that goes from zero to one at time τ; `pulse`, a signal that is one only in the time interval $[\tau, \tau + d)$, where d is the interval size.

Some useful predicates on signals are defined, such as `rising_edge?`, used to detect if a signal `s` has a rising edge at time `tau`. Operations that apply logical connectives to the values of signals at each given time are defined (`sOR`, `sAND`, `sNOT`). Sample definitions of this theory follow.

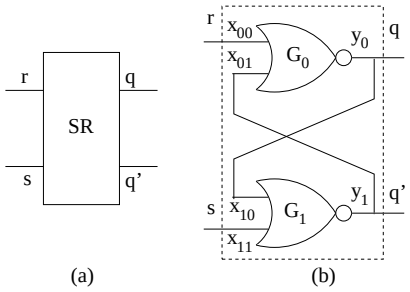

Fig. 1. An SR flip-flop.

```
IMPORTING time_th, logic_levels_th
signal: TYPE = [time -> logic_level]
tres: {t: interval | t > 0}
make_periodic(s: signal, T: interval): signal =
    % definition not shown
constval(v: logic_level): signal =
    LAMBDA (t: time): v
step(tau: time): signal =
    LAMBDA (t: time):
       IF t >= tau THEN one ELSE zero ENDIF
pulse(tau: time, d: posreal): signal =
    % definition not shown
rising_edge?(i: signal, tau: time): bool =
    zero?(i(tau - tres)) AND one?(i(tau + tres))
      AND one?(i(tau))
sAND(s1, s2: signal): signal =
    LAMBDA (t: time): lAND(s1(t), s2(t))
```

C. Digital Modules

In our framework, a control logic is a *composite digital module*, obtained by connecting *basic digital modules*.

Digital modules are characterized by a set of *ports*, a *state*, that is the collection of all signals present at its ports, and a *transition function* that specifies how the state changes according to a module's functionality. The behavior of each module in the framework is defined by its transition function.

Ports are abstractions of the terminals of physical devices. Each port is identified by its *category* (one of *input*, *output*, *internal*) and its *port number* within the category. Basic modules have only input and output ports, whereas composite modules also have internal ports. In a composite module, the input and output ports are its externally visible terminals, and its internal ports are the ports of the (basic) component modules that are not externally visible.

For example, a NOR gate is modeled as a module with two input ports, one output port, and no internal ports. Another example is an SR flip-flop, which can be modeled either as a basic module (Figure 1(a)) with two input ports, two output ports and no internal ports, or as a composite module built from two NOR gates. In the latter case, the resulting system is shown in Figure 1(b), where ports x_{00} of gate G_0 and x_{11} of gate G_1 are input ports, ports y_0 of G_0 and y_1 of G_1 are output ports, and ports x_{01} and x_{10} are internal ports.

Theory `digital_modules_th` contains type definitions for the state of a digital module (`state`) and for transition functions (`digital_module`). Type `state` the value of

the signals present at any time is a record that maintains the lists of signals applied at any time on its ports. It has one list of signals for each of the three port categories, and a port of the system is identified by its position in the list of the corresponding category. In the rest of this paper the term *signal* will sometimes be used instead of *port*, so that "signal x" means "the signal present at port x". The transition function type `digital_module` is time-dependent and has the signature $[time \to [state \to state]]$.

Note that the state of a module is defined as the set of `signals` applied to, or generated by, the module at a given time, and not as the set of their instantaneous values.

The theory includes also a number of auxiliary functions to build lists of ports (i.e., of signals) and to select a specific port of a module, such as `ports(`n`)`, `ports(`s, n`)`, etc. The first definitions of the theory follow.

```
IMPORTING signals_th
ports: TYPE = list[signal]
state: TYPE = [# input: ports, output: ports,
                 internal: ports #]
digital_module: TYPE = [time -> [state -> state]
%-- port constructors
ports(n: nat): RECURSIVE
               {p: ports | length(p) = n} =
  IF n = 0 THEN null
  ELSE cons(constval(U), ports(n - 1)) ENDIF
   MEASURE n
ports(s: signal, n: nat): RECURSIVE
               {p: ports | length(p) = n} =
    % definition not shown
%-- port selectors
port(p: ports, i: below(length(p))): signal =
   nth(p,i)
```

Types `state` and `digital_module` are very general, and they are refined by subtyping in the theories for basic digital modules and composite digital modules.

D. Basic Digital Modules

Basic digital modules are elements without a visible internal structure, defined only by their input and output ports and by their transition function. The state of a basic module has an empty list of internal signals, and the lists of input and output signals have a predefined length.

The theory is parametric with respect to a parameter `delay`, representing the time needed by the component to change its outputs when its inputs change.

In addition to the parameterized definitions for the state and transition function types, the theory contains a state constructor (`new_state`). Part of the theory is shown below.

```
basic_digital_modules_th[delay: nonneg_real]: THEORY
 BEGIN IMPORTING digital_modules_th
  state(nIN, nOUT: nat): TYPE =
           {s: state | length(input(s)) = nIN AND
                       length(output(s)) = nOUT AND
                       length(internal(s)) = 0 }
  basic_digital_module(nIN, nOUT: nat): TYPE =
    [time -> [state(nIN, nOUT) -> state(nIN, nOUT)]]
```

This theory is imported by other theories that define various classes of basic blocks, such as logic gates, timers, and flip-flops, presented in the following.

1) Logic gates: The `logic_gates_th` theory defines the transition functions of the basic combinatorial gates. The theory is parameterized by the propagation delay of the gates.

As the state is defined by the *signals* at the ports (and not the instantaneous values), the new state will normally be equal to the previous one, unless the environment applies different signals to the inputs (e.g., a pulse replaces a constant level). The definition for the NOR gate is shown below.

```
logic_gates_th[delay: nonneg_real]: THEORY
BEGIN IMPORTING basic_digital_modules_th
gateNOR: basic_digital_module(2, 1) =
  LAMBDA (t: time): LAMBDA (s: state(2, 1)):
    s WITH [output := ports(time_shift(
            sNOR(port0(input(s)), port1(input(s))),
            delay))]
```

2) Timers: The `timers_th` theory defines devices that generate a single pulse when they receive a rising edge on their input port. The pulse duration is a parameter of the device. Their response to the input depends on previous values of the output and possibly of the input(s).

```
timers_th[delay: nonneg_real]: THEORY
BEGIN IMPORTING basic_digital_modules_th
  timerM(d: posreal): basic_digital_module(1, 1) =
    LAMBDA (t: time): LAMBDA (s: state(1, 1)):
      IF rising_edge?(port0(input(s)), t) AND
         zero?(port0(output(s)), t)
      THEN s WITH [output := ports(pulse(t+delay, d))]
      ELSE s ENDIF
```

The theory defines also resettable timers (not shown), whose output drops to zero on receiving a rising edge at the reset port.

3) Flip-flops: The `flipflop_th` theory defines 1-bit registers, such as the SR flip-flop (Figure 1(a)). Ports s and r are the set and reset terminals, the stored bit is on the output marked q, and q' is its complement. Ports q and q' hold their previous value when s and r are both zero. If s becomes one while r is zero, then q is one, and stays at one even after s returns zero. Similarly, if r becomes one while s is zero, then q is zero, and stays at zero even after r returns zero.

```
flipflop_th[delay: nonneg_real]: THEORY
BEGIN IMPORTING basic_digital_modules_th
flipflopSR: basic_digital_module(2, 2) =
 LAMBDA (t: time): LAMBDA (st: state(2, 2)):
  LET r = port0(input(st)), s = port1(input(st)),
    q = port0(output(st)), q_prime = port1(output(st))
  IN IF zero?(s, t) AND zero?(r, t) THEN st
     ELSIF one?(s, t) AND zero?(r, t)
     THEN IF zero?(q, t) AND one?(q_prime, t)
          THEN st WITH [output := ports
                 (step(t+delay), sNOT(step(t+delay)))]
          ELSE st ENDIF
     ELSIF zero?(s, t) AND one?(r, t)
     THEN IF one?(q, t) AND zero?(q_prime, t)
          THEN st WITH [output := ports
                 (sNOT(step(t+delay)), step(t+delay))]
           ELSE st ENDIF
     ELSE st WITH [output := ports(2)] ENDIF
```

E. Composite Digital Modules

Basic digital modules can be connected together to create *composite digital modules*. The corresponding theory contains only the high-level definition for the state and the transition function, and for a state constructor (not shown).

```
BEGIN IMPORTING digital_modules_th
state(nIN, nOUT, nINT: nat): TYPE =
  {s: state | length(input(s)) = nIN AND
              length(output(s)) = nOUT AND
              length(internal(s)) = nINT}
composite_digital_module(nIN, nOUT, nINT: nat):
   TYPE = [time -> [state(nIN, nOUT, nINT)
                    -> state(nIN, nOUT, nINT)]]
```

1) Building Composite Digital Modules: A composite module is modeled by the composition of the transition functions of its components, whose form depends on the interconnections of the components.

In order to build the composite module, one must first define the *system* state, i.e., the union of its input, output, and internal ports. Then the subsets of the composite system state relative to the components (*component substates*) must be identified. Then the transition function is defined along the following lines: (i) Each port of the composite module is assigned a unique name by equating the port to a variable of type `signal` in a `LET` expression (e.g., `r = port0(input(st))` gives the name `r` to the first input port of state `st`); (ii) for each basic component, we define its current substate by selecting its input and output signals from the current system state; (iii) for each basic component, we define its next substate as a variable of type `state`, and we equate it to the basic component's transition function applied to the current substate defined in the previous step; (iv) the output signals of the new system state are the union of the output signals of the new substates of the basic components connected to the system output; (v) the internal signals of the next system state are the union of the internal signals of the new substates of the basic components.

As an example, we show the composite module of the SR flip-flop built from a pair of cross-coupled NOR logic gates. With reference to Figure 1(b), in this example port `x01` is renamed as `r1`, and `x10` as `s1`.

```
flipflopSR: composite_digital_module(2, 2, 2) =
LAMBDA (t: time): LAMBDA (st: state(2, 2, 2)):
 LET r = port0(input(st)), s = port1(input(st)),
  q = port0(output(st)), q_prime = port1(output(st)),
  r1 = port0(internal(st)), s1 = port1(internal(st)),
  nor0 = gateNOR[tres](t)(new_state(2, 1)
       WITH [input := ports(r, r1),
             output := ports(q)]),
  nor1 = gateNOR[tres](t)(new_state(2, 1)
       WITH [input := ports(s, s1),
             output := ports(q_prime)])
 IN st WITH [output := ports(port0(output(nor0)),
                         port0(output(nor1))),
             internal := ports(port0(output(nor1)),
                           port0(output(nor0)))]
```

In the system transition function `flipflopSR`, we let signal `r` be the signal on the first input port (`port0`) of the current system state `st`, and similarly for `s`, `q`, `q_prime`, `s1`, and `r1`. Then, substate `nor0` of gate `G0` is the result of transition function `gateNOR`. The argument of this function is a state with input signals `r` and `r1`, and output signal `q`. A similar description applies to `nor1`.

V. The Event-driven Simulator

This section describes an event-driven simulator of digital modules. First, we introduce *events*, i.e., instants when a signal may change its value. Second, we enrich the specification of the system with events. Third, we present the event-driven simulation engine, which uses the enriched specification to evaluate the system only at specific instants, instead of at periodic steps as in time-driven approaches [6].

A. Events

Theory `events_th` defines the type `event` as a record with fields `t`, the instant of a single event or of the first of a series of periodic events, and `T`, the period of the series (single events have `T=0`). The theory includes the ordering relation between events and operations on list of events. Some definitions are shown below.

```
BEGIN IMPORTING time_th
 event: TYPE = [# t: time, T: interval #];
 <(e1, e2: event): bool =
     (t(e1) < t(e2)) OR
       (t(e1) = t(e2) AND T(e1) < T(e2));
```

B. Annotated Signals

In theory `annotated_signals_th` we annotate the formal specification of signals with the list of events associated with each signal. We redefine the type `signal` as a record with the fields `val`, the functional specification of the signal, and `evts`, the set of instants when the waveform changes value. For example, the set of events associated with a constant level generator is empty, while the set of events associated with a pulse generator at time τ and duration d contains events τ and $\tau + d$, both with period $T = 0$.

The basic operators on signals are re-defined to calculate the events of the resulting signal, whose events are the union of events of the operator parameters. Some events in the resulting signal may not affect the signal value. For example, if initially one of the `sOR` inputs is a constant `one`, no set of events on the other input changes the output. Such redundant events, however, do not affect the simulation.

The following fragment shows the definition of `sNOR`.

```
BEGIN IMPORTING events_th, logic_levels_th
  sNOR(s1a, s2a: signal): signal =
    LET s1 = val(s1a), s2 = val(s2a),
      f = LAMBDA (t: time):
      IF one?(s1(t)) OR one?(s2(t)) THEN zero
      ELSIF zero?(s1(t)) AND zero?(s2(t)) THEN one
      ELSE U ENDIF,
      e = evts(s1a) + evts(s2a)
    IN (# val := f, evts := e #)
```

Annotated signals carry all the information needed by the simulator to handle events, so the specification of the digital modules is unchanged.

C. Simulator

The simulator maintains a list of events (*worklist*), initialized with the starting time of the simulation. The events are listed in ascending order without duplicates. At each simulation step, the simulator extracts the first event (*current*

event) from the worklist, and then it computes the next state by applying the system transition function at the time specified by the event. Then, the new events associated with the signals in the generated state are inserted in the worklist.

1) Worklist: Theory `worklist_th` defines the type `worklist` as a list of events, provides the function `get_first` that, given a current time, returns the first event associated with a set of signals and greater than the current time, and the function `update_wl` that updates the worklist. Function `update_wl` finds the new events in the next state and inserts them in the worklist. Note that, since the model of the system may contain ideal modules that update instantaneously their output ports, function `update_wl` must not remove the current event from the worklist as long as the generated state is not stable. These simple worklist manipulators are not shown.

2) Simulation Engine: The simulation engine applies the system transition function and returns the state of the system after a certain number of steps. It uses a customizable `dump` function to output a simulation trace.

```
simulate_system(n_steps: nat)
     (f: [time -> [state -> state]])
     (wl: worklist)(outf: OStream, pn: port_names):
    RECURSIVE [state -> state] =
 LAMBDA (s: state):
  IF n_steps > 0 AND length(wl) > 0
  THEN LET current_t = t(get_first(wl)),
     s_prime = update_state(s)(current_t, f),
     wl_prime = update_wl(wl)(current_t, s, s_prime),
     dbg = dump(outf, pn, s, s_prime,
                 wl, wl_prime, current_t)
    IN simulate_system(n_steps - 1)(f)(wl_prime)
                    (outf, pn)(s_prime)
  ELSE s ENDIF
```

The input parameters are the maximum number of steps, the system transition function, the worklist, the output stream for the trace, and the names of the signals. The function is called with an initial worklist containing all events of the initial state and an event for the initial time.

At each step, the function (i) gets the simulation time from the first event in the worklist, (ii) generates the next system state, (iii) updates the worklist, and (iv) outputs the system state. The simulation terminates when either the new worklist is empty, or the maximum number of steps is reached.

The following excerpt shows how the digital module `flipflopSR` is simulated. In function `sim_flipflopSR`, the initial state is constructed from the signals at the ports, the worklist is initialized, and `simulate_system` is invoked with the transition function as an argument. The *reset* port is initially fed with a constant zero signal, the *set* port with a pulse of $4s$ at time 0.3, and q (q') holds a constant zero (one).

```
sim_flipflopSR(N_STEPS: nat): bool =
 LET r = constval(zero), s = pulse(0.3, 4),
     q = constval(zero), q_prime = constval(one),
     r1 = q_prime, s1 = q,
     initial_st = new_state(2, 2, 2)
       WITH [input := ports(r, s),
             output := ports(q, q_prime),
             internal := ports(r1, s1)],
     initial_wl = worklist(initial_st, 0),
     final_s  = simulate_system(N_STEPS)(flipflopSR)
                               (initial_wl)(outf, pn)
                               (initial_st)
 IN TRUE
```

The simulation trace can be a list of event times, signal values and worklist contents at each step, or a *Value Change Dump* [7] output, readable by a visualization tool (e.g., *GTKWave* [8]).

3) Automated Execution of Test-Cases: The universal and existential quantifiers of PVS can be used to automatically set up different simulation studies, e.g., to analyze the response of the system to different input waveforms. This allows, for instance, instrumenting the framework for the execution of simulations in order to discover interesting test cases.

In the following example, the `test_flipflopSR` function uses the `FORALL` quantifier to generate all possible combinations of logical levels. Each combination defines an initial state for an SR flip-flop, and each state is used to compute a next state.

```
test_flipflop_th: THEORY BEGIN %--imports omitted
  % ...
  discrete_time: TYPE = below(2)
  test_flipflopSR: bool =
   FORALL (t_set, t_reset: discrete_time):
    FORALL (v1, v2: logic_level): v1 /= v2 =>
     (LET initial_st = new_state(2, 2, 2)
            WITH [input := ports(pulse(t_reset, 1),
                                 pulse(t_set, 1)),
                  output := ports(constval(v1),
                                  constval(v2)),
                  internal := ports(constval(v2),
                                  constval(v1))],
          initial_wl = worklist(initial_st, 0),
          final_s = simulate_system(5)(flipflopSR)
                    (initial_wl)(outf,pn)(initial_st)
      IN TRUE)
   %...
 END test_flipflop_th
```

VI. Case Study: A Stepwise Shutdown Logic

As an illustration of the practical applicability of the framework presented in this paper, we report on a simple case study from the field of Instrumentation and Control for NPPs. A high-level description of a control logic, expressed as a Function Block Diagram [9], has been manually translated into a PVS specification using the presented framework, and the specification has been animated to simulate the control logic. Simulated test cases have been automatically generated, allowing a possible malfunction to be detected at this early stage of development.

A. Description of a Stepwise Shutdown Logic

A *stepwise shutdown* process keeps process variables (such as, e.g., temperature or neutron flux) within prescribed thresholds by applying corrective actions (e.g., inserting control rods) not immediately to their full extent, but gradually, in a series of discrete steps separated by settling periods.

A Stepwise Shutdown Logic (SSL) was analyzed in [10] with a model checking approach. The framework proposed in this paper is used to analyze the same system.

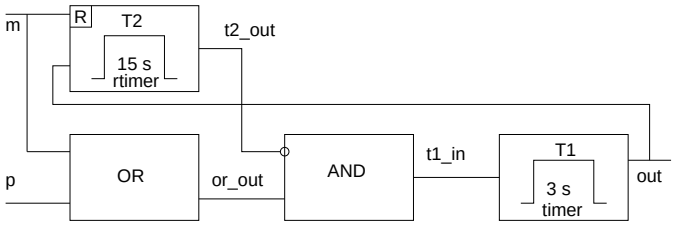

Fig. 2. A close-up view of the stepwise shutdown logic.

The requirements of the SSL, as described in [10], can be informally stated as follows: if an *alarm* signal (e.g., overpressure in a pipe) is asserted, the system must assert a control signal to drive a corrective action for 3 seconds (*active period*), then the control signal is reset for twelve seconds (*wait period*) and the cycle is repeated until either the alarm signal is reset or a complete shutdown is reached. An operator, however, by activating a *manual trip* signal, may force the wait periods to be skipped in order to accelerate the process.

Figure 2 shows the main part of one of the SSL implementations analyzed in [10], where `m` is the manual trip, `p` is an alarm signal, and `out` is the control signal. When all signals are low, the output `t2_out` of timer T2 is low, and the AND gate is enabled. When `p` is asserted, its rising edge passes through the AND gate to the input of the T1 timer that sends a $3s$ pulse to the output. The output is fed back to the input of T2, a resettable timer with a pulse duration of $15s$. The output pulse of T2 disables the AND gate that in turn resets the input of T1. Since T1 is not resettable, its output pulse lasts for three seconds, then returns to low for the remaining $12s$ of the T2 pulse. After this wait period, the output of T2 goes low, the AND gate is enabled, and T1 starts a new pulse if an input signal is still asserted.

If `p` is high, and `m` is asserted during a wait period, T2 is reset and its output enables the AND gate, allowing the trip signal to reach T1 and restart it at the end of the $3s$ pulse.

The SSL is modeled by the `systemC` transition function (see Figure 3), according to the guidelines in Section IV.

B. Simulation of the Stepwise Shutdown Logic

In this section we show some simulated situations.

Simulation 1 Signal `p` is a step function with the rising edge at $t = 0.3s$, and signal `m` is a constant zero (no manual intervention). The control logic produces a series of pulses that drive the plant towards a shutdown, as expected (Fig. 4). In the following, we show the PVS code for this simulation.

```
sim_system1: bool =
 LET initial_st =
  new_state(nIN, nOUT, nINT)
   WITH [input := ports(constval(zero), step(0.3)),
         output := ports(constval(zero)),
         internal := ports(constval(zero), nINT)],
  initial_wl = worklist(initial_st, 0),
  final_s = simulate_system(NSTEPS)(systemC)
            (initial_wl)(outf, pn)(initial_st)
  IN TRUE
```

Fig. 4. Output of sim_system1, displayed with GTKWave.

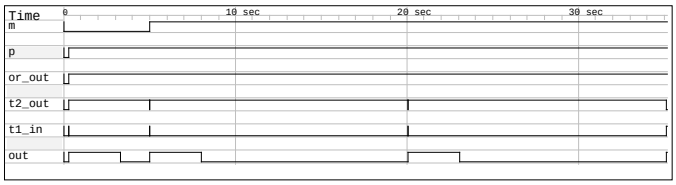

Fig. 5. Output of sim_system2, displayed with GTKWave.

Simulation 2 Signal `p` is a step function with the rising edge at $t = 0.3s$ and signal `m` is a step function with the rising edge at $t = 5s$. This means that the trip switch is pushed during the first wait period. As expected, that wait period is interrupted, a new $3s$ output pulse is generated, and the subsequent pulses are generated with the normal $15s$ cycle, since the trip switch has not been released and the resettable timer responds only to a rising edge (Fig. 5).

```
sim_system2: bool =
 LET initial_st =
  new_state(nIN, nOUT, nINT)
   WITH [input := ports(step(5), step(0.3)),
         output := ports(constval(zero)),
         internal := ports(constval(zero), nINT)],
  initial_wl = worklist(initial_st, 0),
  final_s = simulate_system(NSTEPS)(systemC)
            (initial_wl)(outf, pn)(initial_st)
  IN TRUE
```

Simulation 3 In this instance, signal `p` is a step function with the rising edge at $t = 1s$ and signal `m` is a pulse of duration $1s$ starting at $t = 2s$ followed by another pulse of duration $3s$ at $t = 10s$. In this case, the manual intervention occurs during the active period of the first output pulse. Contrary to expectation, after the end of this output pulse, the output is stuck at zero and no further corrective action takes place, even if the alarm (high pressure) persists and the manual trip switch is pressed again. A fundamental safety requirement is thus violated (Fig. 6).

```
sim_system3: bool =
 LET initial_st =
  new_state(nIN, nOUT, nINT)
   WITH [input := ports(sOR(pulse(2,1), pulse(10,3)),
                        step(1)),
         output := ports(constval(zero)),
         internal := ports(constval(zero), nINT)],
  initial_wl = worklist(initial_st, 0),
  final_s = simulate_system(NSTEPS)(systemC)
            (initial_wl)(outf, pn)(initial_st)
  IN TRUE
```

Test-Cases Interesting simulation examples, such as `sim_system3`, can be discovered by instrumenting the framework for the execution of test cases.

```
systemC: composite_digital_module(nIN, nOUT, nINT) =
 LAMBDA (t: time): LAMBDA (st: state(nIN, nOUT, nINT)):
  LET m = port0(input(st)), p = port1(input(st)), out = port0(output(st)),
      t2_in = port0(internal(st)), t2_out = port1(internal(st)),
      %- similar definitions for or_in, and_en, and_out
      rtimer = rtimerM[T1](D2)(t)(new_state(2,1) WITH [input:=ports(t2_in,m), output:=ports(t2_out)]),
      or2 = gateOR[T0](t)(new_state(2,1) WITH [input:=ports(or_in,p), output:=ports(or_out)]),
      inh_and = gateANDH[T0](t)(new_state(2,1) WITH [input:=ports(and_en,and_in), output:=ports(and_out)]),
      timer = timerM[T2](D1)(t)(new_state(2,1) WITH [input:=ports(t1_in), output:=ports(out)])
   IN st WITH [input := ports(m, p), output := ports(port0(output(timer))),
               internal := ports(port0(output(timer)), port0(output(rtimer)), m, port0(output(or2)),
                port0(output(rtimer)), port0(output(or2)), port0(output(inh_and)), port0(output(inh_and)))]
```

Fig. 3. PVS model of the Stepwise Shutdown Logic.

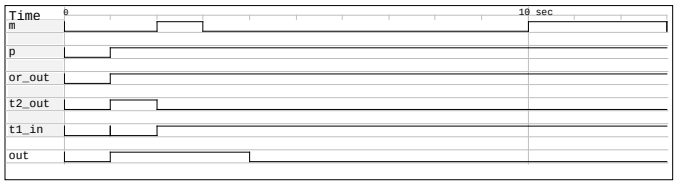

Fig. 6. Output of sim_system3, displayed with GTKWave.

In the following example, function `test_system` uses the `FORALL` quantifier to automatically generate the initial state for the different test cases. The initial states differ by the starting time of the pulse applied to the manual trip port. The ground evaluator implicitly transforms the universally quantified formula on `t0` into a loop that, at each iteration, generates a new initial state with a pulse starting at `t0 = 0,1,..,N-1` on the manual trip port, and applies the simulator for `NSTEPS` steps.

```
sim_system_test(N: nat): bool =
 FORALL(t0: below(N)):
  LET initial_state =
      new_state(nIN, nOUT, nINT)
       WITH [input := ports(pulse(t0,1), step(1)),
             output := ports(constval(zero)),
             internal := ports(constval(zero), nINT)],
     initial_wl = worklist(initial_st, 0),
     final_s = simulate_system(NSTEPS)(systemC)
               (initial_wl)(outf, pn)(initial_st)
    IN TRUE
```

VII. Conclusion and Related Work

PVS has been used in various works to describe hardware systems, e.g., in [11], [12], [13]. With our approach, the formal specifications are executable and they can be simulated with the ground evaluator of PVS. This way, once the simulation experiments give developers sufficient confidence in the correctness of the specification, the same PVS models can serve as the basis for the formal verification of properties in the theorem prover of PVS. It is known that a large share of defects in computing systems stem from errors in the formulation of specifications [14].

In the present work, a library of (purely logic) specifications for typical control logic components is presented, and an approach to define an event-driven simulator capable of executing the logic specifications is shown. The library includes theories to model logic signals over time, where time is a variable in the domain of real numbers. The simulator is based on the paradigm of event-driven-simulation, and its core component is defined as a function in the higher-order logic language of the PVS system. proving environment. The approach has been applied to a simple case study in the field of NPPs. The same case study had been previously studied by other researchers with a model checking approach [10].

This work is part of our current research activity aiming at developing a simulation and analysis framework for control logics that enables developers to rely both on simulation and theorem proving to assess the correctness of specifications and designs.

References

[1] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. on Software Engineering*, vol. 21, no. 2, pp. 107–125, 1995.

[2] "Railway applications – Software for railway control and protection systems," CENELEC, European Committee for Electrotechnical Standardization, Tech. Rep. EN 50128:2001 E, 2001, european standard.

[3] "Software for Computer Based Systems Important to Safety in Nuclear Power plants," IAEA, International Atomic Energy Agency, Tech. Rep. NS-G-1.1, 2000.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "Evaluating, testing, and animating pvs specifications," Computer Science Laboratory, SRI International, Tech. Rep., 2001.

[5] C. Muñoz, "Rapid prototyping in PVS," National Institute of Aerospace, Hampton, VA, USA, Tech. Rep. NASA/CR-2003-212418, 2003.

[6] A. M. Law and D. Kelton, *Simulation Modeling and Analysis*. McGraw-Hill, 2000.

[7] "IEEE Standard Verilog Hardware Description Language," IEEE, Tech. Rep. IEEE Std 1076-2000, 2000.

[8] "GTKWave 3.3 Wave Analyzer User's Guide," BSI, Tech. Rep., 2009.

[9] International Electrotechnical Commission, *Programmable controllers - Part 3: Programming languages, Ed 2.0, International Standard IEC 61131-3, IEC*, 2003.

[10] K. Björkman, J. Frits, J. Valkonen, J. Lahtinen, K. Heljanko, I. Niemelä, and J. J. Hämäläinen, "Verification of Safety Logic Designs by Model Checking," in *Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT 2009)*, 2009.

[11] S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A tutorial on using PVS for hardware verification," in *Theorem Provers in Circuit Design (TPCD '94)*, ser. LNCS, R. Kumar and T. Kropf, Eds. Springer-Verlag, 1997, no. 901, pp. 258–279.

[12] M. Srivas, H. Rueß, and D. Cyrluk, "Hardware verification using PVS," in *Formal Hardware Verification: Methods and Systems in Comparison*, ser. LNCS, T. Kropf, Ed. Springer-Verlag, 1997, no. 1287, pp. 156–205.

[13] C. Berg, C. Jacobi, and D. Kröning, "Formal verification of a basic circuits library," in *Proc. of the IASTED International Conference on Applied Informatics, Innsbruck (AI 2001)*. ACTA Press, 2001.

[14] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *Proceedings of the IEEE International Symposium on Requirements Engineering*, 1993, pp. 126–133.