Verifying SystemC with Scenario

Nicolas Ayache, Loïc Correnson, Franck Védrine
CEA, LIST, Boîte 65, Gif-sur-Yvette, F-91191 France
nicolas.ayache@cea.fr, loic.correnson@cea.fr, franck.vedrine@cea.fr

Abstract

This paper proposes a new approach for the analysis and verification of complex systems. The core of the method consists in combining model-checking and abstract interpretation for analysis and verification. The system is modeled by a Labeled Transition System obtained from a SystemC description, and the properties to be verified are formalized as an observer automaton with assertions. To ease the specification of properties, we introduce a dedicated language, named *Scenario*.
The contributions of the paper are twofold: the language for describing the expected properties and behavior of a system as *Scenario*, and a static analysis for verifying such properties.

Keywords: SystemC, static-analysis, compositional verification, model-checking

1. INTRODUCTION

Modern system languages like SystemVerilog or SystemC [9] have widely spread in the industry for modeling complex, component-based systems. A SystemC code typically contains a large number of components that interacts through ports, channels and events. The overall complexity of systems makes standard model-checking or static analysis not applicable.

A common "divide and conquer" approach for such systems is to use an observer automaton to drive the analysis for only a small number among all the possible behaviors of the system at the same time. Although observer automata are widely known, it is not a common formalism to write specifications and properties with. We thus introduce a dedicated *Scenario* language for this purpose.

The scenario language is fully imperative, like pascal or C, and it is able to express many behaviors for the entire system and to specify properties that must occur during the execution of the system. It can also be used for analyzing Quality-of-Service (QoS) properties. Finally, a scenario is compiled into an observer automaton.

Model-checking is a common framework for analyzing component-based systems. For very complex systems, it suffers from exhaustively analyzing *all* the possible states of the system during its execution. We introduce a more *static* analysis method, based on abstract interpretation [4] in order to manipulate directly large collections of system-states, without requiring exhaustive exploration.

We combine the two techniques by defining a semi-abstract model for the product of the SystemC code with the Scenario observer automaton. As an intermediate transformation, we use classical techniques for translating SystemC code into a Labeled Transition System (LTS), for instance by using the SystemC$^{\mathrm{FL}}$ framework [12].

With *Scenario*, we provide engineers with a comfortable and natural programming language for expressing system behaviors and properties as assertion statements. By applying our verification algorithms, such assertions can be proved to be always verified. When an assertion comes to be violated, the engineer has enough feedback on the system context for understanding the reason(s) of the failure. Debugging on an automatically generated counterexample scenario brings such a feedback.

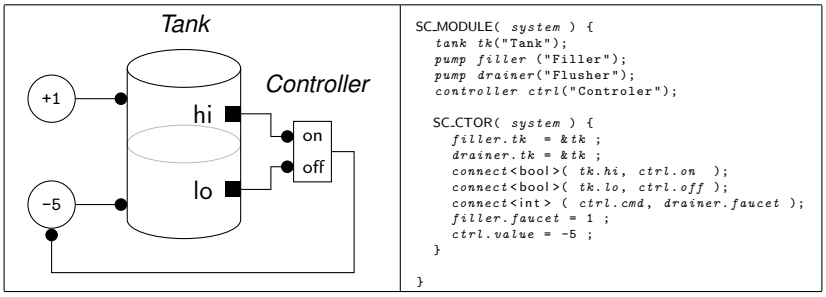
The paper is structured as follows: first, we introduce in Section 2 a metaphoric example that is representative of standard SystemC programming patterns. Then, we present the syntax of the *Scenario* language in Section 3, and the compilation process into the observer automaton back-end in Section 3.4. The second part of the paper, in Section 4, is dedicated to the analysis and verification techniques.

We conclude by generalizing the approach to other input formalisms and more precise analysis methods.

2. COMPONENT-BASED SYSTEMS WITH SYSTEMC

SystemC is used for designing systems with both hardware and software components. It is a C++ library that defines templates for modeling components, communication ports, and connections. Each component is represented by a *class* that holds data. The behavior of each component is computed inside *threads* that are executed concurrently by the SystemC library following a precise operational semantics [9].

As an illustration of such a system, we introduce an example which is representative for system-on-chip descriptions. It is a kind of FIFO, represented by a tank, with a consumer and a producer represented by pumps, as depicted in figure 1. The tank is equipped with two detectors that are sensitive to the high and low level of its content. These detectors are connected to a controller that powers on the draining pump when the level inside the tank becomes too high, until it reaches down the lower level.




```
SC_MODULE( system ) {
  tank tk("Tank");
  pump filler ("Filler");
  pump drainer("Flusher");
  controller ctrl("Controler");

  SC_CTOR( system ) {
    filler.tk  = &tk ;
    drainer.tk = &tk ;
    connect<bool>( tk.hi, ctrl.on  );
    connect<bool>( tk.lo, ctrl.off );
    connect<int> ( ctrl.cmd, drainer.faucet );
    filler.faucet = 1 ;
    ctrl.value = -5 ;
  }

}
```

Figure 1: The "Tank" System

2.1. Description of the Components

Beside the system graphical representation in figure 1 is the corresponding SystemC code. The constructor SC_CTOR creates the instances of the tank, the two pumps and the controller. The template function *connect*, not represented here, just creates a channel and connects an output-port with an input-port with that channel.

The tank and the detectors are modelled by the class-component *tank* in figure 2. Adding or removing content from the tank is performed by calling its *add* method with a positive or negative amount. The two detectors are represented by output ports of type sc_out<bool> that are updated by the *add* method.

The two pumps are modeled by the same class-component *pump* in figure 2. A pump has one thread that fills the tank at a clocked rate. The amount of fuild added or removed from the tank is hold by the *faucet* input port of the pump, of type sc_in<int>. The thread code of a pump is a typical SystemC optimized code for a such a behavior: basically, the thread repeatedly waits for the clock-signal, then fills the tank with the value of the faucet. However, if the faucet comes to be zero, the thread exits the clocked loop and waits until further modification of the faucet occurs. These two waiting points of pump's thread are identified by `@1` and `@2` comments in figure 2.

Finally the controller holds two input ports of type sc_in<bool> for turning *on* and *off* the faucet of the draining pump through the output port *cmd* of type sc_out<int>. The controller is modeled

by a SystemC method-thread introduced by SC_METHOD. A method-thread is sensitive to several events, and is fired each time any of these events is notified. Here, the controller method adjusts the faucet value each time any of the detectors changes its value.

```
SC_MODULE(pump) {

  sc_port<int> faucet ;
  tank* tk ;

  SC_CTOR(pump) { SC_THREAD(run); }

  void run() {
    for(;;) {
      wait( faucet );          //@1
      while( faucet != 0 ) {
        wait( clock );         //@2
        tk->add( faucet );
      }
    }
  }

}
```

```
SC_MODULE(tank) {

  sc_out<bool> hi ;
  sc_out<bool> lo ;

  int level = 0;
  int hi_level = 900 ;
  int lo_level = 600 ;

  SC_CTOR(tank) {}

  void add(int amount) {
    level += amount ;
    hi = level > hi_level ;
    lo = level < lo_level ;
  }

}
```

```
SC_MODULE(controller) {

  sc_in<bool> on ;
  sc_in<bool> off ;
  sc_out<int> cmd ;
  int value ;

  SC_CTOR(controller) {
    SC_METHOD(adjust);
    sensitive << on << off ;
  }

  void adjust() {
    if (off) cmd = 0;
    else if (on) cmd = value ;
  }

}
```

Figure 2: The "Tank" SystemC-modules

2.2. Behaviors of the System

There are a lot of possible behaviors for such a system during simulation. Intuitively, if the draining pump is powered enough to encompass the filling one, the expected behavior is as follows: first the tank is filled in by the filling pump, until the high-detector is fired. Then, the drain is powered by the controller, and the tank is eventually emptied down to the low-detector. At this point, the drain is disabled and the tank starts to be filled again.

A classical model checker must analyze all the concrete states of the system, that ranges between 900 and 1000 different ones in this tiny example. This is in contradiction with the intuitive simple behavior depicted above that only deals alternating filling and draining phases. Such a simple behavior is naturally represented by the following automaton written in pseudo-code style:

```
while (true) { Fill; Drain }
```

where *Fill* is the filling behavior until the high detector is triggered and *Drain* is the symetric draining phase. However, the behavior of the system actually consists in *three* distinct phases. In the first one, the tank is filled from zero to the low-detector level. In the second phase, the tank is filled up to the high-detector level. And in the last phase, the tank is drained back to the low-detector level. So, a more precise description is provided by the following scenario:

```
Fill ; while (true) { Drain; Fill' }
```

By putting distinction between the initial filling phase (*Fill*) and subsequent ones (*Fill'*), the engineer obtains a more precise analysis. In fact, what is needed is a way to specify sequences of phases where different properties holds, even when the threads of the system are in the same configuration of activity. In this example, the tank level is between zero and low level during phase *Fill*, and between high and low level during phase *Fill'*, whereas the tank is filled in during both phases.

3. SCENARIO

Our objective is here to specify the properties and the expected behaviors of a given system. As system languages have specific development flow, they require some kind of integrated verification plateform. SystemC^{FL} [13] defines a proper intermediate plateform between SystemC and Model checkers. Lussy [10] is an intermediary between SystemC and the synchronous Lustre language. Instead of directly exporting the intern results into another formalism, the scenario concept [8] based on Live Sequence Charts rather authorizes reasoning, constructions, manipulation and verification in the formalism itself. With different kinds of scenarios, like use-case scenario, specification scenario, integration scenario, Quality of Service scenario, the scenarios are a hope to follow the development flow with successive transformations.

Many concepts of our scenarios are related to the assume-guarantee approach. However, in a scenario, assume and guarantee expressions are strongly integrated with the control-flow of the system, and is much more expressive. Our fully imperative programming language allows for more complex behavior descriptions. The recent results of [3] extended by [1] may offer a way to automatically generate scenario parts for composing systems.

3.1. Syntax for Scenario

A scenario is a sequence of statements organized into loops, conditionals, and such. Atomic statements allow to compute values for local variables. Expressions may refer to the system components variables and port values.

$$
\begin{array}{rcll}
\mathcal{S} & ::= & \texttt{void} & | \quad \textbf{if}\,(e)\ \mathcal{S}\ \textbf{else}\ \mathcal{S} \\
 & | & x\ \texttt{=}\ e\ \texttt{;} & | \quad \textbf{while}(e)\ \mathcal{S} \\
 & | & \texttt{\{}\ \mathcal{S} \ldots \mathcal{S}\ \texttt{\}} & \\
e & ::= & x \mid f(e, \ldots, e) &
\end{array}
$$

The most important statement for specifying the system behavior deals with synchronization. Such statements are defined in the next section.

3.2. Synchronization Expression

Synchronization with the system process is specified by instruction **sync** ω, where ω is an expression combining event notifications with logical combinators:

$$\omega ::= \top \mid \bot \mid \omega \wedge \omega \mid \omega \vee \omega \mid \neg\omega \mid a \in \mathcal{A}$$

The precise semantics of **sync** ω is defined in Section 6. Intuitively, **sync** ω specifies that the system executes for a while, until enough notified events have been observed for satisfying the ω condition. For instance **sync** $a \wedge b$ blocks until both events a and b are notified.

However, the engineer wants generally to distinguish between different behaviors of the system, with respect to which event condition occurred. Also, different behaviors might occur depending on special conditions. Thus, there is the need for synchronizing with events, but also the need for filtering and verifying properties over the system.

The synchronization expressions are introduced by the following statements:

$$
\begin{array}{rcl}
\mathcal{S} & ::= & \textbf{sync}\ \omega\ \texttt{;} \\
 & | & \mathcal{S}\ \texttt{||}\ \mathcal{S} \\
 & | & \textbf{assume}\ e\ \texttt{;} \\
 & | & \textbf{assert}\ e\ \texttt{;}
\end{array}
$$

The parallel construct $S_1 \,||\, S_2$ is used when the system either behaves either like S_1 or like S_2. Generally, S_1 and S_2 would start by conditions (**assume** e) and synchronizations (**sync** ω).

When the internals of the system does not satisfy an expression e, the execution of the system does not synchronize with the **assume** e statement. Actually, one may *filter out* such simulations, in order to restrict the use-case of the system for instance; otherwise, one may prefer to *signal* an error when such a case is considered as an assertion violation, by using the **assert** e statement.

3.3. Observer Automaton as Scenario Language back-end

Our static analysis works with an observer automaton, compiled from the scenario language. The (finite) states or points of an automaton are denoted by η, and the control-graph of the automaton is defined by a collection of directed transitions. The control-graph is directly obtained by compilation of the scenario (see Section 3.4), thus it is a finite graph.

Each transition from state η to state η' is ruled by one of the three following forms:

Guarding rule **:**	$\eta \xrightarrow{e} \eta'$	e *evaluates to* `true`
Update rule **:**	$\eta \xrightarrow{x:=e} \eta'$	x *is assigned to the evaluation of* e
Synchronization **:**	$\eta \xrightarrow{\omega} \eta'$	*the evolution of the system is synchronized with* ω

The semantics of an observing automaton is expressed by a kind of synchronous product between the system under analysis and the automaton. This product may be considered as a new system that consists of:

- the state η of the observer automaton,
- the values of each variable of the scenario,
- the state of the observed system

The Section 4 explores into more details this product and the associated semantics.

3.4. Compilation of Scenario into Observer Automaton

As a preliminary remark, let us point out that many other languages compile naturally into observer automata. Examples include standard imperative programming languages, but also safety properties in temporal logic formalisms like [6].

For each construction of the *Scenario* language, a compilation scheme is defined for generating an automaton from node η to node η', using intermediate (fresh) nodes $\eta_1 \dots \eta_n$. To simplify notations, any (recursive) call to the compilation scheme for scenario S is also denoted by $\eta \xrightarrow{S} \eta'$.

The compilation scheme for imperative constructs are straightforward:

`x = e ;`	$\eta \xrightarrow{x:=e} \eta'$
if `(e)` S_1 **else** S_2	$\eta \xrightarrow{e} \eta_1 \xrightarrow{S_1} \eta'$, $\eta \xrightarrow{\neg e} \eta_2 \xrightarrow{S_2} \eta'$

`{` $S_1 \dots S_n$ `}`	$\eta \xrightarrow{S_1} \eta_1 \cdots \eta_{n-1} \xrightarrow{S_n} \eta'$
while`(e)` S	$\eta \longrightarrow \eta_0 \xrightarrow{\neg e} \eta'$, $\eta_0 \xrightarrow{e} \eta_1 \xrightarrow{S} \eta_0$

Remark that other standard forms can be added to enrich the language, such as **for** loops and **break** statements.

For compiling assertions, we introduce a special node `fail` from which no transition exits. Any simulation that synchronizes with the scenario and fall into the `fail` node corresponds to the violation of the expected property. Other synchronization statements compilation scheme are straightforward:

sync ω **;**	$\eta \xrightarrow{\omega} \eta'$
$S_1 \,\|\|\, S_2$	$\eta \xrightarrow{S_1} \eta'$, $\eta \xrightarrow{S_2} \eta'$

assume `e ;`	$\eta \xrightarrow{e} \eta'$
assert `e ;`	$\eta \xrightarrow{e} \eta'$, $\eta \xrightarrow{\neg e}$ `fail`

3.5. Scenario and the tank example

Now, let us specify some example behaviors like "*filling until hi/lo is notified*" with scenario. This is simply identified by a synchronization statement **sync** as follows, where *main* denotes the instance of the system:

```
sync main.tank.hi ;
while (true) {
  sync main.clock ;
  sync main.tank.hi ;
```

```
}
```

As another example, it is also possible to filter out some system simulations, by excluding those simulations where an event would occur. Hence, a more precise scenario can be written as follows:

```
sync main.tank.hi ;
while (true) {
  sync main.tank.lo ∧ ¬main.tank.hi ;
  sync main.tank.hi ∧ ¬main.tank.lo ;
}
```

Finally, by using local variables the following scenario states that the tank's level has decreased during the draining phase:

```
sync S.tank.hi ;
while (true) {
  int mark = main.tank.level ;
  sync main.tank.lo ;
  assert main.tank.level ≤ mark ;
  sync main.tank.hi ;
}
```

4. ANALYSIS & VERIFICATION

In this section, we introduce a practical framework for analyzing a system with respect to the observer automaton compiled from a scenario. Although our approach reuses existing techniques, we adapted them in order to fit well with each others.

Recall the overall methodology. The SystemC description of the components to be analyzed is translated in some LTS automaton. On another side, an observer automaton is obtained by compilation of the scenario to be analyzed.

We then define a product analysis of these two automata, that operates on a semi-abstract model. This model is precise over the states of the automaton, but abstracts the variables with over-approximations, following the abstract-interpretation theory [4].

Thus, applying model-checking techniques together with fixpoint computation over abstract domains, we obtain efficiently an over-approximation of all the possible executions that are represented by the scenario. The scenario is said to be *verified* when there is no execution that falls into the `fail` node.

This section illustrates the different parts of the methodology. First, we briefly recall how an LTS can be obtained from SystemC code ; the same process can be adapted for other source languages such as VHDL, Lustre or Signal. Second, we define the abstract model and illustrate how to compute the product between the LTS and the observing automaton. Finally, we illustrate the technique with an example scenario over the "tank" system.

4.1. System LTS-Model

There are many techniques to translate SystemC code into an LTS. For instance, one can use the SystemC^{FL} framework. We choose LTS and not an intern formalism like [11] or finite state machine [7] first to make this language independant from SystemC, and second to be able to build it and to refine it on the fly.

The SystemC framework commonly distinguishes between different phases for the execution of the system:

- During the *elaboration phase*, components are initialized, ports are connected with each others, and threads are defined and spawned.

- During the *simulation phase*, the SystemC library schedules thread executions, communication through ports and event notifications. The simulation phase consists of different levels of *cycles*, summarized below [9] Section 4.2.1:
  1. all runnable threads are executed until they block on a wait.
  2. notified events are propagated to wake up the blocked threads = evaluate phase.
  3. port values are updated through channels = update phase.
  4. when 1-3 loop is finished, the threads specified to wake-up at 0 unit of time effectively wake-up and the execution goes back to 1 = delta notification.
  5. when 1-4 loop is finished, time advance is notified and the execution goes back to 1 = timed notification phase.

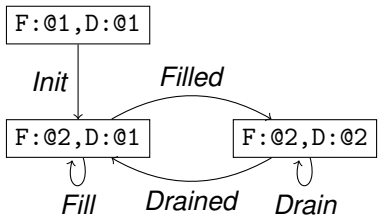
For analyzing a system with scenario, we assume the elaboration phase to be terminated, so that all components, ports and thread are completely defined and initialized. Then, the system can be represented as a collection of p threads that operates on m shared variables.

For analyzing system-level properties, we are not interested in the very low-levels of SystemC scheduling. Rather, we focus on the evolution of the system at δ-cycle. At the beginning of a δ-cycle, each thread is waiting on a wait statement. We call *configuration* of the system the array $(p_1, \ldots, p_n)$ registering the waiting statements of each thread.

The LTS model of the system thus consists in one distinct state for any possible system configuration. Each transition of the LTS is obtained by collecting the effects of all internal steps of execution and conditions until a new waiting-configuration is reached. Effects consist of conditions, variable modifications and event notifications.

For determining these transitions, we must take into account with model-checking techniques the possible interleaving of the SystemC scheduler. However, this exhaustive exploration never goes beyond the next waiting configuration, thus avoiding much of the possible combinatorial explosion in practice.

For the "tank" example, we have few accessible configurations and transitions. There is one thread running for each pump, one method-thread for the controller, and one thread that shortly terminates for initiating the filling pump. The initiating thread has no waiting point and the controller thread has only one implicit waiting point. Thus, configurations are completely defined by the two pump's thread waiting points. See figure 3 for a (simplified) illustration of the LTS obtained for the "tank" system (where F and D are the filling and draining pumps, T denotes the tank).



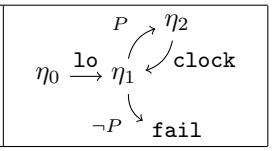
Transition	Effects
Init	$F.faucet$=1
Fill	$T.level$<900, $T.level$+=1
Filled	$T.level$>=900, $T.level$+=1, $D.faucet$=-5
Drain	$T.level$>600, $T.level$+=1-=5,
Drained	$T.level$<=600, $T.level$+=1-=5,

Figure 3: The "Tank" Transition System

4.2. Compiling the Scenario

To illustrate our methodology, we introduce the following example of scenario for representing the "tank" behavior:

```
sync main.tank.lo ;
for(;;) {
  assert 695 <= main.tank.level <= 905 ;
  sync clock ;
}
```



This scenario first waits for the tank to be filled above the lower detector. Then, it asserts that the tank level remains between the lower and higher detectors. Notice that the range have been enlarged with respect to detectors levels, in order for the controller action to take place.

4.3. The Abstract Model

For modeling the product of the system-LTS and the observer automaton of the scenario, we introduce an abstract model for representing a collection of many system states. Contrary to classical model checking techniques where each variable is assigned to one *single* value, we model the possible values of (integer) variables with an *interval* including all its reachable values.

This over-approximation is inspired by abstract-interpretation techniques. In practice, we use more precise (but more complex) abstract domains, that enable to take into account relational properties between variables.

The abstract model represents the possible configurations of the system that might occur at each point of the scenario. For each possible configuration, both the shared variables of the system and the local variables of the scenario are represented by intervals.

Thus, at each point η of the scenario, we have a matrix $\mathcal{D}$, with one row for each configuration, and one column for each of the n processes and the m variables:

$$\mathcal{D} ::= \begin{array}{|c|c|} \hline \mathit{Config.} & \mathit{Variables} \\ \hline p_1^1 \dots p_n^1 & I_1^1 \dots I_m^1 \\ \vdots & \vdots \\ p_1^r \dots p_n^r & I_1^r \dots I_m^r \\ \hline \end{array}$$

On the “tank” example, the only variable of interest if the tank level. The other ones are not represented in the following examples to save place. Here, we represent the matrices with four columns: two for the filling and draining waiting-points of the pumps, and two columns for the draining pump one, and two columns for the lower and upper bounds of the tank.

Initially, the domain representing the state of the system is:

$$D(\eta_0) = \begin{array}{|c|c|} \hline \texttt{@1,@1} & \texttt{0..0} \\ \hline \end{array}$$

4.4. Analysis

The instruction **sync** $\mathit{main.tank.lo}$ synchronizes with the system and the synchronization result assumes the tank level is over 600. Then, the first approximation of the model at point η_1 is:

$$D(\eta_1) = \begin{array}{|c|c|} \hline \texttt{@2,@1} & \texttt{601..601} \\ \hline \end{array}$$

The next iteration enters the loop, verifies the assert and synchronizes with the clock. As it returns in the same configuration than η_1 a merge occurs with the previous state, resulting in

$$D(\eta_1) = \begin{array}{|c|c|} \hline \texttt{@2,@1} & \texttt{601..602} \\ \hline \end{array}$$

Remark also that $D(\eta_2) = D(\eta_1)$. After few steps of approximations, the fixpoint is not yet reached. Following abstract interpretation methodology, we apply *widening* and *narrowing* strategies [5] to reach the fixpoint. This leads to:

$$D(\eta_1) = \begin{array}{|c|c|} \hline \texttt{@2,@1} & \texttt{601..900} \\ \hline \end{array}$$

However, a new transition appears from this configuration when the high detector is triggered. We thus find a new configuration where the draining pump has been powered on:

$$D(\eta_1) = \begin{array}{|l|l|}\hline \texttt{@2,@1} & \texttt{601..900} \\ \hline \texttt{@2,@2} & \texttt{901..901} \\ \hline \end{array}$$

Again, the domain is not stable, and exploration must continue. In a similar way to filling phase, we find the next approximation:

$$D(\eta_1) = \begin{array}{|l|l|}\hline \texttt{@2,@1} & \texttt{601..900} \\ \hline \texttt{@2,@2} & \texttt{897..901} \\ \hline \end{array}$$

Applying again widening and narrowing, we finally reach the correct definitive fixpoint:

$$D(\eta_1) = D(\eta_2) = \begin{array}{|l|l|}\hline \texttt{@2,@1} & \texttt{601..900} \\ \hline \texttt{@2,@2} & \texttt{597..901} \\ \hline \end{array}$$

Remark the assertion verification is compatible with the tank level interval in both configurations. As the **fail** point is not reachable, the synchronous product ensures all local properties are verified on the scenario.

5. CONCLUSION

Scenarios reveal to be a very interesting paradigm to drive the formal verification of component-based systems. The synchronization verdict not only delivers a success or failure verdict, but also a stable description of (1) the domain of the scenario variables, (2) the possible reachable system configurations, (3) the domain of the system variables, and finally (4) the possible sets of notified events at each scenario point. Note that on a failure verdict, an automatic refinement like [2] may generate more structure in the scenario to make it successfully synchronize.

Many extensions and applications of the scenario language are possible. To cite only few of them, remark that an observer automaton is very closed to the LTS of a system. Actually, it is possible to consider scenario for simulating, abstracting and specifying components of the system. In such a context, applying the synchronous product between two scenarios may be considered as assembling component specifications.

ACKNOWLEDGMENT

This work is supported by the French RNTL (Réseau National des Technologies Logicielles) project APE.

Bibliography

[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proceedings of 17th International Conference Computer Aided Verification*, volume 3576 of *LNCS*, pages 548–562. Springer-Verlag, 2005.

[2] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Proceedings of FMCAD 2002*, volume 2517 of *LNCS*, pages 33–51. Springer-Verlag, 2002.

[3] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of TACAS 2003*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.

[4] P. Cousot and R. Cousot. Abstract interpretation frameworks. In *Journal of Logic and Computations*, pages 511–547, 1992.

[5] D. Gopan and T. W. Reps. Lookahead widening. In *Proceedings of the* 18^{th} *International Conference on Computer Aided Verification, CAV 2006.*, LNCS, pages 452–466. Springer-Verlag, 2006.

[6] D. Große and R. Drechsler. Formal verification of LTL formulas for SystemC designs. In *IEEE International Symposium on Circuits and Systems (ISCAS 2003)*, volume 5, pages 245–248, 2003.
[7] A. Habibi, H. Moinudeen, and S. Tahar. Generating finite state machines from SystemC. In Georges G. E. Gielen, editor, *DATE Designers' Forum*, pages 76–81. European Design and Automation Association, Leuven, Belgium, 2006.
[8] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
[9] Open SystemC Initiative. SystemC v2.1 language reference manual, 2005.
[10] L. Maillet-Contoz M. Moy, F. Maraninchi. Lussy: a toolbox for the analysis of systems-on-a-chip at the transactional level. In *Proceedings of Fifth International Conference on Application of Concurrency to System Design (ACSD 2005)*., 2005.
[11] L. Maillet-Contoz M. Moy, F. Maraninchi. Pinapa: The extraction tool for SystemC descriptions of systems-on-a-chip. In *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT) 2005*, 2005.
[12] K.L. Man. Formal communication semantics of SystemC^{FL}. In *IEEE Proceedings of the 8th Euromicro Conference on Digital System Design - DSD05*, 2006.
[13] K.L. Man, A. Fedeli, M. Mercaldi, and M.P. Schellekens. SystemC^{FL}: An infrastructure for a tlm formal verification proposal (with an overview on a tool set for practical formal verification of SystemC descriptions). In *Proceedings of the IEEE East-West Design & Test Workshop EWDTW*, 2006.

6. APPENDIX: ω SYNCHRONIZATION

As presented above, the behavior of the system under analysis is observed through of *event conditions*. Recall that a simulation of the system corresponds to a sequence of triggered events $A_1 \dots A_n$.

It is common to use rewriting rules for expressing the semantics of logical expressions. Thus, the verification of a condition ω is basically defined by standard boolean reduction rules:

$$\begin{array}{cc} \omega \wedge \top \triangleright \omega, & \omega \vee \top \triangleright \top \\ \omega \wedge \bot \triangleright \bot, & \omega \vee \bot \triangleright \omega \\ \neg\top \triangleright \bot, & \neg\bot \triangleright \top \\ \multicolumn{2}{c}{\dots} \end{array}$$

Such common rules are refined into two *distinct* operations to distinguish the observation of *positive* and *negative* events previously introduced:

Trigger ($\triangleright_A$): when the system triggers a set A of events, we define $\omega \triangleright_A \omega'$ to be $a \triangleright_A \top$ when $a \in A$, completed by all possible reductions through $\triangleright$.

Accept ($\triangleright_?$): when condition ω only contains events a in negative position (such as $\neg a$), it should be reduced to $\top$. This is endorsed by defining $\omega \triangleright_? \omega'$ to be $a \triangleright_? \bot$ for any a, completed by all possible reduction through $\triangleright$.

Combining these two operations, one can say that condition ω is fulfilled by the set A of triggered events when $\omega \triangleright_A \omega'$ and $\omega' \triangleright_? \top$. This exactly means that *positive* constraints of ω are matched by A, and that any remaining constraint is living in *negative* position.

For instance, consider the condition $a \wedge \neg b$ which intuitively means: "*synchronize with event 'a' unless event 'b' occurs*." Depending on which combination of events a, b and c have been triggered by the system, we would have the following results:

$$\begin{array}{cccc} a \wedge \neg b & \triangleright_{\{a\}} & \neg b & \triangleright_? \top \\ a \wedge \neg b & \triangleright_{\{b\}} & \bot & \triangleright_? \bot \\ a \wedge \neg b & \triangleright_{\{c\}} & a \wedge \neg b & \triangleright_? a \\ a \wedge \neg b & \triangleright_{\{a,b\}} & \top \wedge \neg\top & \triangleright_? \bot \end{array}$$

In the last case, the reduction by $\triangleright_{\{a,b\}}$ was not fully applied in order to show the intermediate result: both a and b are triggered, and finally the condition is not met. We can now define how to synchronize a *simulation* of the system with a *condition*:

Definition 1 (Synchronization) *Consider a condition ω and a simulation $\sigma \xrightarrow{A_1} \ldots \xrightarrow{A_n} \sigma'$. The synchronization of the system transition from σ to σ' with the condition ω is denoted and defined by:*

$$\sigma \xrightarrow{\omega} \sigma' \iff \omega \triangleright_{A_1} \ldots \triangleright_{A_n} \omega' \triangleright_? \top$$

providing n is minimal for this property.

Such a definition exactly models the intuition of observing a simulation of the system until the condition holds: the events A_i triggered during the simulation of the system successively reduce the condition ω, until only negative constraint remains.