Modeling an Asynchronous Circuit Dedicated to the Protection Against Physical Attacks

Radu Mateescu, Wendelin Serwe, Aymane Bouzafour, Marc Renaudin

Modeling an Asynchronous Circuit Dedicated to the Protection Against Physical Attacks

Radu Mateescu Wendelin Serwe
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, F-38000 Grenoble France
First.Last@inria.fr
Aymane Bouzafour Marc Renaudin
Tiempo Secure – SAS, Montbonnot-Saint-Martin, France
First.Last@tiempo-secure.com

Asynchronous circuits have several advantages for security applications, in particular their good resistance to attacks. In this paper, we report on experiments with modeling, at various abstraction levels, a patented asynchronous circuit for detecting physical attacks, such as cutting wires or producing short-circuits.

1 Introduction

The number of connected devices that make up the IoT (*Internet of Things*) already exceeded 20 billion, and is constantly increasing. However, a study of Hewlett Packard[1] indicates that 90% of the objects collect and thus potentially expose data, that 80% of the objects do not use identification and source authentication mechanisms, and that 70% do not use a mechanism of encrypting the transmitted data. These vulnerabilities open the way to DDoS (*Distributed Denial-of-Service*) attacks, which exploit over $100,000$ infected devices (e.g., cameras, video recorders, etc.) to overload various services and websites with a deluge of data. Therefore, strengthening the security of connected devices is a critical issue.

The SECURIOT-2 project[2], led by Tiempo Secure, aims at developing a SMCU (*Secure Micro-Controller Unit*) that will bring to the IoT a level of security similar to banking transactions and transport (smart cards) and identification (electronic passports). Besides ensuring the necessary security services (key management, authentication, confidentiality and integrity of stored/exchanged data), the SMCU needs a power management scheme adequate with the low power consumption constraints of the IoT.

Given these constraints, a natural implementation of an SMCU is by means of asynchronous circuits, whose components are not governed by a clock signal, but operate independently, on an "on-demand" basis. Compared to classical synchronous circuits, this functioning has the potential for lower power dissipation, more harmonious electromagnetic emission, and better overall timing performance.

In this paper, we study the so-called *shield* [25, 26], a particular asynchronous circuit designed and patented by Tiempo Secure for the protection of another (asynchronous) circuit against certain physical attacks (e.g., cutting a wire, setting a wire to a constant voltage, and introducing a short-circuit between two wires). The behavior of asynchronous circuits can be suitably modeled in the action-based setting, using the composition operators of process calculi, as witnessed by CHP [21], Tangram [4], Balsa [10], which are all inspired by CSP [6], and have been successfully applied to design asynchronous circuits,

*Institute of Engineering Univ. Grenoble Alpes

[1] https://www8.hp.com/us/en/hp-news/press-release.html?id=1909050

[2] http://www.pole-scs.org/en/projects/securiot-2

A. Fehnker and H. Garavel (Eds.):
Models for Formal Analysis of Real Systems (MARS 2020)
EPTCS 316, 2020, pp. 200–239, doi:10.4204/EPTCS.316.8

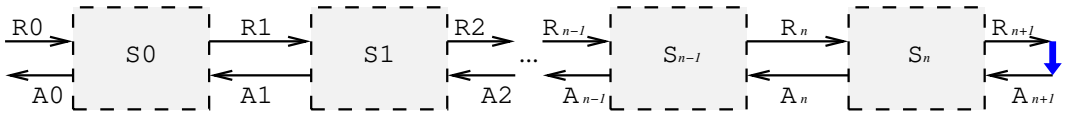


Figure 1: Shield: serial pipelined composition of sequencers

e.g., [2, 22, 24]. All these approaches are equipped with a compilation scheme [21], producing QDI (*Quasi-Delay-Insensitive*) circuits, the correct operation of which is independent of the delay in operators (or logical gates) and wires, except for certain wires that form *isochronic forks* [21]. An isochronic fork is a wire connecting an emitter to several receivers, which receive any signal with identical delays.

We consider the modeling and analysis of the shield at two abstraction levels. First, at circuit level, we take into account only the components of the circuit and their interconnection, without modeling the implementation details of these components. This level is appropriate for reasoning about the desired properties of the shield, namely the detection of physical attacks. The regular structure of the shield (serial pipeline) enables to apply inductive arguments to reduce all possible attack configurations to a finite set, which we analyzed exhaustively. Next, we undertake a gate level modeling, focusing on the implementation of a component in terms of logical gates. Here we explore a range of different modeling variants for gates, electric wires, and forks (isochronic or not), and analyze their respective impact on the faithfulness of the global circuit model, the size of the underlying state spaces, the expression of correctness properties, and the overall ease of verification. We also point out that certain modeling variants lead to deadlocks in the circuit.

In this study, we mainly use the modern LNT [13] language for concurrent systems, which offers a user-friendly syntax and a formal semantics inherited from process calculi, and has been shown to be close to industrial hardware description languages for asynchronous circuits [5]. For the modeling of attacks, we also use the synchronization vectors of EXP [20], which operate on networks of communicating automata. LNT and EXP are input languages of the CADP toolbox [12][3] for modeling and verification of asynchronous concurrent systems.

The paper is organized as follows. Section 2 describes the protection circuit and its behavior. Sections 3 and 4 are devoted to the analysis of the protection circuit at two abstraction levels (circuit-level and gate-level, respectively), in regard to its properties of detecting physical attacks. Both sections illustrate the approach on examples; the full models can be found in the appendices. Section 5 discusses related work and Section 6 gives concluding remarks and directions for future work.

2 Shield

To protect an integrated circuit against physical attacks, the patent [25, 26] suggests to add as a top layer a particular electric circuit, called *shield* in the sequel. Physical attacks to the circuit are generally meant to allow the attacker to probe for the sensitive information that is stored in the internal registers/wires of the circuit during its ordinary operation. These probing procedures will inevitably lead to a tampering with the shield as it constitutes the top layer of the accessible part of the circuit. Hence, proving that the shield is able to flag any tampering attempts amounts to proving that the circuit itself is able to detect a physical attack and to stop its normal operation and take an appropriate countermeasure (e.g., completely deactivate the circuit).

[3] http://cadp.inria.fr

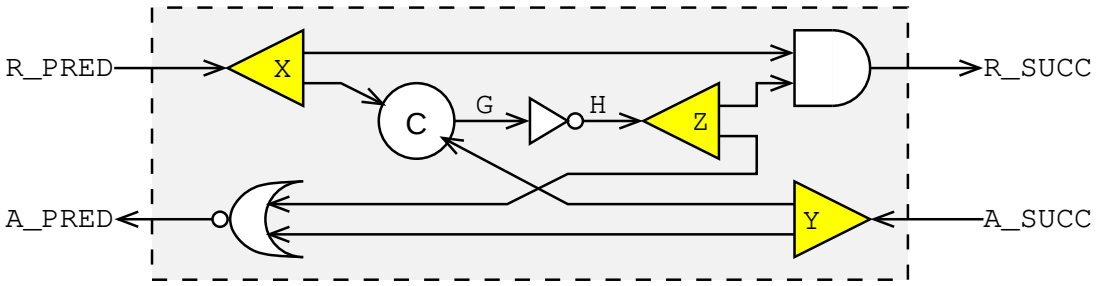
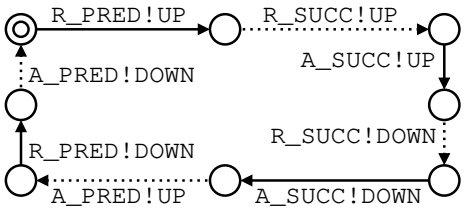


Figure 2: Gate-level design of a sequencer

Fig.	R_PRED	A_SUCC	G	H	R_SUCC	A_PRED
4	DOWN	DOWN	DOWN	UP	DOWN	DOWN
5	UP	DOWN	DOWN	UP	UP	DOWN
6	UP	UP	UP	DOWN	DOWN	DOWN
7	UP	DOWN	UP	DOWN	DOWN	UP
8	DOWN	DOWN	DOWN	UP	DOWN	DOWN

(a) Behavior of a sequencer (according to figures of [26])



(b) Protocol automaton

Figure 3: Behavior of a sequencer

A shield is a serial composition of $n+1$ *sequencers* (see Fig. 1 or [26, Fig. 3]), or even a parallel composition of several series of sequencers (see [26, Figs. 9–11]). Each sequencer transmits a first signal, called *request* to its successor; when the last sequencer outputs a request, a second signal, called *acknowledgment* is transmitted through the series of sequencers in the opposite direction. If a sequencer is designed in such a way that any physical modification on the connections for the transmission of the request (respectively, the acknowledgment) blocks the transmission of the acknowledgment (respectively, the request), a physical attack can be detected by the absence of an acknowledgment for a request sent into the shield.

Figure 2 shows the gate-level design of a circuit presented as a possible implementation of a sequencer in the patent.[4] The intended evolution over time of the circuit [26, Figs. 4–8] is summarized by the table in Fig. 3a. When considering only the externally visible wires, the behavior of a sequencer corresponds to the automaton shown in Fig. 3b, where input (respectively, output) transitions are depicted with plain (respectively, dotted) arrows. Initially, the output G of the Muller C element [23] and the two inputs R_PRED and A_SUCC are DOWN. When input R_PRED goes UP, output R_SUCC goes UP as well. As a reaction, input A_SUCC is expected to go UP, triggering output R_SUCC to go DOWN. When A_SUCC goes DOWN, output A_PRED goes UP. Finally, R_PRED goes DOWN, bringing the circuit back to its initial state. This cycle is related to the so-called four-phase handshake protocol, frequently used in the design of asynchronous circuits [21]. Note that there is no constraint on reaction delays of the circuit: a change of an input should be eventually followed by the corresponding output change. Similarly, the environment of the circuit should respect the protocol by reacting to a change of an output by changing an input as specified by the protocol.

[4] There are small differences with [26, Figs. 4–8]: (1) omission of the input RST to the Muller C element; (2) no distinction between links carrying 0 (dashed in [26]) and 1 (plain in [26]); (3) naming forks and highlighting them (in yellow); (4) explicit drawing of the fork Z after the inverter; and (5) labeling of the visible and internal wires.

3 Circuit-level Analysis of the Shield

3.1 Modeling a Serial Shield

Considering a sequencer as a black-box, its behavior can be defined in the LNT modeling language [13] by the following process:

```
process PROTOCOL [R_PRED, A_PRED, R_SUCC, A_SUCC: LINK] is
   loop
      R_PRED (UP);      R_SUCC (UP);    A_SUCC (UP);      R_SUCC (DOWN);
      A_SUCC (DOWN);    A_PRED (UP);    R_PRED (DOWN);    A_PRED (DOWN)
   end loop
end process
```

This process interacts with its environment on four LNT gates (`R_PRED`, `A_PRED`, `R_SUCC`, and `A_SUCC`), each of which is a channel carrying a voltage (`DOWN` or `UP`). The LTS (*Labeled Transition System*) generated for process `PROTOCOL` is exactly the automaton shown in Figure 3b.

Composition of sequencers in the manner of a pipeline as shown in Fig. 1 yields a model of a (serial) shield. To ease the iterative construction of such sequence, we choose the EXP language [20] to define a composition operator $pipe(C_1, C_2)$ to pipe two sequencers C_1 and C_2 stored in the files `"C1.bcg"` and `"C2.bcg"`:

```
hide R, A in
   par R, A in
      rename R_SUCC -> R, A_SUCC -> A in "C1.bcg" end rename
   || rename R_PRED -> R, A_PRED -> A in "C2.bcg" end rename
   end par
```

Supposing that C_1 and C_2 both have the set {`R_PRED`, `A_PRED`, `R_SUCC`, `A_SUCC`} of observable gates, *pipe* renames the right-hand (respectively, left-hand) side gates of C_1 (respectively, C_2) into a pair of new gates {`R`, `A`}, on which C_1 and C_2 are synchronized. Hiding `R` and `A` in the composition ensures then that the observable gates of $pipe(C_1, C_2)$ are once more {`R_PRED`, `A_PRED`, `R_SUCC`, `A_SUCC`}. Thus, *pipe* can be easily extended to series of sequencers, by defining $pipe^0(S) = S$ and $pipe^{n+1}(P) = pipe\big(P, pipe^n(P)\big)$.

Letting P stand for the LTS of process `PROTOCOL`, equivalence checking (e.g., using BISIMULATOR [3]) shows that $pipe(P, P) \equiv P$, where $\equiv$ denotes equivalence with respect to divergence-sensitive branching bisimulation. It follows by a straightforward induction that:

$$(\forall n \in \mathbb{N}) \qquad pipe^n(P) \equiv P \tag{1}$$

3.2 Modeling Attacks

Intuitively, a physical attack on the shield corresponds to a modified composition of some sequencers, which can be represented by a modification of the composition operator. The attacks can be grouped into two classes: those that impact the interface between two sequencers, and those that impact more than two sequencers. If an attack can be expressed using operations for which divergence-sensitive branching bisimulation is a congruence, then it is sufficient to analyze this attack on any pair (or triple) of sequencers in a pipeline, because any non-trivial sequence of correctly pipelined sequencers can be reduced to a single sequencer. In this section, we will present only one example of each class of attacks; a more complete treatment can be found in Appendix B.

Figure 4: Serial pipeline of three sequencers

3.2.1 Attacks impacting two sequencers: wire cuts and stuck at a constant

Forcing a wire `W` to a constant value `V` can be represented by allowing only `V` as value during each rendezvous on the LNT gate `W`. Using the constraint-oriented specification style favored by the multi-way rendezvous [6, 15, 27, 14], it is sufficient to add a parallel process, which continuously accepts a rendezvous on `W` with value `V`:

```
process STUCKAT [W: LINK] (V: VOLTAGE) is
   loop
      W (V)
   end loop
end process
```

There are two possibilities to modify the composition operator: either both sequencers are synchronized with process `STUCKAT`, or only the sequencer receiving on `W` (i.e., the right-hand side sequencer C_2 for gate `R`, and the left-hand sequencer C_1 for gate `A`). The latter is expressed by the following composition expression in EXP, where file `"STUCKAT_UP.bcg"` contains the LTS of process `STUCKAT [W] (UP)`:

```
hide R, A in
   par
      A ->     rename R_SUCC -> R, A_SUCC -> A in "C1.bcg" end rename
   || R, A -> rename R_PRED -> R, A_PRED -> A in "C2.bcg" end rename
   || R ->     rename W -> R in "STUCKAT_UP.bcg" end rename
   end par
```

Note that `"C1.bcg"` can always perform a rendezvous on `R`, without any constraint on the value.

A wire-cut can be handled similarly to a wire stuck at a constant: any rendezvous is blocked when synchronizing with the deadlocking process `stop` (rather than `STUCKAT`), and any rendezvous is enabled by simply desynchronizing the two sequencers.

3.2.2 Attacks impacting more than two sequencers: short-circuits

For a series of three sequencers as shown in Fig. 4, there are six possible short-circuits around the sequencer `S1`: `R1-R2`, `R1-A1`, `R2-A1`, `R2-A2`, `A1-A2`, and `A1-R2`.[5] The definition of a composition operator modeling a short-circuit requires the use of synchronization vectors, to properly handle any disagreement between the value exchanged during the rendezvous. The most generic way to cope with these situations is to consider the resulting value to be non-deterministic. Another, more constrained modeling option would be to enforce a majority-based policy (in a three-party rendezvous with two possible values, there is always a majority).

The following composition EXP expression models a short-circuit between `R1` and `A2`:

[5] The short-circuits `R1-A1` and `R2-A2` are similar in the sense that they could both be defined by the same attack for a series of only two sequencers.

```
hide R1, A2, R2, A2, R1A2 in
  label par using
    -- synchronization vectors for unmodified wires
    "R_PRED␣!DOWN" * _           * _               -> "R_PRED␣!DOWN",
    "R_PRED␣!UP"   * _           * _               -> "R_PRED␣!UP",
    "A_PRED␣!DOWN" * _           * _               -> "A_PRED␣!DOWN",
    "A_PRED␣!UP"   * _           * _               -> "A_PRED␣!UP",
    "A1␣!DOWN"     * "A1␣!DOWN" * _                -> "A1␣!DOWN",
    "A1␣!UP"       * "A1␣!UP"   * _                -> "A1␣!UP",
    _              * "R2␣!DOWN" * "R2␣!DOWN"      -> "R2␣!DOWN",
    _              * "R2␣!UP"   * "R2␣!UP"        -> "R2␣!UP",
    _              * _          * "R_SUCC␣!DOWN" -> "R_SUCC␣!DOWN",
    _              * _          * "R_SUCC␣!UP"   -> "R_SUCC␣!UP",
    _              * _          * "A_SUCC␣!DOWN" -> "A_SUCC␣!DOWN",
    _              * _          * "A_SUCC␣!UP"   -> "A_SUCC␣!UP",
    -- synchronization vectors for the short-circuit
    "R1␣!DOWN"     * "R1␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!DOWN",
    "R1␣!DOWN"     * "R1␣!DOWN" * "A2␣!UP"       -> "R1A2␣!DOWN",
    "R1␣!DOWN"     * "R1␣!DOWN" * "A2␣!UP"       -> "R1A2␣!UP",
    "R1␣!UP"       * "R1␣!UP"   * "A2␣!DOWN"     -> "R1A2␣!DOWN",
    "R1␣!UP"       * "R1␣!UP"   * "A2␣!DOWN"     -> "R1A2␣!UP",
    "R1␣!UP"       * "R1␣!UP"   * "A2␣!UP"       -> "R1A2␣!UP",
    "R1␣!DOWN"     * "A2␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!DOWN",
    "R1␣!DOWN"     * "A2␣!UP"   * "A2␣!UP"       -> "R1A2␣!DOWN",
    "R1␣!DOWN"     * "A2␣!UP"   * "A2␣!UP"       -> "R1A2␣!UP",
    "R1␣!UP"       * "A2␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!DOWN",
    "R1␣!UP"       * "A2␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!UP",
    "R1␣!UP"       * "A2␣!UP"   * "A2␣!UP"       -> "R1A2␣!UP"
  in
     rename R_SUCC -> R1, A_SUCC -> A1 in "C1.bcg" end rename
  || rename R_PRED -> R1, A_PRED -> A1, R_SUCC -> R2, A_SUCC -> A2 in
        "C2.bcg"
     end rename
  || rename R_PRED -> R2, A_PRED -> A2 in "C3.bcg" end rename
  end par
```

There are two kinds of rules for unmodified wires: there is no synchronization for those externally visible (`R_PRED`, `A_PRED`, `R_SUCC`, and `A_SUCC`), and a binary synchronization for the remaining ones (`A1` and `R2`). For the three-party rendezvous of the short-circuit (in the example between `R1` and `A2`), a new gate `R1A2` is introduced; the voltage on `R1A2` is non-deterministic if and only if there is disagreement on the voltage.

3.2.3 Circuit-Level Analysis Results

The SVL (*Script Verification Language*) [11] script given in Appendix C automates the circuit-level validation of the shield. To show that the shield detects an attack, we check whether the expected behavior (see Fig. 3b) is included in the model generated for the corresponding attack.

All attacks forcing a wire to a constant value are detected. A wire-cut might be left undetected only if the attack is modeled such that the gates on both ends of the wire are free to perform arbitrary rendezvous on the gate without ever synchronizing, which is not a realistic assumption. All short-circuits but `R1-A1` and `R2-A2` are detected. This can be explained by the fact that these short-circuits cut the pipe-line of sequencers, reducing them to a shorter shield, keeping the same functionality. However, because the

electric connections for R and A are located in different layers of the chip [25, 26], attacking the chip in this way is impossible (without damaging the entire chip).

Because attacks are modeled by operators congruent for branching bisimulation, these results extend to a series of N sequencers. Indeed, using equation (1) any non-trivial series of sequencers can be replaced by a single sequencer, so that any (single) attack can be reduced to one of the studied configurations with two or three sequencers.

4 Gate-level Analysis

For a more precise study of a sequencer, we refine the model of a sequencer by representing the internal operation of all the electric wires and gates according to Fig. 2: one binary wire, three forks, one inverter, one AND gate, one NOR gate, and one Muller C element. Intuitively, each of them can be considered as a process, which reacts on its input(s) by (possibly) producing some output. However, there are various possibilities with respect to the degree of synchronization and possible propagation delays. This section aims at exploring these possibilities, discussing the respective advantages in terms of ease of modeling, size of the corresponding state space, and practicality for verification.

A major choice when modeling a gate is whether the model represents only *transitions* (i.e., changing voltage, as in the protocol of Fig. 3b) or rather current *state* of a wire. Considering only transitions yields smaller models and corresponding state spaces, but might seem less intuitive because it requires stronger assumptions (e.g., synchronous communication).

4.1 Modeling Wires and Forks

A wire can be modeled in essentially two ways: as an LNT gate or as an instance of the LNT process

```
process WIRE [INPUT, OUTPUT: LINK] is
   var X : VOLTAGE in
      loop
         INPUT (?X);  OUTPUT (X)
      end loop
   end var
end process
```

Representing a wire by an LNT gate models the immediate transmission from the input to the output, whereas the LNT process models the possibility of a communication delay, because other actions of the circuit might be interleaved between the input and the output. Notice that (similar to [19]) process WIRE enforces alternation, i.e., it only accepts the next input after it has delivered the output.

A fork is a wire with more than one output, and can also be modeled as a gate (using a multi-way rendezvous [6, 15, 27, 14]), faithfully representing isochronic forks [18]. It is also possible to model a fork as a dedicated process. However, as there is more than one output, it is possible to specify their order of occurrence, from simultaneous (using a multi-way rendezvous for the outputs) to unspecified (using a parallel composition). The latter option is modeled by the LNT process

```
process FORK [INPUT, OUTPUT1, OUTPUT2: LINK] is
   var X : VOLTAGE in
      loop
         INPUT (?X);
         par
            OUTPUT1 (X)
```

```
         || OUTPUT2 (X)
         end par
      end loop
   end var
end process
```

To ensure a correct functioning of an asynchronous circuit, some forks must be assumed to be *isochronic*. An isochronic fork can be modeled using the process `WIRE`, using a multi-way rendezvous on gate `OUTPUT` (similar to [18]).

Notice that all these models of wires and forks are valid for both the transition and state oriented style, because they do not have a memory about the last transmitted value.

4.2 Modeling a Sequencer

Supposing that we have LNT processes modeling gates, the sequencer of Fig. 2 can be modeled by a spectrum of possibilities, considering the options of modeling wires and forks. A first model using multi-way rendezvous for wires and forks is:

```
process SEQUENCER_RV [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H: LINK]
                     (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED, A_SUCC, G -> MULLER [R_PRED, A_SUCC, G] (X1, X2, INIT_C)
   || R_PRED, H -> AND [R_PRED, H, R_SUCC] (X1, NOT (INIT_C))
   || G, H -> INV [G, H] (INIT_C)
   || A_SUCC, H -> NOR [A_SUCC, H, A_PRED] (X2, NOT (INIT_C))
   end par
end process
```

On the other extreme, the following LNT process represents wires and forks as explicit processes, and all its forks but `Z` are isochronic:

```
process SEQUENCER_IIP [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED2, A_SUCC2, G2, H1, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC2, G ->
         MULLER [R_PRED2, A_SUCC2, G] (X1, X2, INIT_C)
   || R_PRED2, H1 -> AND [R_PRED2, H1, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED2 -> WIRE [R_PRED, R_PRED2]
   || A_SUCC2 -> WIRE [A_SUCC, A_SUCC2]
   || H, H1, H2 -> FORK [H, H1, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process
```

Models for other combinations are given in Appendix A.5.

4.3 Modeling Gates

We illustrate the various ways to model gates on the binary `AND` gate; models of the other gates are given in Appendix A. Fig. 5 shows various (state-oriented) bodies that can replace "..." in the LNT process

```
loop
   select
      INPUT1 (?X1)
   [] INPUT2 (?X2)
   end select;
   OUTPUT (X1 AND X2)
end loop
```

(a) intuitive

```
loop
   select
      INPUT1 (?X1);
      select
         null
      [] INPUT2 (?X2)
      end select
   [] INPUT2 (?X2);
      select
         null
      [] INPUT1 (?X1)
      end select
   end select;
   OUTPUT (X1 AND X2)
end loop
```

(b) state-oriented

```
loop
   par
      select
         INPUT1 (?X1)
      [] null
      end select
   || select
         INPUT2 (?X2)
      [] null
      end select
   end par;
   OUTPUT (X1 AND X2)
end loop
```

(c) parallel

```
loop
   select
      INPUT1 (?X1)
   [] INPUT2 (?X2)
   [] OUTPUT (X1 AND X2)
   end select
end loop
```

(d) free

Figure 5: State-oriented models of a binary AND gate

```
process AND [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   ...
end process
```

The most intuitive model is state-oriented and shown in Fig. 5(a). It reacts to a rendezvous on one of its inputs with a rendezvous on its output. An issue with Fig. 5(a) is that it requires an output to occur between any two inputs, but it might be the case that the inputs arrive almost simultaneously. This issue is addressed by Fig. 5(b), which may accept one or two inputs in an arbitrary order before generating an output. Using a parallel composition operator, Fig. 5(b) can be simplified to Fig. 5(c). Fig. 5(c) has the inconvenient that it might generate outputs that are not triggered by an input. Permitting also that an input might not generate an output, one obtains Fig. 5(d), which is always ready to accept a new input or to (re)generate the current output.

A transition-oriented model of a gate must generate an output in reaction to a change in one of the inputs if and only if the output changes. This requires to remember the previous output, as in the process:

```
process AND [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   var RESULT, NEW_RESULT: VOLTAGE in
      RESULT := X1 AND X2;
      loop
         select
            INPUT1 (?X1)
         [] INPUT2 (?X2)
         end select;
         NEW_RESULT := X1 AND X2;
         if NEW_RESULT != RESULT then
            RESULT := NEW_RESULT;
            OUTPUT (RESULT)
         end if
      end loop
```

```
    end var
end process
```

4.4 Gate-level Analysis Results

We used the Grid'5000 platform to generate the state spaces for one and two sequencers, using various combinations of the modeling choices discussed previously. Table 1 summarizes the results. The first column indicates the model of gates: INTUITIVE, STATE, PARALLEL, and FREE refer respectively to the state-oriented style of Fig. 5(a), (b), (c), and (d), and TRANSITION refers to the transition-oriented style (see Sect. 4.3). The second column indicates the model of wires and forks: RV indicates that wires and forks are modeled by a rendezvous, and a triple $F_1F_2F_3$ (with $F_i \in \{\mathrm{I},\mathrm{P}\}$) indicates the isochrony of the three forks (X, Y, and Z in Fig. 2) in a sequencer (I stands for an isochronic fork, and P for a non-isochronic one); in all cases but for RV, wires are modeled as separate processes. The third and fourth columns indicate the size of one sequencer, minimized for divergence-sensitive branching bisimulation (after hiding all gates but R_PRED, A_PRED, R_SUCC, and A_SUCC). The fifth and sixth columns indicate the size of one sequencer with stubs (generating input according to the protocol and absorbing repeated outputs, see Appendix A.4), minimized for divergence-sensitive branching bisimulation. The seventh column indicates whether a sequencer with stubs contains a deadlock (after minimizing for branching bisimulation to remove livelocks masking deadlocks). The eighth and nineth columns indicate the size of a pipeline of two sequencers (not minimized for branching bisimulation), whenever the composition was possible. The last column indicates whether the model of two sequencers contains a deadlock. The most time-consuming task is the (explicit) generation of the LTS for the composition of two sequencers: depending on the model, this step takes from five seconds to more than several days.[6]

A first observation is that explicitly modeling wires and forks increases the size of the models by several orders of magnitude (compared to models RV). Also, the state-oriented (other than FREE) models are significantly larger then the transition-oriented one.

A second observation is the presence of deadlocks in some compositions of two sequencers. For instance, in the INTUITIVE model with wires and forks represented by LNT gates using rendezvous (RV), the following sequence of rendezvous leads to the deadlock: "R_PRED␣!UP", "R␣!UP", "R_SUCC␣!UP", "G_L␣!DOWN", "G_R␣!DOWN", "A_SUCC␣!UP", "R_PRED␣!UP". The deadlock can be explained by the fact that the composition of two sequencers contains a cycle (formed by the (forking) wires R, G_R, H_R, A, G_L, H_L)[7], which can absorb only a bounded number of inputs (R_PRED and A_SUCC). Because each input must be absorbed twice by the cycle (e.g., R_PRED has to alternate with both R and G_L and in the trace above), the cycle can fill if inputs arrive faster than outputs (R_SUCC and A_PRED) are generated. We deduce that the INTUITIVE model is not appropriate. We also observe that the presence of deadlocks in the TRANSITION models can be used to pinpoint the forks that need not to be isochronic; this provides a valuable information to the hardware designer about possible substantial optimizations in area and performance.

A third observation is that, contrary to the circuit-level model, the composition of two sequencers in the gate-level models is not equivalent to a single sequencer, so that there is no counterpart to the induction result of Section 3. However, when constraining the visible actions (R_PRED, A_PRED, R_SUCC, and A_SUCC) by appropriate stubs eliminating repeated actions (see Appendix A.4), we observe that a

[6] For the missing results for models STATE and PARALLEL, the generation was stopped after 62 hours, at which point each composition had required already 150 GB of RAM and produced a file of more than 100 GB. We also experimented with distributed generation, using up to 80 processors; this also failed due to a lack of disk space (more than 4 TB).

[7] See Appendix B.1 for the corresponding composition expression.

model	forks	one sequencer		one sequencer with stubs			two sequencers		
		states	transitions	states	transitions	lock	states	transitions	lock
INTUITIVE	RV	90	222	8	8	no	308	790	yes
	III	2,586	7,922	8	8	no	288,771	1,093,552	yes
	IIP	6,124	21,454	8	8	no	1,307,889	5,968,266	yes
	IPI	6,475	19,985	1,444	4,141	yes	2,588,785	10,624,729	yes
	IPP	15,422	54,969	4,780	15,402	yes	12,433,518	61,288,551	yes
	PII	6,475	19,985	1,444	4,141	yes	1,562,907	6,452,580	yes
	PIP	15,422	54,969	4,780	15,402	yes	7,291,527	36,213,052	yes
	PPI	14,900	46,486	4,032	13,710	yes	13,450,533	59,014,624	yes
	PPP	38,680	139,558	14,273	53,496	yes	76,518,596	400,442,323	yes
STATE	RV	766	2,406	8	8	no	230,906	906,342	no
	III	33,258	127,380	8	8	no	474,187,601	2,514,512,879	no
	IIP	86,846	374,292	8	8	no	3,002,896,049	18,494,246,894	no
	IPI	82,041	315,312	32,990	113,773	no	2,780,162,577	15,419,740,546	no
	IPP	216,470	931,696	89,238	339,090	no			
	PII	82,041	315,312	32,990	113,773	no	2,795,890,977	15,509,939,437	no
	PIP	216,470	931,696	89,238	339,090	no			
	PPI	194,738	752,128	82,020	316,170	no			
	PPP	531,576	2,287,840	238,574	1,000,138	no			
PARALLEL	RV	916	3,404	8	16	no	342,674	1,625,792	no
	III	54,394	258,456	8	16	no	1,308,613,124	8,868,967,479	no
	IIP	146,002	734,800	8	16	no	8,464,022,990	61,740,118,299	no
	IPI	127,578	606,064	72,431	301,189	no	6,962,294,015	48,364,449,041	no
	IPP	339,047	1,704,667	194,704	854,767	no			
	PII	127,578	606,064	72,431	301,189	no	7,000,306,907	48,656,915,179	no
	PIP	339,047	1,704,667	194,704	854,767	no			
	PPI	292,906	1,391,688	173,252	1,115,840	no			
	PPP	778,468	3,913,592	474,676	2,254,762	no			
FREE	RV	24	186	8	16	no	567	6,517	no
	III	384	2,664	8	16	no	147,360	1,602,532	no
	IIP	768	5,544	8	16	no	589,440	6,741,584	no
	IPI	764	5,306	7,145	37,733	no	583,560	6,500,110	no
	IPP	1,528	11,040	14,346	80,865	no	2,334,240	27,307,944	no
	PII	764	5,306	7,145	37,733	no	582,224	6,485,364	no
	PIP	1,528	11,040	14,346	80,865	no	2,328,896	27,245,680	no
	PPI	1,520	10,568	15,764	90,732	no	2,310,400	26,344,640	no
	PPP	3,040	21,984	33,774	206,894	no	9,241,600	110,534,400	no
TRANSITION	RV	34	112	8	8	no	279	1,101	no
	III	496	1,614	8	8	no	34,461	153,267	no
	IIP	1,320	4,870	8	8	no	238,811	1,270,154	no
	IPI	952	3,155	702	2,077	yes	136,092	665,059	yes
	IPP	2,475	9,313	1,938	6,525	yes	912,702	5,269,597	yes
	PII	952	3,155	702	2,077	yes	135,814	666,185	yes
	PIP	2,475	9,313	1,938	6,525	yes	911,332	5,263,119	yes
	PPI	1,814	6,104	1,335	8,233	yes	537,434	2,855,374	yes
	PPP	4,712	17,972	4,789	18,942	yes	3,624,160	22,403,699	yes

Table 1: State spaces for various gate-level models of up to two sequencers

sequencer is branching bisimilar to the protocol if and only if the forks X and Y are both isochronic (i.e., models RV, III, and IIP).[8] Hence, the LNT model is useful to determine whether a fork must be isochronic (this is the case for X and Y for the sequencer) or not (this is the case for Z, and formally justifies why Z is represented differently than X and Y in [26, Figs. 4–8].

Theoretically, attacks could be analyzed as described in Sect. 3.2 for circuit-level models. However, because the state spaces of gate-level models are much larger, we did not yet attempt this.

5 Related Work

Process calculi, that were initially designed for concurrent systems, are natural candidates for describing asynchronous circuits.

CSP_M, a machine-readable dialect of CSP, was used in [18] to model various asynchronous circuits (C element, n-way mutual exclusion element, tree arbiter) and verify them using the FDR tool. The CSP operators were shown to be suitable for asynchronous circuits, in particular multi-way synchronization facilitates the modeling of isochronic forks. A general model for asynchronous circuits and its translation to CSP is proposed in [31], together with a compositional verification approach suitable for FDR. This makes possible a modeling at three levels (Balsa, handshake expansion, and gate-level), and was applied to the design of realistic circuits, such as the AMULET processor [30].

A modeling style for asynchronous circuits in CCS was proposed in [28] and illustrated on distributed arbiters. In [19] it is shown how to represent DI (*Delay-Insensitive*) asynchronous circuits in CCS. A circuit C was defined as DI if its composition with a FRW (*Foam Rubber Wrapper*) making the communications on input and output wires arbitrarily long is equivalent to C modulo the MUST-testing equivalence, provided by the Concurrency Workbench [8].

The modeling and verification (by equivalence checking) of asynchronous designs using Circal is discussed in [1]. It is also shown how the diagnostic facility of the Circal system helps in determining the forks that are required to be isochronic.

Besides process calculi, a number of other formalisms have been used to model asynchronous circuits at gate-level: Petri nets [33], signal transition graphs [32], XDI [29], Receptive Process Theory [17], stable events [16], and complete trace structures [9].

We do not consider here the verification of security properties, because the main focus of the present paper is the faithful modeling of an asynchronous circuit regardless of its purpose.

6 Conclusion

In this paper, we used the LNT language to formally model an asynchronous circuit at circuit- and gate-level, investigating also various modeling styles for wires and gates. Analyzing these models using the CADP toolbox, we found that they can provide the designer with valuable information, such as the necessity to ensure isochrony of forks. For circuit-level analysis, we used an induction proof to extend results of our attack analysis to a shield of arbitrary size. Obtaining a comparable result for gate-level models is challenging, due to the necessity of stubs. Last, but not least, we believe that, due to their extensibility and state space size, these models provide a challenging benchmark.

[8] Table 1 indicates the size of the sequencers with stubs reduced for divergence-sensitive branching bisimulation. Because the models PARALLEL and FREE contain livelocks (as a gate might generate outputs not triggered by inputs), there are 16 transitions for RV, III, and IIP (one livelock per state).

Acknowledgments. The present work has been partly funded by BPI France and FEDER (*Fonds Européen de Développement Économique Régional*) Rhône-Alpes Auvergne under the national project "SecurIoT-2" supported by the four competitiveness clusters Minalogic, SCS, Systematic Paris-Région, and Derbi. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Andrew Bailey, George A. McCaskill & George J. Milne (1994): *An exercise in the automatic verification of asynchronous designs*. *Formal Methods in System Design* 4(3), pp. 213–242, doi:10.1007/BF01384047.

[2] Edith Beigné, Fabien Clermidy, Pascal Vivet, Alain Clouard & Marc Renaudin (2005): *An Asynchronous NoC Architecture Providing Low Latency Service and Its Multi-Level Design Framework*. In: *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC'05 (New York, USA)*, IEEE Computer Society Press, pp. 54–63, doi:10.1109/ASYNC.2005.10.

[3] Damien Bergamini, Nicolas Descoubes, Christophe Joubert & Radu Mateescu (2005): *BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking*. In Nicolas Halbwachs & Lenore Zuck, editors: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Edinburgh, Scotland, UK*, *Lecture Notes in Computer Science* 3440, Springer Verlag, pp. 581–585, doi:10.1007/978-3-540-31980-1_42.

[4] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs & Frits Schalij (1991): *The VLSI-Programming Language Tangram and its Translation into Handshake Circuits*. In: *Proceedings of the Conference on European Design Automation (Amsterdam, The Netherlands)*, IEEE Computer Society Press, pp. 384–389, doi:10.1109/EDAC.1991.206431.

[5] Aymane Bouzafour, Marc Renaudin, Hubert Garavel, Radu Mateescu & Wendelin Serwe (2018): *Model-checking Synthesizable SystemVerilog Descriptions of Asynchronous Circuits*. In Milos Krstic & Ian W. Jones, editors: *Proceedings of the 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'18), Vienna, Austria*, IEEE, pp. 34–42, doi:10.1109/ASYNC.2018.00021.

[6] Stephen D. Brookes, C. A. R. Hoare & A. W. Roscoe (1984): *A Theory of Communicating Sequential Processes*. *Journal of the ACM* 31(3), pp. 560–599, doi:10.1145/828.833.

[7] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe & Gideon Smeding (2019): *Reference Manual of the LNT to LOTOS Translator (Version 6.8)*. Available at http://cadp.inria.fr/publications/Champelovier-Clerc-Garavel-et-al-10.html. INRIA, Grenoble, France.

[8] Rance Cleaveland, Joachim Parrow & Bernhard Steffen (1989): *The Concurrency Workbench*. In Joseph Sifakis, editor: *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, *Lecture Notes in Computer Science* 407, Springer Verlag, pp. 24–37, doi:10.1007/3-540-52148-8_3.

[9] David L. Dill (1988): *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. Acm distinguished dissertation, Carnegie Mellon University, Pittsburgh, PA, USA. Available at http://reports-archive.adm.cs.cmu.edu/anon/scan/CMU-CS-88-119.pdf.

[10] Doug Edwards & Andrew Bardsley (2002): *Balsa: An Asynchronous Hardware Synthesis Language*. *The Computer Journal* 45(1), pp. 12–18, doi:10.1093/comjnl/45.1.12.

[11] Hubert Garavel & Frédéric Lang (2001): *SVL: a Scripting Language for Compositional Verification*. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang & Danhyung Lee, editors: *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea*, Kluwer Academic Publishers, pp. 377–392. Available at http://cadp.inria.fr/publications/Garavel-Lang-01.html.

[12] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2013): *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes*. Springer International Journal on Software Tools for Technology Transfer (STTT) 15(2), pp. 89–107, doi:10.1007/s10009-012-0244-z.

[13] Hubert Garavel, Frédéric Lang & Wendelin Serwe (2017): *From LOTOS to LNT*. In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science 10500, Springer Verlag, pp. 3–26, doi:10.1007/978-3-319-68270-9_1.

[14] Hubert Garavel & Wendelin Serwe (2017): *The Unheralded Value of the Multiway Rendezvous: Illustration with the Production Cell Benchmark*. In Holger Hermanns & Peter Höfner, editors: *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems (MARS'17), Uppsala, Sweden*, Electronic Proceedings in Computer Science 244, pp. 230–270, doi:10.4204/EPTCS.244.10.

[15] C. A. R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall. Available at http://usingcsp.com/cspbook.pdf.

[16] Tingting Jia, Caihong Li & Anping He (2017): *Modeling and Verification of Circuit with Stable-Event*. In: *Proceedings of the 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), Nanjing, China*, IEEE, pp. 471–475, doi:10.1109/CyberC.2017.73.

[17] Mark B. Josephs (1992): *Receptive Process Theory*. Acta Informatica 29(1), pp. 17–31, doi:10.1007/BF01178564.

[18] Mark B. Josephs (2007): *Gate-level Modelling and Verification of Asynchronous Circuits using CSPM and FDR*. In: *Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'07), Berkeley, California, USA*, IEEE, pp. 83–94, doi:10.1109/ASYNC.2007.19.

[19] Hemangee K. Kapoor & Mark B. Josephs (2004): *Modelling and Verification of Delay-Insensitive Circuits using CCS and the Concurrency Workbench*. Information Processing Letters 89(6), pp. 293–296, doi:10.1016/j.ipl.2003.12.007.

[20] Frédéric Lang (2005): *EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods*. In Judi Romijn, Graeme Smith & Jaco van de Pol, editors: *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'05), Eindhoven, The Netherlands*, Lecture Notes in Computer Science 3771, Springer Verlag, pp. 70–88, doi:10.1007/11589976_6.

[21] Alain J. Martin (1986): *Compiling Communicating Processes into Delay-Insensitive VLSI Circuits*. Distributed Computing 1(4), pp. 226–234, doi:10.1007/BF01660034.

[22] Alain J. Martin (2014): *25 Years Ago: The First Asynchronous Microprocessor*. Computer Science Technical Reports 2014.001, California Institute of Technology, Pasadena, California, USA. Available at https://resolver.caltech.edu/CaltechAUTHORS:20140206-111915844.

[23] David E. Muller (1955): *Theory of Asynchronous Circuits*. Research report 66, University of Illinois at Urbana-Champaign, Department of Computer Science. Available at https://archive.org/details/theoryofasynchro66mull.

[24] Luis A. Plana, P. A. Riocreux, W. J. Bainbridge, Andrew Bardsley, Steve Temple, Jim D. Garside & Z. C. Yu (2003): *SPA – a secure Amulet core for smartcard applications*. Microprocessors and Microsystems 27(9), pp. 431–446, doi:10.1016/S0141-9331(03)00093-0.

[25] Marc Renaudin, Bertrand Folco & Boulahia Boubkar (2018): *Circuit intégré protégé*. Brevet d'invention 16 57129 3 054 344, Institut National de la Propriété Industrielle.

[26] Marc Renaudin, Bertrand Folco & Boulahia Boubkar (2019): *Circuit intégré protégé*. Fascicule de Brevet Europeen EP 3 276 656 B1, European Patent Office.

[27] A. W. Roscoe, C. A. R. Hoare & Richard Bird (1997): *The Theory and Practice of Concurrency*. Prentice Hall. Available at https://archive.org/details/theorypracticeof00rosc.

[28] Ken Stevens, John Aldwinckle, Graham Birtwistle & Ying Liu (1993): *Designing Parallel Specifications in CCS*. In: *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pp. 983–986, doi:10.1109/CCECE.1993.332460.

[29] Tom Verhoeff (1998): *Analyzing Specifications for Delay-Insensitive Circuits*. In: *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'98), San Diego, California, USA*, IEEE, pp. 172–183, doi:10.1109/ASYNC.1998.666503.

[30] X. Wang, M. Kwiatkowska, G. Theodoropoulos & Q. Zhang (2006): *Opportunities and Challenges in Process-Algebraic Verification of Asynchronous Circuit Designs*. In: *Proceedings of the Second Workshop on Globally Asynchronous Locally Synchronous Design (FMGALS'05)*, *Electronic Notes in Theoretical Computer Science* 146, pp. 189–206, doi:10.1016/j.entcs.2005.05.042.

[31] Xu Wang & Marta Z. Kwiatkowska (2007): *On Process-algebraic Verification of Asynchronous Circuits*. *Fundamenta Informaticae* 80(1–3), pp. 283–310. Available at https://content.iospress.com/download/fundamenta-informaticae/fi80-1-3-16?id=fundamenta-informaticae%2Ffi80-1-3-16.

[32] Alexandre Yakovlev, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno & Marta Pietkiewicz-Koutny (1996): *On the Models for Asynchronous Circuit Behaviour with OR Causality*. *Formal Methods in System Design* 9(3), pp. 189–233, doi:10.1007/BF00122082.

[33] Alexandre V. Yakovlev, Albert M. Koelmans, Alexei L. Semenov & David J. Kinniment (1996): *Modelling, Analysis and Synthesis of Asynchronous Control Circuits using Petri Nets*. *Integration* 21(3), pp. 143–170, doi:10.1016/S0167-9260(96)00010-7.

A LNT Models

The LNT model of the shield takes advantage of modules [7] to split the overall model into different files: a module with the definition of basic data types, a (family of) module(s) for the gates, a module defining stub processes, and the main module.

A.1 Data Types, Operations, and Channels

Module `VOLTAGE` defines an enumerated data type for voltages, logical operations (used in the gates), channels for communicating voltages, and processes to represent wires and (isochronic) forks.

```
module VOLTAGE is

type VOLTAGE is
   DOWN, UP
   with "==", "!="
end type
```

```
function NOT (X: VOLTAGE) : VOLTAGE is
   if X == DOWN then
      return UP
   else
      return DOWN
   end if
end function

function _AND_ (X1, X2: VOLTAGE) : VOLTAGE is
   case X1, X2 in
      UP, UP   -> return UP
```

```
    | any, any -> return DOWN
   end case
end function

function _OR_ (X1, X2: VOLTAGE) : VOLTAGE is
   case X1, X2 in
      DOWN, DOWN -> return DOWN
    | any, any   -> return UP
   end case
end function

function MULLER (X1, X2, PREVIOUS: VOLTAGE) : VOLTAGE is
   if (X1 == DOWN) and (X2 == DOWN) then
      return DOWN
   elsif (X1 == UP) and (X2 == UP) then
      return UP
   else
      return PREVIOUS
   end if
end function
```

```
channel LINK is
   (VOLTAGE)
end channel
```

```
-- straight wire (with non-zero delay)
-- also used for modeling an isochronic fork

process WIRE [INPUT, OUTPUT: LINK] is
   var X : VOLTAGE in
      loop
         INPUT (?X);
         OUTPUT (X)
      end loop
   end var
end process
```

```
-- forking wire replicating an input on both outputs in any order

process FORK [INPUT, OUTPUT1, OUTPUT2: LINK] is
   var X : VOLTAGE in
      loop
         INPUT (?X);
         par
            OUTPUT1 (X)
         || OUTPUT2 (X)
         end par
      end loop
```

```
   end var
end process

-------------------------------------------------------------------------------
-- wire stuck at a given voltage

process STUCKAT [W: LINK] (V: VOLTAGE) is
   loop
      W (V)
   end loop
end process

end module
```

A.2 Gates

For each modeling style (see Fig. 5), there is dedicated version of the module GATES.

A.2.1 Transition-Oriented Gates

```
module GATES (VOLTAGE) is

process INV [INPUT, OUTPUT: LINK] (in var X: VOLTAGE) is
   -- inverter
   var RESULT, NEW_RESULT: VOLTAGE in
      RESULT := NOT (X);
      loop
         INPUT (?X);
         NEW_RESULT := NOT (X);
         if NEW_RESULT != RESULT then
            RESULT := NEW_RESULT;
            OUTPUT (RESULT)
         end if
      end loop
   end var
end process

process AND [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary AND gate
   var RESULT, NEW_RESULT: VOLTAGE in
      RESULT := X1 AND X2;
      loop
         select
            INPUT1 (?X1)
         [] INPUT2 (?X2)
         end select;
         NEW_RESULT := X1 AND X2;
         if NEW_RESULT != RESULT then
            RESULT := NEW_RESULT;
            OUTPUT (RESULT)
```

```
         end if
      end loop
   end var
end process

process NOR [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary NOR gate
   var RESULT, NEW_RESULT: VOLTAGE in
      RESULT := NOT (X1 OR X2);
      loop
         select
            INPUT1 (?X1)
         [] INPUT2 (?X2)
         end select;
         NEW_RESULT := NOT (X1 OR X2);
         if NEW_RESULT != RESULT then
            RESULT := NEW_RESULT;
            OUTPUT (RESULT)
         end if
      end loop
   end var
end process

process MULLER [INPUT1, INPUT2, OUTPUT: LINK]
               (in var X1, X2, RESULT: VOLTAGE) is
   -- Muller's C element
   var NEW_RESULT: VOLTAGE in
      loop
         select
            INPUT1 (?X1)
         [] INPUT2 (?X2)
         end select;
         NEW_RESULT := MULLER (X1, X2, RESULT);
         if NEW_RESULT != RESULT then
            RESULT := NEW_RESULT;
            OUTPUT (RESULT)
         end if
      end loop
   end var
end process

end module
```

A.3 Intuitive State-Oriented Gates

```
module GATES (VOLTAGE) is

process INV [INPUT, OUTPUT: LINK] (in var X: VOLTAGE) is
   -- inverter
   loop
      INPUT (?X);
      OUTPUT (NOT (X))
```

```
   end loop
end process

process BINARY [INPUT1, INPUT2: LINK] (in out X1, X2: VOLTAGE) is
   -- accept one of two inputs
   use X1, X2; -- to keep lnt2lotos silent
   select
      INPUT1 (?X1)
   [] INPUT2 (?X2)
   end select
end process

process AND [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary AND gate
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      OUTPUT (X1 AND X2)
   end loop
end process

process NOR [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary NOR gate
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      OUTPUT (NOT (X1 OR X2))
   end loop
end process

process MULLER [INPUT1, INPUT2, OUTPUT: LINK]
               (in var X1, X2, RESULT: VOLTAGE) is
   -- Muller's C element
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      RESULT := MULLER (X1, X2, RESULT);
      OUTPUT (RESULT)
   end loop
end process

end module
```

A.3.1 State-Oriented Gates

```
module GATES (VOLTAGE) is

process INV [INPUT, OUTPUT: LINK] (in var X: VOLTAGE) is
   -- inverter
   loop
      INPUT (?X);
      OUTPUT (NOT (X))
   end loop
end process
```

```
process BINARY [INPUT1, INPUT2: LINK] (in out X1, X2: VOLTAGE) is
   -- accept one or two inputs in arbitrary order
   use X1, X2; -- to keep lnt2lotos silent
   select
      INPUT1 (?X1);
      select
         null
      [] INPUT2 (?X2)
      end select
   [] INPUT2 (?X2);
      select
         null
      [] INPUT1 (?X1)
      end select
   end select
end process

process AND [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary AND gate
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      OUTPUT (X1 AND X2)
   end loop
end process

process NOR [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary NOR gate
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      OUTPUT (NOT (X1 OR X2))
   end loop
end process

process MULLER [INPUT1, INPUT2, OUTPUT: LINK]
               (in var X1, X2, RESULT: VOLTAGE) is
   -- Muller's C element
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      RESULT := MULLER (X1, X2, RESULT);
      OUTPUT (RESULT)
   end loop
end process

end module
```

A.3.2 Parallel State-Oriented Gates

```
module GATES (VOLTAGE) is

process INV [INPUT, OUTPUT: LINK] (in var X: VOLTAGE) is
   -- inverter
   loop
```

```
      select
         INPUT (?X)
      [] null
      end select;
      OUTPUT (NOT (X))
   end loop
end process

process BINARY [INPUT1, INPUT2: LINK] (in out X1, X2: VOLTAGE) is
   -- accept one or two inputs in arbitrary order
   use X1, X2; -- to keep lnt2lotos silent
   par
      select
         INPUT1 (?X1)
      [] null
      end select
   || select
         INPUT2 (?X2)
      [] null
      end select
   end par
end process

process AND [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary AND gate
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      OUTPUT (X1 AND X2)
   end loop
end process

process NOR [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary NOR gate
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      OUTPUT (NOT (X1 OR X2))
   end loop
end process

process MULLER [INPUT1, INPUT2, OUTPUT: LINK]
               (in var X1, X2, RESULT: VOLTAGE) is
   -- Muller's C element
   loop
      BINARY [INPUT1, INPUT2] (!?X1, !?X2);
      RESULT := MULLER (X1, X2, RESULT);
      OUTPUT (RESULT)
   end loop
end process

end module
```

A.3.3 Free State-Oriented Gates

```
module GATES (VOLTAGE) is

process INV [INPUT, OUTPUT: LINK] (in var X: VOLTAGE) is
   -- inverter
   loop
      select
         INPUT (?X)
      [] OUTPUT (NOT (X))
      end select
   end loop
end process

process AND [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary AND gate
   loop
      select
         INPUT1 (?X1)
      [] INPUT2 (?X2)
      [] OUTPUT (X1 AND X2)
      end select
   end loop
end process

process NOR [INPUT1, INPUT2, OUTPUT: LINK] (in var X1, X2: VOLTAGE) is
   -- binary NOR gate
   loop
      select
         INPUT1 (?X1)
      [] INPUT2 (?X2)
      [] OUTPUT (NOT (X1 OR X2))
      end select
   end loop
end process

process MULLER [INPUT1, INPUT2, OUTPUT: LINK]
               (in var X1, X2, RESULT: VOLTAGE) is
   -- Muller's C element
   loop
      select
         INPUT1 (?X1)
      [] INPUT2 (?X2)
      [] OUTPUT (RESULT)
      end select;
      RESULT := MULLER (X1, X2, RESULT)
   end loop
end process

end module
```

A.4 Stubs

```
module STUBS (VOLTAGE) is

------------------------------------------------------------------------
-- behaviour (protocol) of an asynchronous sequencer as given by a
-- (sequential) signal transition graph

process PROTOCOL [R_PRED, A_PRED, R_SUCC, A_SUCC: LINK] is
   loop
      R_PRED (UP);
      R_SUCC (UP);
      A_SUCC (UP);
      R_SUCC (DOWN);
      A_SUCC (DOWN);
      A_PRED (UP);
      R_PRED (DOWN);
      A_PRED (DOWN)
   end loop
end process

------------------------------------------------------------------------
-- process to absorb repeated inputs, transforming a state-oriented
-- model into a transition-oriented one

process ABSORB [W: LINK] (V: VOLTAGE) is
   loop L in
      select
         W (V)
      [] break L
      end select
   end loop
end process

------------------------------------------------------------------------
-- stub processes enforcing the state-oriented protocol with the same
-- gates as the sequencer so as to use the same composition operator
-- to add a stub instead of adding a sequencer
-- contrary to the transition-oriented protocol, repeated outputs are
-- permitted: thus, left and right stubs are different

process STUB_L [R_PRED, A_PRED, R_SUCC, A_SUCC: LINK] is
   var V: VOLTAGE in
      V := UP;
      loop
         ABSORB [A_SUCC] (NOT (V));
         R_PRED (V); R_SUCC (V); -- immediate propagation
         ABSORB [A_SUCC] (NOT (V));
         A_SUCC (V);
         A_PRED (V);
         V := NOT (V)
      end loop
```

```
   end var
end process

process STUB_R [R_PRED, A_PRED, R_SUCC, A_SUCC: LINK] is
   var V: VOLTAGE in
      V := UP;
      loop
         ABSORB [R_PRED] (NOT (V));
         R_PRED (V);
         R_SUCC (V);
         ABSORB [R_PRED] (V);
         A_SUCC (V); A_PRED (V); -- immediate propagation
         V := NOT (V)
      end loop
   end var
end process

end module
```

A.5 Sequencer

```
module SEQUENCER (GATES, STUBS) is

-------------------------------------------------------------------------
-- behaviour of a sequencer as given by the composition of its
-- elementary gates (AND, NOR, INV, and MULLER), explicitly modeling
-- forks by dedicated processes; an isochronic fork is modeled as a
-- wire with a three-party rendezvous on the output
-- there are eight processes, one for each combination of isochronic
-- and parallel (non-isochronic) forks

process SEQUENCER_PPP [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED1, R_PRED2, A_SUCC1, A_SUCC2,
                       G2, H1, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC1, G ->
         MULLER [R_PRED2, A_SUCC1, G] (X1, X2, INIT_C)
   || R_PRED1, H1 -> AND [R_PRED1, H1, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED1, R_PRED2 -> FORK [R_PRED, R_PRED1, R_PRED2]
   || A_SUCC1, A_SUCC2 -> FORK [A_SUCC, A_SUCC1, A_SUCC2]
   || H, H1, H2 -> FORK [H, H1, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

process SEQUENCER_PPI [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED1, R_PRED2, A_SUCC1, A_SUCC2,
```

```
                        G2, H2: LINK]
                       (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC1, G ->
         MULLER [R_PRED2, A_SUCC1, G] (X1, X2, INIT_C)
   || R_PRED1, H2 -> AND [R_PRED1, H2, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED1, R_PRED2 -> FORK [R_PRED, R_PRED1, R_PRED2]
   || A_SUCC1, A_SUCC2 -> FORK [A_SUCC, A_SUCC1, A_SUCC2]
   || H, H2 -> WIRE [H, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

process SEQUENCER_PIP [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED1, R_PRED2, A_SUCC2, G2, H1, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC2, G ->
         MULLER [R_PRED2, A_SUCC2, G] (X1, X2, INIT_C)
   || R_PRED1, H1 -> AND [R_PRED1, H1, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED1, R_PRED2 -> FORK [R_PRED, R_PRED1, R_PRED2]
   || A_SUCC2 -> WIRE [A_SUCC, A_SUCC2]
   || H, H1, H2 -> FORK [H, H1, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

process SEQUENCER_PII [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED1, R_PRED2, A_SUCC2, G2, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC2, G ->
         MULLER [R_PRED2, A_SUCC2, G] (X1, X2, INIT_C)
   || R_PRED1, H2 -> AND [R_PRED1, H2, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED1, R_PRED2 -> FORK [R_PRED, R_PRED1, R_PRED2]
   || A_SUCC2 -> WIRE [A_SUCC, A_SUCC2]
   || H, H2 -> WIRE [H, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
process SEQUENCER_IPP [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED2, A_SUCC1, A_SUCC2, G2, H1, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC1, G ->
         MULLER [R_PRED2, A_SUCC1, G] (X1, X2, INIT_C)
   || R_PRED2, H1 -> AND [R_PRED2, H1, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED2 -> WIRE [R_PRED, R_PRED2]
   || A_SUCC1, A_SUCC2 -> FORK [A_SUCC, A_SUCC1, A_SUCC2]
   || H, H1, H2 -> FORK [H, H1, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

process SEQUENCER_IPI [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED2, A_SUCC1, A_SUCC2, G2, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC1, G ->
         MULLER [R_PRED2, A_SUCC1, G] (X1, X2, INIT_C)
   || R_PRED2, H2 -> AND [R_PRED2, H2, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED2 -> WIRE [R_PRED, R_PRED2]
   || A_SUCC1, A_SUCC2 -> FORK [A_SUCC, A_SUCC1, A_SUCC2]
   || H, H2 -> WIRE [H, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

process SEQUENCER_IIP [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED2, A_SUCC2, G2, H1, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC2, G ->
         MULLER [R_PRED2, A_SUCC2, G] (X1, X2, INIT_C)
   || R_PRED2, H1 -> AND [R_PRED2, H1, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED2 -> WIRE [R_PRED, R_PRED2]
   || A_SUCC2 -> WIRE [A_SUCC, A_SUCC2]
   || H, H1, H2 -> FORK [H, H1, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process
```

```
-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

process SEQUENCER_III [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                       R_PRED2, A_SUCC2, G2, H2: LINK]
                      (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED2, A_SUCC2, G ->
         MULLER [R_PRED2, A_SUCC2, G] (X1, X2, INIT_C)
   || R_PRED2, H2 -> AND [R_PRED2, H2, R_SUCC] (X1, NOT (INIT_C))
   || G2, H -> INV [G2, H] (INIT_C)
   || A_SUCC2, H2 -> NOR [A_SUCC2, H2, A_PRED] (X2, NOT (INIT_C))
   || R_PRED2 -> WIRE [R_PRED, R_PRED2]
   || A_SUCC2 -> WIRE [A_SUCC, A_SUCC2]
   || H, H2 -> WIRE [H, H2]
   || G, G2 -> WIRE [G, G2]
   end par
end process

-------------------------------------------------------------------------
-- behaviour of a sequencer as given by the composition of its
-- elementary gates (AND, NOR, INV, and MULLER), using multiway
-- rendezvous to model forking wires, and a gate to model (binary)
-- wires

process SEQUENCER_RV [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H: LINK]
                     (X1, X2, INIT_C: VOLTAGE) is
   par
      R_PRED, A_SUCC, G -> MULLER [R_PRED, A_SUCC, G] (X1, X2, INIT_C)
   || R_PRED, H -> AND [R_PRED, H, R_SUCC] (X1, NOT (INIT_C))
   || G, H -> INV [G, H] (INIT_C)
   || A_SUCC, H -> NOR [A_SUCC, H, A_PRED] (X2, NOT (INIT_C))
   end par
end process

-------------------------------------------------------------------------
-- sequencer with the same visible interface as PROTOCOL[]

process SEQUENCER_HIDDEN [R_PRED, A_PRED, R_SUCC, A_SUCC: LINK] is
   hide G, H, R_PRED1, R_PRED2, A_SUCC1, A_SUCC2, G2, H1, H2: LINK in
      SEQUENCER_PPP [R_PRED, A_PRED, R_SUCC, A_SUCC, G, H,
                     R_PRED1, R_PRED2, A_SUCC1, A_SUCC2, G2, H1, H2]
                    (DOWN, DOWN, DOWN)
   end hide
end process

-------------------------------------------------------------------------
-- shield with one element (and closed right end)

process SHIELD_1 [R0, A0: LINK] is
   hide R1, A1: LINK in
      par R1, A1 in
```

```
         SEQUENCER_HIDDEN [R0, A0, R1, A1]
      || WIRE [R1, A1]
      end par
   end hide
end process

------------------------------------------------------------------------
-- shield with two elements (and closed right end)

process SHIELD_2 [R0, A0: LINK] is
   hide R1, A1, R2, A2: LINK in
      par
                  R1, A1 -> SEQUENCER_HIDDEN [R0, A0, R1, A1]
      || R1, A1, R2, A2 -> SEQUENCER_HIDDEN [R1, A1, R2, A2]
      || R2, A2 ->         WIRE [R2, A2]
      end par
   end hide
end process

------------------------------------------------------------------------
-- shield with five elements (and closed right end)

process SHIELD_5 [R0, A0: LINK] is
   hide R1, A1, R2, A2, R3, A3, R4, A4, R5, A5: LINK in
      par
                  R1, A1 -> SEQUENCER_HIDDEN [R0, A0, R1, A1]
      || R1, A1, R2, A2 -> SEQUENCER_HIDDEN [R1, A1, R2, A2]
      || R2, A2, R3, A3 -> SEQUENCER_HIDDEN [R2, A2, R3, A3]
      || R3, A3, R4, A4 -> SEQUENCER_HIDDEN [R3, A3, R4, A4]
      || R4, A4, R5, A5 -> SEQUENCER_HIDDEN [R4, A4, R5, A5]
      || R5, A5 ->         WIRE [R5, A5]
      end par
   end hide
end process

------------------------------------------------------------------------
-- shield with ten elements (and closed right end)

process SHIELD_10 [R0, A0: LINK] is
   hide
      R1, A1, R2, A2, R3, A3, R4, A4, R5,  A5,
      R6, A6, R7, A7, R8, A8, R9, A9, R10, A10: LINK
   in
      par
                   R1,  A1  -> SEQUENCER_HIDDEN [R0, A0, R1,  A1]
      || R1,  A1, R2,  A2  -> SEQUENCER_HIDDEN [R1, A1, R2,  A2]
      || R2,  A2, R3,  A3  -> SEQUENCER_HIDDEN [R2, A2, R3,  A3]
      || R3,  A3, R4,  A4  -> SEQUENCER_HIDDEN [R3, A3, R4,  A4]
      || R4,  A4, R5,  A5  -> SEQUENCER_HIDDEN [R4, A4, R5,  A5]
      || R5,  A5, R6,  A6  -> SEQUENCER_HIDDEN [R5, A5, R6,  A6]
      || R6,  A6, R7,  A7  -> SEQUENCER_HIDDEN [R6, A6, R7,  A7]
      || R7,  A7, R8,  A8  -> SEQUENCER_HIDDEN [R7, A7, R8,  A8]
```

```
      || R8,  A8, R9,  A9  -> SEQUENCER_HIDDEN [R8, A8, R9,  A9]
      || R9,  A9, R10, A10 -> SEQUENCER_HIDDEN [R9, A9, R10, A10]
      || R10, A10 ->          WIRE [R10, A10]
      end par
   end hide
end process

end module
```

B EXP Compositions

This section only presents those composition expressions that cannot be inlined in the SVL script (see Appendix C).

B.1 Correct Pipelining of Sequencers

In the EXP expression corresponding to a correct composition, the names of the two sequencers have to be replaced by concrete file names (see also Sect. 3.1). The SVL script (see Appendix C) uses `sed` to generate appropriate instances.

```
hide R, A in
   par R, A in
      rename R_SUCC -> R, A_SUCC -> A in
         "%%%C1%%%"
      end rename
   || rename R_PRED -> R, A_PRED -> A in
         "%%%C2%%%"
      end rename
   end par
```

To generate more informative diagnostic traces to the deadlocks, it is better not to hide any gates and to rename the internal gates of the sequencers so as to distinguish between those of the left and right one.

```
par R, A in
   rename
      R_SUCC -> R, A_SUCC -> A, G -> G_L, H -> H_L,
      R_PRED1 -> R_PRED1_L, R_PRED2 -> R_PRED2_L,
      A_SUCC1 -> A_SUCC1_L, A_SUCC2 -> A_SUCC2_L,
      H1 -> H1_L, H2 -> H2_L
   in
      "%%%C1%%%"
   end rename
|| rename
      R_PRED -> R, A_PRED -> A, G -> G_R, H -> H_R,
      R_PRED1 -> R_PRED1_R, R_PRED2 -> R_PRED2_R,
      A_SUCC1 -> A_SUCC1_R, A_SUCC2 -> A_SUCC2_R,
      H1 -> H1_R, H2 -> H2_R
   in
      "%%%C2%%%"
   end rename
end par
```

Unsurprisingly, this second composition operator yields larger state spaces, e.g.:

model	forks	one sequencer		two sequencers	
		states	transitions	states	transitions
INTUITIVE	RV	130	294	651	1707
INTUITIVE	PPP	94852	357348	429321272	2443520901
TRANSITION	PPP	18136	72974	28586926	189893233

B.2 Short-Circuits

The definition of short-circuits requires synchronization vectors, which are not yet supported inline by SVL.

B.2.1 Short-Circuit R1-R2

```
hide R1, A1, R2, A2, R1R2 in
  label par using
    -- synchronization vectors for unmodified wires
    "R_PRED␣!DOWN" * _           * _               -> "R_PRED␣!DOWN",
    "R_PRED␣!UP"   * _           * _               -> "R_PRED␣!UP",
    "A_PRED␣!DOWN" * _           * _               -> "A_PRED␣!DOWN",
    "A_PRED␣!UP"   * _           * _               -> "A_PRED␣!UP",
    "A1␣!DOWN"     * "A1␣!DOWN" * _               -> "A1␣!DOWN",
    "A1␣!UP"       * "A1␣!UP"   * _               -> "A1␣!UP",
    _              * "A2␣!DOWN" * "A2␣!DOWN"     -> "A2␣!DOWN",
    _              * "A2␣!UP"   * "A2␣!UP"       -> "A2␣!UP",
    _              * _          * "R_SUCC␣!DOWN" -> "R_SUCC␣!DOWN",
    _              * _          * "R_SUCC␣!UP"   -> "R_SUCC␣!UP",
    _              * _          * "A_SUCC␣!DOWN" -> "A_SUCC␣!DOWN",
    _              * _          * "A_SUCC␣!UP"   -> "A_SUCC␣!UP",
    -- synchronization vectors for the short-circuit
    "R1␣!DOWN"     * "R1␣!DOWN" * "R2␣!DOWN"     -> "R1R2␣!DOWN",
    "R1␣!DOWN"     * "R1␣!DOWN" * "R2␣!UP"       -> "R1R2␣!DOWN",
    "R1␣!DOWN"     * "R1␣!DOWN" * "R2␣!UP"       -> "R1R2␣!UP",
    "R1␣!UP"       * "R1␣!UP"   * "R2␣!DOWN"     -> "R1R2␣!DOWN",
    "R1␣!UP"       * "R1␣!UP"   * "R2␣!DOWN"     -> "R1R2␣!UP",
    "R1␣!UP"       * "R1␣!UP"   * "R2␣!UP"       -> "R1R2␣!UP",
    "R1␣!DOWN"     * "R2␣!DOWN" * "R2␣!DOWN"     -> "R1R2␣!DOWN",
    "R1␣!DOWN"     * "R2␣!UP"   * "R2␣!UP"       -> "R1R2␣!DOWN",
    "R1␣!DOWN"     * "R2␣!UP"   * "R2␣!UP"       -> "R1R2␣!UP",
    "R1␣!UP"       * "R2␣!DOWN" * "R2␣!DOWN"     -> "R1R2␣!DOWN",
    "R1␣!UP"       * "R2␣!DOWN" * "R2␣!DOWN"     -> "R1R2␣!UP",
    "R1␣!UP"       * "R2␣!UP"   * "R2␣!UP"       -> "R1R2␣!UP"
  in
    rename R_SUCC -> R1, A_SUCC -> A1 in
      "PROTOCOL.bcg"
    end rename
  || rename R_PRED -> R1, A_PRED -> A1, R_SUCC -> R2, A_SUCC -> A2 in
      "PROTOCOL.bcg"
    end rename
  || rename R_PRED -> R2, A_PRED -> A2 in
      "PROTOCOL.bcg"
    end rename
```

```
  end par
```

B.2.2 Short-Circuit R1-A1

```
hide R1, A1, R2, A2, R1A1 in
  label par using
    -- synchronization vectors for unmodified wires
    "R_PRED␣!DOWN" * _           * _                -> "R_PRED␣!DOWN",
    "R_PRED␣!UP"   * _           * _                -> "R_PRED␣!UP",
    "A_PRED␣!DOWN" * _           * _                -> "A_PRED␣!DOWN",
    "A_PRED␣!UP"   * _           * _                -> "A_PRED␣!UP",
    _              * "R2␣!DOWN" * "R2␣!DOWN"      -> "R2␣!DOWN",
    _              * "R2␣!UP"   * "R2␣!UP"        -> "R2␣!UP",
    _              * "A2␣!DOWN" * "A2␣!DOWN"      -> "A2␣!DOWN",
    _              * "A2␣!UP"   * "A2␣!UP"        -> "A2␣!UP",
    _              * _          * "R_SUCC␣!DOWN" -> "R_SUCC␣!DOWN",
    _              * _          * "R_SUCC␣!UP"   -> "R_SUCC␣!UP",
    _              * _          * "A_SUCC␣!DOWN" -> "A_SUCC␣!DOWN",
    _              * _          * "A_SUCC␣!UP"   -> "A_SUCC␣!UP",
    -- synchronization vectors for the short-circuit
    "R1␣!DOWN"     * "R1␣!DOWN" * _                -> "R1A1␣!DOWN",
    "R1␣!UP"       * "R1␣!UP"   * _                -> "R1A1␣!UP",
    "R1␣!DOWN"     * "A1␣!DOWN" * _                -> "R1A1␣!DOWN",
    "R1␣!DOWN"     * "A1␣!UP"   * _                -> "R1A1␣!DOWN",
    "R1␣!DOWN"     * "A1␣!UP"   * _                -> "R1A1␣!UP",
    "R1␣!UP"       * "A1␣!DOWN" * _                -> "R1A1␣!DOWN",
    "R1␣!UP"       * "A1␣!DOWN" * _                -> "R1A1␣!UP",
    "R1␣!UP"       * "A1␣!UP"   * _                -> "R1A1␣!UP",
    "A1␣!DOWN"     * "R1␣!DOWN" * _                -> "R1A1␣!DOWN",
    "A1␣!DOWN"     * "R1␣!UP"   * _                -> "R1A1␣!DOWN",
    "A1␣!DOWN"     * "R1␣!UP"   * _                -> "R1A1␣!UP",
    "A1␣!UP"       * "R1␣!DOWN" * _                -> "R1A1␣!DOWN",
    "A1␣!UP"       * "R1␣!DOWN" * _                -> "R1A1␣!UP",
    "A1␣!UP"       * "R1␣!UP"   * _                -> "R1A1␣!UP",
    "A1␣!DOWN"     * "A1␣!DOWN" * _                -> "R1A1␣!DOWN",
    "A1␣!UP"       * "A1␣!UP"   * _                -> "R1A1␣!UP"
  in
    rename R_SUCC -> R1, A_SUCC -> A1 in
      "PROTOCOL.bcg"
    end rename
  || rename R_PRED -> R1, A_PRED -> A1, R_SUCC -> R2, A_SUCC -> A2 in
      "PROTOCOL.bcg"
    end rename
  || rename R_PRED -> R2, A_PRED -> A2 in
      "PROTOCOL.bcg"
    end rename
  end par
```

There are no three-party rendezvous: thus, this attack could also be defined for only two sequencers.

B.2.3 Short-Circuit R1-A2

```
hide R1, A1, R2, A2, R1A2 in
  label par using
     -- synchronization vectors for unmodified wires
     "R_PRED␣!DOWN" * _           * _               -> "R_PRED␣!DOWN",
     "R_PRED␣!UP"   * _           * _               -> "R_PRED␣!UP",
     "A_PRED␣!DOWN" * _           * _               -> "A_PRED␣!DOWN",
     "A_PRED␣!UP"   * _           * _               -> "A_PRED␣!UP",
     "A1␣!DOWN"     * "A1␣!DOWN" * _               -> "A1␣!DOWN",
     "A1␣!UP"       * "A1␣!UP"   * _               -> "A1␣!UP",
     _              * "R2␣!DOWN" * "R2␣!DOWN"     -> "R2␣!DOWN",
     _              * "R2␣!UP"   * "R2␣!UP"       -> "R2␣!UP",
     _              * _          * "R_SUCC␣!DOWN" -> "R_SUCC␣!DOWN",
     _              * _          * "R_SUCC␣!UP"   -> "R_SUCC␣!UP",
     _              * _          * "A_SUCC␣!DOWN" -> "A_SUCC␣!DOWN",
     _              * _          * "A_SUCC␣!UP"   -> "A_SUCC␣!UP",
     -- synchronization vectors for the short-circuit
     "R1␣!DOWN"     * "R1␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!DOWN",
     "R1␣!DOWN"     * "R1␣!DOWN" * "A2␣!UP"       -> "R1A2␣!DOWN",
     "R1␣!DOWN"     * "R1␣!DOWN" * "A2␣!UP"       -> "R1A2␣!UP",
     "R1␣!UP"       * "R1␣!UP"   * "A2␣!DOWN"     -> "R1A2␣!DOWN",
     "R1␣!UP"       * "R1␣!UP"   * "A2␣!DOWN"     -> "R1A2␣!UP",
     "R1␣!UP"       * "R1␣!UP"   * "A2␣!UP"       -> "R1A2␣!UP",
     "R1␣!DOWN"     * "A2␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!DOWN",
     "R1␣!DOWN"     * "A2␣!UP"   * "A2␣!UP"       -> "R1A2␣!DOWN",
     "R1␣!DOWN"     * "A2␣!UP"   * "A2␣!UP"       -> "R1A2␣!UP",
     "R1␣!UP"       * "A2␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!DOWN",
     "R1␣!UP"       * "A2␣!DOWN" * "A2␣!DOWN"     -> "R1A2␣!UP",
     "R1␣!UP"       * "A2␣!UP"   * "A2␣!UP"       -> "R1A2␣!UP"
  in
     rename R_SUCC -> R1, A_SUCC -> A1 in
        "PROTOCOL.bcg"
     end rename
  || rename R_PRED -> R1, A_PRED -> A1, R_SUCC -> R2, A_SUCC -> A2 in
        "PROTOCOL.bcg"
     end rename
  || rename R_PRED -> R2, A_PRED -> A2 in
        "PROTOCOL.bcg"
     end rename
  end par
```

B.2.4 Short-Circuit R2-A1

```
hide R1, A1, R2, A2, R2A1 in
  label par using
     -- synchronization vectors for unmodified wires
     "R_PRED␣!DOWN" * _           * _               -> "R_PRED␣!DOWN",
     "R_PRED␣!UP"   * _           * _               -> "R_PRED␣!UP",
     "A_PRED␣!DOWN" * _           * _               -> "A_PRED␣!DOWN",
     "A_PRED␣!UP"   * _           * _               -> "A_PRED␣!UP",
     "R1␣!DOWN"     * "R1␣!DOWN" * _               -> "R1␣!DOWN",
     "R1␣!UP"       * "R1␣!UP"   * _               -> "R1␣!UP",
     _              * "A2␣!DOWN" * "A2␣!DOWN"     -> "A2␣!DOWN",
```

```
      _                * "A2␣!UP"    * "A2␣!UP"        -> "A2␣!UP",
      _                * _           * "R_SUCC␣!DOWN" -> "R_SUCC␣!DOWN",
      _                * _           * "R_SUCC␣!UP"   -> "R_SUCC␣!UP",
      _                * _           * "A_SUCC␣!DOWN" -> "A_SUCC␣!DOWN",
      _                * _           * "A_SUCC␣!UP"   -> "A_SUCC␣!UP",
      -- synchronization vectors for the short-circuit
      "A1␣!DOWN"       * "A1␣!DOWN" * "R2␣!DOWN"      -> "A1R2␣!DOWN",
      "A1␣!DOWN"       * "A1␣!DOWN" * "R2␣!UP"        -> "A1R2␣!DOWN",
      "A1␣!DOWN"       * "A1␣!DOWN" * "R2␣!UP"        -> "A1R2␣!UP",
      "A1␣!UP"         * "A1␣!UP"   * "R2␣!DOWN"      -> "A1R2␣!DOWN",
      "A1␣!UP"         * "A1␣!UP"   * "R2␣!DOWN"      -> "A1R2␣!UP",
      "A1␣!UP"         * "A1␣!UP"   * "R2␣!UP"        -> "A1R2␣!UP",
      "A1␣!DOWN"       * "R2␣!DOWN" * "R2␣!DOWN"      -> "A1R2␣!DOWN",
      "A1␣!DOWN"       * "R2␣!UP"   * "R2␣!UP"        -> "A1R2␣!DOWN",
      "A1␣!DOWN"       * "R2␣!UP"   * "R2␣!UP"        -> "A1R2␣!UP",
      "A1␣!UP"         * "R2␣!DOWN" * "R2␣!DOWN"      -> "A1R2␣!DOWN",
      "A1␣!UP"         * "R2␣!DOWN" * "R2␣!DOWN"      -> "A1R2␣!UP",
      "A1␣!UP"         * "R2␣!UP"   * "R2␣!UP"        -> "A1R2␣!UP"
  in
      rename R_SUCC -> R1, A_SUCC -> A1 in
         "PROTOCOL.bcg"
      end rename
  || rename R_PRED -> R1, A_PRED -> A1, R_SUCC -> R2, A_SUCC -> A2 in
         "PROTOCOL.bcg"
      end rename
  || rename R_PRED -> R2, A_PRED -> A2 in
         "PROTOCOL.bcg"
      end rename
  end par
```

B.2.5 Short-Circuit R2-A2

As for the short-circuit `R1-A1`, there are no three-party rendezvous.

```
hide R1, A1, R2, A2, R2A2 in
  label par using
      -- synchronization vectors for unmodified wires
      "R_PRED␣!DOWN" * _           * _               -> "R_PRED␣!DOWN",
      "R_PRED␣!UP"   * _           * _               -> "R_PRED␣!UP",
      "A_PRED␣!DOWN" * _           * _               -> "A_PRED␣!DOWN",
      "A_PRED␣!UP"   * _           * _               -> "A_PRED␣!UP",
      "R1␣!DOWN"     * "R1␣!DOWN" * _                -> "R1␣!DOWN",
      "R1␣!UP"       * "R1␣!UP"   * _                -> "R1␣!UP",
      "A1␣!DOWN"     * "A1␣!DOWN" * _                -> "A1␣!DOWN",
      "A1␣!UP"       * "A1␣!UP"   * _                -> "A1␣!UP",
      _              * _          * "R_SUCC␣!DOWN"  -> "R_SUCC␣!DOWN",
      _              * _          * "R_SUCC␣!UP"    -> "R_SUCC␣!UP",
      _              * _          * "A_SUCC␣!DOWN"  -> "A_SUCC␣!DOWN",
      _              * _          * "A_SUCC␣!UP"    -> "A_SUCC␣!UP",
      -- synchronization vectors for the short-circuit
      _              * "R2␣!DOWN" * "R2␣!DOWN"      -> "R2A2␣!DOWN",
      _              * "R2␣!UP"   * "R2␣!UP"        -> "R2A2␣!UP",
      _              * "R2␣!DOWN" * "A2␣!DOWN"      -> "R2A2␣!DOWN",
```

```
      _               * "R2␣!DOWN" * "A2␣!UP"       -> "R2A2␣!DOWN",
      _               * "R2␣!DOWN" * "A2␣!UP"       -> "R2A2␣!UP",
      _               * "R2␣!UP"   * "A2␣!DOWN"     -> "R2A2␣!DOWN",
      _               * "R2␣!UP"   * "A2␣!DOWN"     -> "R2A2␣!UP",
      _               * "R2␣!UP"   * "A2␣!UP"       -> "R2A2␣!UP",
      _               * "A2␣!DOWN" * "R2␣!DOWN"     -> "R2A2␣!DOWN",
      _               * "A2␣!DOWN" * "R2␣!UP"       -> "R2A2␣!DOWN",
      _               * "A2␣!DOWN" * "R2␣!UP"       -> "R2A2␣!UP",
      _               * "A2␣!UP"   * "R2␣!DOWN"     -> "R2A2␣!DOWN",
      _               * "A2␣!UP"   * "R2␣!DOWN"     -> "R2A2␣!UP",
      _               * "A2␣!UP"   * "R2␣!UP"       -> "R2A2␣!UP",
      _               * "A2␣!DOWN" * "A2␣!DOWN"     -> "R2A2␣!DOWN",
      _               * "A2␣!UP"   * "A2␣!UP"       -> "R2A2␣!UP"
  in
     rename R_SUCC -> R1, A_SUCC -> A1 in
        "PROTOCOL.bcg"
     end rename
  || rename R_PRED -> R1, A_PRED -> A1, R_SUCC -> R2, A_SUCC -> A2 in
        "PROTOCOL.bcg"
     end rename
  || rename R_PRED -> R2, A_PRED -> A2 in
        "PROTOCOL.bcg"
     end rename
  end par
```

B.2.6 Short-Circuit A1-A2

```
hide R1, A1, R2, A2, A1A2 in
  label par using
     -- synchronization vectors for unmodified wires
     "R_PRED␣!DOWN" * _           * _               -> "R_PRED␣!DOWN",
     "R_PRED␣!UP"   * _           * _               -> "R_PRED␣!UP",
     "A_PRED␣!DOWN" * _           * _               -> "A_PRED␣!DOWN",
     "A_PRED␣!UP"   * _           * _               -> "A_PRED␣!UP",
     "R1␣!DOWN"     * "R1␣!DOWN" * _               -> "R1␣!DOWN",
     "R1␣!UP"       * "R1␣!UP"   * _               -> "R1␣!UP",
     _              * "R2␣!DOWN" * "R2␣!DOWN"     -> "R2␣!DOWN",
     _              * "R2␣!UP"   * "R2␣!UP"       -> "R2␣!UP",
     _              * _          * "R_SUCC␣!DOWN" -> "R_SUCC␣!DOWN",
     _              * _          * "R_SUCC␣!UP"   -> "R_SUCC␣!UP",
     _              * _          * "A_SUCC␣!DOWN" -> "A_SUCC␣!DOWN",
     _              * _          * "A_SUCC␣!UP"   -> "A_SUCC␣!UP",
     -- synchronization vectors for the short-circuit
     "A1␣!DOWN"     * "A1␣!DOWN" * "A2␣!DOWN"     -> "A1A2␣!DOWN",
     "A1␣!DOWN"     * "A1␣!DOWN" * "A2␣!UP"       -> "A1A2␣!DOWN",
     "A1␣!DOWN"     * "A1␣!DOWN" * "A2␣!UP"       -> "A1A2␣!UP",
     "A1␣!UP"       * "A1␣!UP"   * "A2␣!DOWN"     -> "A1A2␣!DOWN",
     "A1␣!UP"       * "A1␣!UP"   * "A2␣!DOWN"     -> "A1A2␣!UP",
     "A1␣!UP"       * "A1␣!UP"   * "A2␣!UP"       -> "A1A2␣!UP",
     "A1␣!DOWN"     * "A2␣!DOWN" * "A2␣!DOWN"     -> "A1A2␣!DOWN",
     "A1␣!DOWN"     * "A2␣!UP"   * "A2␣!UP"       -> "A1A2␣!DOWN",
```

```
    "A1␣!DOWN"      * "A2␣!UP"    * "A2␣!UP"        -> "A1A2␣!UP",
    "A1␣!UP"        * "A2␣!DOWN"  * "A2␣!DOWN"      -> "A1A2␣!DOWN",
    "A1␣!UP"        * "A2␣!DOWN"  * "A2␣!DOWN"      -> "A1A2␣!UP",
    "A1␣!UP"        * "A2␣!UP"    * "A2␣!UP"        -> "A1A2␣!UP"
in
    rename R_SUCC -> R1, A_SUCC -> A1 in
       "PROTOCOL.bcg"
    end rename
|| rename R_PRED -> R1, A_PRED -> A1, R_SUCC -> R2, A_SUCC -> A2 in
       "PROTOCOL.bcg"
    end rename
|| rename R_PRED -> R2, A_PRED -> A2 in
       "PROTOCOL.bcg"
    end rename
end par
```

C SVL Script

The following SVL script generates the expected behavior, performs the circuit-level analysis of the various attacks, and attempts the generation of a pipe-line of two sequencers modeled at gate level, for each of the various models of wires, forks, and gates.

Running this SVL script on a laptop with an Intel® Core™i5 M560 CPU at 2.67 GHz, 8 GB of RAM, Debian GNU/Linux 9, and CADP 2019-k (Pisa) in 32-bit mode takes about 15 minutes.

```
% DEFAULT_PROCESS_FILE="sequencer.lnt"

"!-----------------------------------------------------------------------"
-- choice of an implementation of gates

% SET_MODEL () {
%       # "$1" should be one of:
%       # FREE, INTUITIVE, PARALLEL, STATE, TRANSITION
%       rm -f GATES.lnt
%       $CADP/src/com/cadp_ln ${1}_GATES.lnt GATES.lnt
% }
% SVL_RECORD_FOR_CLEAN GATES.lnt
% SET_MODEL TRANSITION

"!-----------------------------------------------------------------------"
-- pipelined parallel composition of two sequencers

% PIPE () {
% # $1 : C1 (composant a gauche)
% # $2 : C2 (composant a droite)
% # $3 : C1 ||_{R,A} C2

        % sed -e "s/%%%C1%%%/$1.bcg/" -e "s/%%%C2%%%/$2.bcg/" \
        % composition.exp > ${SVL_TMP_PREFIX}_composition.exp
        % SVL_RECORD_FOR_SWEEP ${SVL_TMP_PREFIX}_composition.exp

        "$3.bcg" =
                divbranching reduction of
```

```
                "${SVL_TMP_PREFIX}_composition.exp" ;
% }

------------------------------------------------------------------------
-- circuit-level analysis
------------------------------------------------------------------------

-- generation of shields with one, two, and three sequencers

"PROTOCOL.bcg" = reduction of "PROTOCOL" ;
% PIPE PROTOCOL PROTOCOL PROTOCOL_PROTOCOL
% PIPE PROTOCOL PROTOCOL_PROTOCOL PROTOCOL_PROTOCOL_PROTOCOL

------------------------------------------------------------------------

property INDUCTION_BASIS
        "PROTOCOL || PROTOCOL = PROTOCOL"
        "PROTOCOL || PROTOCOL || PROTOCOL = PROTOCOL"
is
        comparison "PROTOCOL_PROTOCOL.bcg" == "PROTOCOL.bcg" ;
        expected TRUE ;
        comparison "PROTOCOL_PROTOCOL_PROTOCOL.bcg" == "PROTOCOL.bcg" ;
        expected TRUE ;
end property

------------------------------------------------------------------------

"STUCKAT_DOWN.bcg" = reduction of "STUCKAT [W] (DOWN)" ;
"STUCKAT_UP.bcg" = reduction of "STUCKAT [W] (UP)" ;

------------------------------------------------------------------------

property STUCKAT (WIRE, VOLTAGE)
        "wire $WIRE stuck at $VOLTAGE"
is
        -- both sequencers have to synchronize on $WIRE at $VOLTAGE
        branching comparison
                hide R, A in
                        par
                           R, A ->
                                rename R_SUCC -> R, A_SUCC -> A in
                                        "PROTOCOL.bcg"
                                end rename
                        || R, A ->
                                rename R_PRED -> R, A_PRED -> A in
                                        "PROTOCOL.bcg"
                                end rename
                        || "$WIRE" ->
                                rename W -> "$WIRE" in
                                        "STUCKAT_$VOLTAGE.bcg"
                                end rename
                        end par
```

```
                end hide
        >=
        "PROTOCOL.bcg" ;
        expected FALSE ;

        -- the emitting sequencer can put any voltage on $WIRE
        -- the receiving sequencer can only get $VOLTAGE on $WIRE
        % if [ $WIRE = R ] ; then
        %       OTHER_WIRE=A
        %       SEND=SUCC
        %       RECV=PRED
        % else
        %       OTHER_WIRE=R
        %       SEND=PRED
        %       RECV=SUCC
        % fi
        branching comparison
                hide R, A in
                        par
                           "$OTHER_WIRE" ->
                                rename "R_$SEND" -> R, "A_$SEND" -> A in
                                        "PROTOCOL.bcg"
                                end rename
                        || R, A ->
                                rename "R_$RECV" -> R, "A_$RECV" -> A in
                                        "PROTOCOL.bcg"
                                end rename
                        || "$WIRE" ->
                                rename W -> "$WIRE" in
                                        "STUCKAT_$VOLTAGE.bcg"
                                end rename
                        end par
                end hide
        >=
        "PROTOCOL.bcg" ;
        expected FALSE ;
end property

check STUCKAT (R, DOWN) ;
check STUCKAT (R, UP) ;
check STUCKAT (A, DOWN) ;
check STUCKAT (A, UP) ;
```

```
property CUT (WIRE)
        "wire␣$WIRE␣cut"
        "(generates␣warnings␣about␣a␣missing␣synchronization)"
is
        -- both sequencers cannot synchronize on $WIRE
        branching comparison
                hide R, A in
```

```
                par
                   R, A ->
                        rename R_SUCC -> R, A_SUCC -> A in
                                "PROTOCOL.bcg"
                        end rename
                || R, A ->
                        rename R_PRED -> R, A_PRED -> A in
                                "PROTOCOL.bcg"
                        end rename
                || "$WIRE" ->
                        rename W -> "$WIRE" in
                                stop
                        end rename
                end par
        end hide
>=
"PROTOCOL.bcg" ;
expected FALSE ;

-- the emitting sequencer can synchronize on $WIRE
-- the receiving sequencer cannot synchronize on $WIRE
% if [ $WIRE = R ] ; then
%       OTHER_WIRE=A
%       SEND=SUCC
%       RECV=PRED
% else
%       OTHER_WIRE=R
%       SEND=PRED
%       RECV=SUCC
% fi
branching comparison
        hide R, A in
                par
                   "$OTHER_WIRE" ->
                        rename "R_$SEND" -> R, "A_$SEND" -> A in
                                "PROTOCOL.bcg"
                        end rename
                || R, A ->
                        rename "R_$RECV" -> R, "A_$RECV" -> A in
                                "PROTOCOL.bcg"
                        end rename
                || "$WIRE" ->
                        rename W -> "$WIRE" in
                                stop
                        end rename
                end par
        end hide
>=
"PROTOCOL.bcg" ;
expected FALSE ;

-- both sequencer can synchronize on $WIRE,
```

```
        -- but without synchronizing with each other
        -- this kind of wire cut is unrealistic and not detected
        branching comparison
                hide R, A in
                        par "$OTHER_WIRE" in
                                rename R_SUCC -> R, A_SUCC -> A in
                                        "PROTOCOL.bcg"
                                end rename
                        ||      rename R_PRED -> R, A_PRED -> A in
                                        "PROTOCOL.bcg"
                                end rename
                        end par
                end hide
        >=
        "PROTOCOL.bcg" ;
        expected TRUE ;
end property

check CUT (R) ;
check CUT (A) ;

------------------------------------------------------------------------

property SHORTCIRCUIT (WIRE1, WIRE2, RESULT)
        "shortcircuit␣between␣$WIRE1␣and␣$WIRE2"
is
        "composition_shortcircuit_$WIRE1$WIRE2.bcg" =
                branching reduction of
                "composition_shortcircuit_$WIRE1$WIRE2.exp" ;
        branching comparison
                "composition_shortcircuit_$WIRE1$WIRE2.bcg"
        >=
        "PROTOCOL.bcg" ;
        expected "$RESULT" ;
end property

check SHORTCIRCUIT (R1, R2, FALSE) ;
check SHORTCIRCUIT (R1, A1, TRUE) ;
check SHORTCIRCUIT (R1, A2, FALSE) ;
check SHORTCIRCUIT (R2, A1, FALSE) ;
check SHORTCIRCUIT (R2, A2, TRUE) ;
check SHORTCIRCUIT (A1, A2, FALSE) ;

------------------------------------------------------------------------
-- gate-level analysis
------------------------------------------------------------------------

"STUB_L.bcg" = divbranching reduction of "STUB_L" ;
"STUB_R.bcg" = divbranching reduction of "STUB_R" ;

% for MODEL in TRANSITION INTUITIVE FREE STATE PARALLEL ; do
%       SET_MODEL $MODEL
```

```
%       for FORKS in RV PPP PPI PIP PII IPP IPI IIP III ; do

                "$MODEL.$FORKS.bcg" =
                        divbranching reduction of
                        gate hide all but R_PRED, A_PRED, R_SUCC, A_SUCC
                        in "SEQUENCER_${FORKS} (DOWN, DOWN, DOWN)" ;

%               SEQ=$MODEL.$FORKS

%               PIPE STUB_L $SEQ L_$SEQ
%               PIPE L_$SEQ STUB_R L_${SEQ}_R

                branching comparison
                        "L_${SEQ}_R.bcg" == "PROTOCOL.bcg" ;

                deadlock of branching reduction of "L_${SEQ}_R.bcg" ;

%               if [ $FORKS != RV ] ; then
                        -- avoid the generation of large models
%                       continue
%               fi
%               PIPE $SEQ $SEQ ${SEQ}_${SEQ}

                deadlock of branching reduction of "${SEQ}_${SEQ}.bcg" ;
%       done

% done
```