Scenarios For Validating SystemC Descriptions
Nicolas Ayache, Loïc Correnson, Franck Védrine
CEA, LIST, Software Reliability Lab, Boîte 65
CEA Saclay, Gif sur Yvette, F-91191 FRANCE

Abstract: - Verification is always a major bottleneck for implementing systems. In this paper, we show how to use an approach based on scenario in order to grasp such systems. In this context, SystemC plays a particular role since the same language enables to describe systems at various modeling levels. To reach verification besides the development flow, our approach based on scenario combines the synchronization algorithms of Model Checking with Abstract Interpretation techniques. Following the bottom-up design of a concrete example, we show how the scenarios help to qualify the implementation and to optimize QoS properties; we also describe the underlying technologies.

Key-Words: - Scenario, SystemC, Formal Methods, Verification, Model Checking, Design Flow

1 Introduction
According to the “International Technology Roadmap For Semiconductors 2007” [1], verification is the major bottleneck for designing systems integrating both software and hardware. For such projects, the verification engineers are at least 2 times more numerous than developers with a ratio of 3 for 1 for the most complex systems.

To reduce the verification costs, a divide and conquer strategy is systematically applied at various levels. This strategy applies for (1) codesign languages describing both hardware and software, (2) Transactional Level Modeling in SystemC TLM [2], (3) Bus Cycle Accurate Modeling in SystemC, (4) Synthesizable Descriptions in VHDL, Verilog, RTL, netlists. This strategy applies with a hierarchical top-down design flow. Conversely, verification is required to ensure correction and consistency properties between all these implementation levels.

In practice, the verification should integrate partial models, modular design and implicit or explicit contextual constraints. Formal models and tests offer some verification consistency in such design flow. The scenarios, as defined by [3] through Live Message Sequence Charts, are also a good candidate to play an intermediate role between formal models like [4,5,6,7,8] and tests.

In this paper, we define a Scenario language for SystemC inspired by these techniques that extends the assume-guarantee [9] verification with observer automata [10,11]. We show how to use them in a bottom-up verification approach. The goal is to represent the global behavior of the system sequentially, in opposition to the parallel descriptions of SystemC components.

Moreover, the Scenario language we propose benefits from good formal manipulation techniques, such as composition, interface analysis, abstraction and simulation. Thus, the Scenario language is a practical and effective formalism to support verification requirements during the complete design process.

2 The FIFO example
We model here a simple FIFO (First In First Out Queue). The read and write operations may block until the queue becomes (respectively) non-empty or non-full. To analyze the behavior of the FIFO, we use a system that consists of three components: the queue, a producer and a consumer as illustrated in Fig. 1.

![Fig. 1: The FIFO Example](image)

At this early design stage, we assume there is no information available about the internals of the producer and the consumer (partial models). Moreover we would suppose here that the size of the queue is not yet fixed. Hence, we model the producer and the consumer components by asynchronous loops that write and read the queue at random dates. The code for the FIFO is parameterized by the actual size of the queue. The SystemC code for such a system is given in Fig. 2.

This example illustrates a common situation during the System Design flow where some components are fully specified, while other ones remain in a more abstract description level. For instance, optimizing the size of the queue is an objective for analyzing this system. More generally, our framework is oriented at optimizing Quality of Service (QoS) for such systems.

Validating such models actually requires handling with synchronization and interleaving of each compo-
### 3 The Scenario Language

Consider now several components connected with each others inside a complex system. The elements of interest for describing the behavior of such a system are:

- The processes running in the system.
- The events used by process for synchronization.
- The input/output ports data flow.
- The shared variables.

The scenario language aims at describing the sequence of event notifications, port I/O events, and shared variable modifications that occur for these items during the execution of the system. Thus, scenarios are sequential descriptions of the system behavior. The most natural notation for such descriptions is some kind of imperative programming language.

Thus, the basic constructs of our scenario language are made of sequences, while and for loops, conditionals, etc. Local variables and references to the system components (ports and events) improve expressiveness. Several special statements are introduced to synchronize the scenario with the system execution. The simpler one, `sync(a)` let the system running one or several steps until the first notification of event ‘a’. Intuitively, a piece of scenario is a generator for sequences of such statements that synchronize with the system. The language is presented in Fig. 3.

The most important constructs of the language are: the observation of system internals (variables and ports), the synchronization `select(…)` statement, the random expression `rand()` and the filtering statements `assume` and `assert`.

Before illustrating how to use the scenario language the system bottom-up validation process, let us explain the semantics of the core constructs. For instance, consider two events `a` and `b`, and the synchronizing statement:

```
select { event a & !b: A; event b & !a: B } C;
```

It represents any execution of the system, that either:

- triggers `a`, but not `b` - the environment has behaved according to scenario A - then behaves as scenario C, or
- triggers `b`, but not `a` - the environment has behaved according to scenario B - then behaves as scenario C.

The random expression `rand()` introduces unknown values that may also lead to nondeterministic behaviors. Finally, assume and assert statements introduce a more precise semantics for scenario. Actually, a scenario can be considered as a filter that associates a verdict to any execution trace of the system. Verdicts range over:

- **OK**: the trace is synchronized with the scenario;
- **OUT**: the trace is out of the scenario’s behaviors;
- **FAIL**: the trace is accepted by the scenario, but some assertion of the scenario is violated.

When violated, an assume statement assigns the OUT verdict to the trace, whereas the assert statement assigns the FAIL verdict to it. Notice that in the case where any possible trace is out of the scenario, a FAIL verdict is reported.

As a simple example, consider the use-case scenario besides.

```
for (int t = 0; t < 4; t++)
  { sync(queue.write_event);
    sync(queue.read_event);
  }
```

---

**Fig. 2: The FIFO Code**

```cpp
template <int MAX>
SC_MODULE(system) {
  producer<MAX> prod;
  consumer<MAX> cons;
  fifo<MAX> queue;
  SCCTOR(system) ...
    prod.queue = &queue;
    cons.queue = &queue;
}:

template <int MAX>
SC_MODULE(fifo) {
  char data[MAX];
  int num, fst;
  sc_event read_event;
  sc_event write_event;
  SC_CTOR(fifo) {
    prod.queue = &queue;
    SCCTOR(fifo); ...
    cons.queue = &queue;
  }

  void write(char c) {
    if (num == MAX)
      wait(read_event);
    data[fst++ % MAX] = c;
  }

  void read() {
    if (num == 0)
      wait(write_event);
    char read;
    read = data[fst-- % MAX];
    read_event.notify();
    return read;
  }

  int num() {
    return num;
  }

  { num = num - 1;
    std::cout << queue->read();
    if (num == 0)
      wait(read_event);
    return num;
  }

  void copy(fifo&amp; o) {
    for (int i = 0; i < num; i++)
      o.data[i] = data[i];
  }

  ...;
}
```

**Fig. 3 The Scenario Language**

- triggers `a`, but not `b` - the environment has behaved according to scenario A - then behaves as scenario C, or
- triggers `b`, but not `a` - the environment has behaved according to scenario B - then behaves as scenario C.

The random expression `rand()` introduces unknown values that may also lead to nondeterministic behaviors. Finally, assume and assert statements introduce a more precise semantics for scenario. Actually, a scenario can be considered as a filter that associates a verdict to any execution trace of the system. Verdicts range over:

- **OK**: the trace is synchronized with the scenario;
- **OUT**: the trace is out of the scenario’s behaviors;
- **FAIL**: the trace is accepted by the scenario, but some assertion of the scenario is violated.

When violated, an assume statement assigns the OUT verdict to the trace, whereas the assert statement assigns the FAIL verdict to it. Notice that in the case where any possible trace is out of the scenario, a FAIL verdict is reported.

As a simple example, consider the use-case scenario besides.
This scenario assigns an OK verdict to any trace that consists of a 4-times repeated sequence of two elements written in the queue followed by one element read out of it. By adding assume and assert conditions along this scenario, the engineer can specify various use-cases and/or specifications. We explore in the next section of the paper how to use scenario for system design, verification and debug.

4 Scenario and Bottom-Up Verification

The scenarios can have many different uses during the design process. Each use requires its own programming style for scenario-code. We also introduce some formal manipulations of scenario-code for translating a scenario from one style into another one(s). All these manipulations actually rely on the synchronous product algorithm, which is defined in section 5.1.

The bottom-up design process leads to the interactions between several actors and assets: a specification for several components, a specification for the integration, an implementation for the components and eventually, customization of all these components to correctly work with each others. The design process involves writing such specifications and verifying the consistency of implementations with them. We propose to use scenario for that purpose, at both component and system level. We thus distinguish the scenario to be written by the implementers of the components from those to be manipulated by the integrators.

Use-case scenario. A use-case is defined during the specification as a contract between the integrator and the implementer. Such a scenario is deterministic and only refers to the public interface of the components. Like UML message sequence charts, this use-case is actually closed to a test-case.

Debug scenario. The debug scenario is defined on-the-fly for debugging the implementation. It is a very precise sequence of events. By exploring a debug scenario, we provide the implementer with a method to detect and correct implementation errors. Using debug scenario can be considered as a kind of unit testing.

Protocol scenario. When two components are connected with complex data exchanges, a protocol of data transportation is required. Such a protocol organizes the possible sequence of events on input and output channels. A protocol scenario is generally non-deterministic in order to represent several correct data flows. It also makes use of assume/assert statements to specify the accepted ranges of input data. However, the protocol scenario only specifies some communication convention between components, not their functional behavior. By requiring successful synchronization with each component implementation, the integrator gets robustness certificates for the final system.

Implementation scenario. Created by the implementer, the implementation scenario defines the actual behavior of the components, rather than its external specification. It refers to the internals of the component. Thus, on the basis of such a scenario, it is possible to prove the consistency of the component implementation with its specification. It also enables to compute or approximate QoS variables.

Interface scenario. When a component becomes mature enough, one simplifies its functional behavior by considering its interface only. The interface scenario cuts down the complexity of proving actual QoS properties during bottom-up integration of the implementation scenario. Such a scenario also ensures non-regression between successive implementations of a component.

Integration and QoS scenario. When several components are integrated, the integrator may consider assembling their interface scenario into a more complex one. With a technique similar to producing interface scenario for one component, the integrator produces an abstract scenario for the assembly. This is what we denote an integration scenario. Such a scenario is considered as an actual specification of the system. The objective is to prove actual QoS properties of the complete system from those of its components.

The Fig. 4 illustrates what kind of scenario is available at various steps of the design flow.

![Fig. 4: Scenario and Design Process](image)

The material required to manipulate scenarios is a debug-like interface and an automated synchronous product (used in section 5.1, 5.2, 5.3, 5.4). The debug-like interface is a symbolic SystemC debugger, used through debug commands. The variables of the SystemC components are assigned to symbolic values. Then, each time a branching condition is not decidable, the debugger asks the user for which branch to follow.

The synchronous product is an algorithm that takes as input a scenario for mastering the expected system behavior and one or several slaves, defined by (abstract) scenarios or (concrete) SystemC processes. When the compound behaviors of the slaves synchronize well
within the master, the algorithm succeeds and produces a more complex scenario embedding all the possible interleaving interactions between the components and their environment. If the components of the system never show the expected behavior modeled by the master, the synchronization algorithm fails and produces a debug scenario focused on the error [12]. Failure might also come from non-expected event trigger or assertion failure.

More generally, typical use of scenario during a design and implementation process falls into the following categories: specifying interfaces, verifying and validating implementations and finally validating and tuning system integration. Let us illustrate these categories for a component ‘C’ in a system ‘S’.

### Specifying interfaces
- Input and output protocols for data-exchange of ‘C’ are defined as protocol scenario.
- Expected behaviors are defined as use-case scenario. In addition to specifying ‘C’, such scenarios will further provide engineers with verification / validation material.

### Implementation verification
- The developer explores its implementation of ‘C’ with debug scenarios defined on-the-fly. The input protocol and use-cases might supply simple unit-tests to drive this exploration.
- By generalizing a bundle of debug scenarios, the developer exhibits several implementation scenarios.

### Implementation validation
The Synchronization Algorithm allows different kinds of validation of the implementation ‘C’ with respect to its expected and actual specification:
- Synchronizing the implementation scenario with the component implementation entails its consistency. Such automated synchronization can also be used for non-regression testing. For the verification/validation of ‘S’, the obtained interface scenarios replace the SystemC code of ‘C’ to break down the complexity.
- Synchronizing the protocol scenario with the component implementation ensures its conformance to the protocol.

### Verifying and validating system integration
- Synchronizing the different component interface scenarios of a system ‘S’ entails the system consistency and creates an integration scenario.
- Synchronizing the use-case scenarios of ‘S’ with the previous integration scenario ensures the correct behavior of the system over use-cases.

### Tuning system integration
- Assume now the integration scenario has been designed to include QoS annotations – for instance, time and power consumption evaluation. By synchronizing such an integration scenario with various environments or use-case scenarios, we statically compare different architectures dimensioning, such as buffer sizes, topologic schemas, clock rates, etc.
- We are also working on automated inference of QoS formulae over the architecture parameters. Thus, synchronization no more requires to instantiate those parameters with ground values. Rather, solving the formulae directly computes the optimal parameters.

### 5 Experiments on the FIFO
Now, let us illustrate the bottom-up methodology on the FIFO example. The engineer starts with scenarios by generating them through a dedicated debug interface. In our experimentation, we first focus on the producer. To simplify the illustration, we have inlined the SystemC code of the FIFO inside the code of the producer:

```systemc
while (true) {
    sc_time delay(rand(), SC_MS); // wait(delay);
    if (queue->num == MAX) // wait(queue->read_event);
    out->data[(queue->first+queue->num)%MAX] = rand(255)+1;
    ++queue->num; queue->write_event.notify();
};
```

Executing symbolically this code and setting some assumptions and assertions at precise times generates a scenario on a single trace. The first step reaches the control point 3. As the wait statement implies the possible execution of other threads, the engineer has to decide for queue->num == MAX. Let us answer with false, thus cutting out execution of 5 and looping on 3. As 4 is encountered multiple times – the engineer decides to reach it only 3 times – during the system inspection, the debugger automatically generates a new loop with a fresh counter k ∈ [0..3] in the debug scenario.

At this time, we do not consider the consumer behavior at all. Worst, let us assume its code is not available yet. Hence, the engineer must add several assumptions over the environment. Namely, the ‘last’ element in the FIFO shall remain unchanged until the producer filled the FIFO again, while the number of elements may have been decreased by some (not yet defined) consumer. We finally expect the producer to fill the FIFO with a single new element. Initially we assume the FIFO contains S elements. The resulting debug scenario follows:

```systemc
assume(queue->num == S && S <= MAX);
for (natural k = 0; k < 3; k = k+1) {
    int num = queue->num;
    int last = queue->first + queue->num;
    sync(producer.delay) {
        assume(queue->num != MAX);
        assume(queue->num <= num);
        assume(last == queue->first + queue->num);
        num = queue->num;
    };
    assert(num+1 == queue->num && queue->num <= MAX);
}
```

As 3 is encountered multiple times – the engineer decides to reach it only 3 times – during the system inspection, the debugger automatically generates a new loop with a fresh counter k ∈ [0..3] in the debug scenario.
5.1 Synchronizing a Scenario

To check the consistency of our debug scenario with the SystemC code of the producer and the FIFO, we run the synchronous product depicted at the end of section 4. Before exploring the synchronization product algorithm internals, let us outline its results.

Actually, the synchronization algorithm computes for each step of the scenario an over-approximation of the domain of each variable, inserted back to the original scenario inside comments. The algorithm also computes all the possible interactions of the system with its environment with respect to the scenario’s assumptions. Thus, new events and relations over variables are inferred and inserted in the original scenario with the notify and guarantee keywords. As a special effect, some assertions may be statically reduced, as illustrated below:

```plaintext
assume(queue->num == S & & S <= MAX);
for (natural k = 0; k < 3; k = k+1) {
    int num = queue->num; // num e [S,S+k]
    int last = queue->first + num; // last e [S,S+k]
 sync(producer.delay) {
    assume(queue->num != MAX & & ...); // queue->num e [S, MAX]
    num = queue->num;
    }
    guarantee queue->data[queue->num] = rand(255)+1;
    guarantee queue->num = num+1;
    notify(queue->write_event);
    assert(num+1 == queue->num); // true (can be reduced)
    assert(queue->num <= MAX); // true (can be reduced)
}
```

Let us remark that the synchronization algorithm builds an Interface Scenario that integrates all the observable events into the original scenario. The algorithm uses Abstract Interpretation fixpoint algorithms [13] with powerful relational domains combining formulae and linear relations [14]. These abstraction domains are infinite and they strictly extend the complete lattices traditionally used by Abstract Interpretation. This choice enforces strictly more general results than Model Checkers [12] that only address finite sequences of transitions, but requires the implementation of acceleration algorithms [15,16].

We will now explicitly take advantage of this extension to non-finite loops for generalizing our debug scenario. More precisely, we relax the constraint k<3 and replace it by k<N where N is a formal parameter for the scenario. Running again the synchronization algorithm, it finds a stable formula over k and N for each variable inside the loop. Acceleration algorithms also infer some formal relations between k and N. This allows for the computation of an upper bound k_{max} of the loop counter such that the assertions inside the loop remain true for all k e [0, k_{max}], as described in [17]. On our example, the obtained value for k_{max} is MAX-S in the case N e MAX-S.

5.2 Verification with Scenario

Coming back to the debugger, the analysis highlights a possible empty scenario. More precisely when k_{max} = MAX becomes greater than N, the following assumption is pointed out to eliminate all the possible executions of the system:

```plaintext
[...]
assume(queue->num != MAX); // queue->num = S+k
[...]
```

Recall this assumption was added to cut out the branch of the producer. This branch corresponds to a full FIFO blocked until some consumption occurs. We then transform the assumption by the specification of this behavior:

```plaintext
[...]
sync (producer.delay) {
    assume(queue->num <= num);
    num = queue->num;
    assume(last == queue->first + queue->num);
    if (num == MAX)
        sync(queue->read_event) { assume(queue->num < num); }
[...]
```

Remark we have now an Integration Scenario that models all the possible behaviors of the producer.

5.3 Connecting Component & System Scenarios

Let us suppose the user has now an Interface Scenario for both the producer and the consumer. These scenarios act as a specification for the components. Looking at the system connecting the consumer and the producer through the FIFO, both scenarios interleave.

For instance, consider the following consumer scenario:

```plaintext
Producer :=
repeat(N) { sync(0.1ms); sync(queue.write_event); }
Consumer :=
repeat(N) { sync(0.5ms); sync(queue.read_event); }
```

Here, the `repeat` statement is an abbreviation of a N-turns for-loop statements. If we assume N=MAX, the behavior of the global system will consist of four main stages. The synchronous product of the Producer and Consumer scenario automatically builds a new scenario for the scenario where these different stages appear:

```plaintext
Producer @ Consumer :=
repeat(k_1) { repeat(5) P ; Q } // 4k_1 <= MAX < 4(k_1+1)
repeat(k_2) { P } // 4k_1 + k_2 = MAX
repeat(k_3) { Q ; P } // 5k_1 + k_2 + k_3 = N
repeat(k_4) { Q } // MAX - k_4 = 0
```

The comments illustrate the inferred relations for the loop counters from the constraint queue.num <= MAX. Intuitively, these four stages can be illustrated by:

- (k_1) the FIFO is filled periodically w.r.t. the consumer and producer rates.
- (k_2) the producer continues until the FIFO is filled.
- (k_3) the consumer and the producer alternate while the FIFO remains full.
- (k_4) the consumer empties the FIFO.
Remark that with this scenario, the synchronization algorithm succeeds at proving the absence of any deadlock: consumer and producer are periodically blocked by each others, but the global behavior of the system is safe. We also have livelock freeness since each $(k_i)$ is bound by a definite value depending on $N$ and $\text{MAX}$.

## 5.4 QoS Formula on Scenario

To continue with the previous integration scenario, we now want to optimize the time for the $N$ elements of the producer to be transmitted through the FIFO. We modify the producer’s scenario as follows:

```plaintext
Producer :=
int t = time();
repeat(N) { sync(0.1ms); sync(queue.write_event); }
qos int delta = time() - t;
```

The local variable $\text{delta}$ receives the $\text{qos}$ attribute in order for the synchronization algorithms to infer relations over it. Running the algorithm with the integration scenario defined in 5.3, we obtain the following formula for $\text{delta}$:

$$\text{delta} = 0.5\text{ms} \cdot k_1 + 0.1\text{ms} \cdot k_2 + 0.5\text{ms} \cdot k_3$$

$$= 0.5\text{ms} \cdot (\text{N-MAX}) + 0.1\text{ms} \cdot (\text{MAX} \% 4)$$

Such a formula enables minimizing $\text{MAX}$ for $\text{delta}$ to be always less than some predefined contract.

## 4 Conclusion

Scenarios reveal to be a very interesting paradigm to deal with software engineering of complex systems. They come with powerful algorithms based on synchronous products. Concretized on a simple but expressive programming language, Scenarios enable to perform verifications at various levels of abstraction:

- local verifications of assertions within the Scenario;
- absence of simulation errors, deadlock and livelock under the scenario hypothesis;
- verification and optimization of QoS properties.

The verification with scenario can follow a bottom-up methodology such as assume/guarantee-based methods. The various forms of scenario conform to the various verification and validation requirements along the top-down methodology. The play-out approach in [3] is a potential interesting extension for generating an implementation scenario from use-case ones. Further work includes more experimentation with formula and making the scenarios also compatible with a top-down approach. This last approach requires a new refinement algorithm to be defined and proven over scenarios, making them addressing the complete development flow.

**References:**