Extracting Logical Formulae that Capture the Functionality of SystemC Designs

Nicolas Vallée, Bruno Monsuez, Vladimir-Alexandru Paun

***Abstract*—Object-oriented hardware design languages like SystemC have become very popular to co-design hardware and software systems. Such designs are classically translated into a transition system in order to verify a specification with model-checkers. However, compositionnality and parametricity of SystemC components complicate their translations into finite transition systems. Processing analysis of high-level designs occurs early in the design flow and aims to greatly reduce the correction costs of eventual errors. In this paper, we propose a formal method to statically analyze SystemC designs. Our approach consists in extracting a logical formula representing the behavior of the system in order to avoid combinatorial explosion. This method combines a symbolic execution of SystemC code to infer logical formulae representing its behavior and a generalization phase of these inferred logical properties.**

Keywords : SystemC, hypergraphs, symbolic execution, abstract interpretation

I. Introduction

Modern embedded systems are growing in complexity rendering the use of high level *Hardware Description Languages* (HDL) mandatory. The increasing sophistication of the design spirits the use of tools that support parametricity and compositionality of *Intellectual Properties* (IP). SystemC is becoming a de facto standard in the design of embedded systems by meeting all these requirements. The blend of HDL with the C++ programming language, empowers SystemC with parametricity and compositionality through the template engine and the object-oriented paradigm respectively.

SystemC has been conceived for system co-designs and simulations at a higher level of abstraction, that should imply a high-level system verification. Nonetheless, formal verification of SystemC has still a long way to go, given the fact that its object-oriented nature and its scheduler father a sophisticated event driven simulation semantics.

Transactional Level Modeling [1] corresponds to an industry-proven approach in order to raise the level of abstraction to specify and model SoC design. Starting from this abstraction level, our approach consists in extracting functional properties of the design. We consider that a relevant analysis technique for SystemC components designed using TLM *(Transactional Level Model)* must throughly comply with several demands [2], [3]:

- be sufficiently compositional;
- minimize the information loss despite the different abstraction levels;
- support refinement.

By compositionality [4] we understand that the global property can be established from composing local properties as the system-level analysis is the composition of component-level analyses. Traditionally, designs of embedded systems

All authors are with the Laboratory of Electronic and Computer Engineering of ENSTA, Paris, France. e-mail: firstname.lastname@ensta-paristech.fr

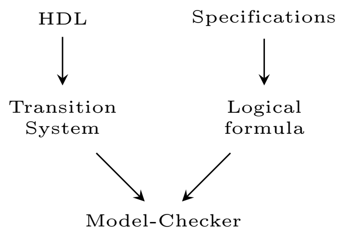

Fig. 1. Classical way to verify HDL designs

are statically verified according to the scheme in Figure 1. HDL design is modeled into a transition system, while the expected specification is translated into temporal logical properties. Thereafter, the results of the transformations are given as argument to a model-checker that makes sure the transition system abides all the expected properties, lending otherwise an execution path representing a counter-example. The problems in synthesizing a transition system from the HDL design, in the context of SystemC, is depicted by the component interdependencies generating the combinatorial explosion of the state space.
Our approach consists in extracting a logical formula representing the behavior of the system so as to avoid combinatorial explosion. We may consequently proceed to convert this logical formula into an ad-hoc representation exploitable by a tool that verifies if this behavior fulfills the specification. To the best of our knowledge, we introduce a novel method to compute this logical formula.

A. Expected analysis

We aim to extract a logical formula that describes the functional behavior of a SystemC component. Considering the member function `clear_locks` of the `simple_bus` class, as pictured in Figure 2, we would like to extract a functional behavior represented by a logical formula, that is to be manipulated with model-checkers or theorem provers so as to check whether a specification is verified.

```
void simple_bus::clear_locks()
{
  for (int i = 0; i < m_requests.size() // test T1
  ; ++i)
    if (m_requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
        // test T2
      m_requests[i]->lock = SIMPLE_BUS_LOCK_SET;
      // instruction block S1
    else
      m_requests[i]->lock = SIMPLE_BUS_LOCK_NO;
      // instruction block S2
}
```

Fig. 2. simple_bus::clear_clocks

In the Figure 2, the guard of the `for` loop depends on the results of a `m_requests.size()` method call. It should thus be considered as a symbolic parameter of the analysis. Unfortunately, the `for` loop cannot be statically unrolled, leading to a combinatorial explosion of the state space. Hopefully, some techniques to generate an invariant of this loop, could subsequently be used. This invariant will be used as a meta-state of the transition system, *i.e.* a state that represents all the states associated to the loop.

The functional behavior of the function in the Figure 2 can be expressed through a first order logical formula:

$$\forall o, [\text{class}(o) = \text{simple\_bus} \Rightarrow \forall r \in \text{o.m\_requests}, \\ (r.\text{lock}_{initial} = \text{granted} \Rightarrow r.\text{lock}_{final} = \text{set}) \\ \vee (r.\text{lock}_{initial} \neq \text{granted} \Rightarrow r.\text{lock}_{final} = \text{no})]$$

where:

- *granted* = `SIMPLE_BUS_LOCK_GRANTED`
- *set* = `SIMPLE_BUS_LOCK_SET`
- *no* = `SIMPLE_BUS_LOCK_NO`

This logic formula is interpreted as:

for all instances o of the *simple_bus* class, for all requests r contained by $o.m\_requests$, if the $lock$ of r is $granted$ then the $lock$ of r becomes set; otherwise the $lock$ of r becomes $unset$.

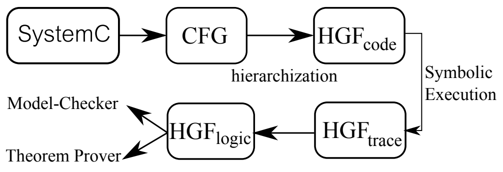

Fig. 3. Analysis steps

The first step of the analysis consists in generating the control flow graph CFG, by parsing the SystemC code. Once the CFG is generated we construct the fractal hypergraph HGF_{code}, a hierarchical model created in a statical way depending on the characteristics of the language, see Figure 4. We thereafter use a trace based semantics to statically build the HGF_{trace} hypergraph, a family of hypergraphs that abstracts the execution traces generated when the HGF_{code} hypergraphs are executed. This step may be seen as a generalized form of symbolic execution. Classically the symbolic execution not only constructs the execution traces, but also adds some extra information in the form of a Path Condition, PC. The logical informations contained in the PC will be the ground of the behaviour extraction from the traces, in the form of logical formulae. Thus the logical fractal hypergraph HGF_{logic} is created.

The functional information are to be transformed through the generalization phase in order to be consequently exploited by specification validation tools.

B. Contributions:

In this paper, we propose an analysis methodology, for TLM-designed SystemC components, that manages to be compositional and that tries to limit the information loss. The main point in our article consists in the generalization of logical formulae representing the component behaviour.

The paper is organized as follows. Section II describes the related works; Section III presents the mathematical model of fractal hypergraphs and a formal method called symbolic execution; Section IV describes how to represent logical formulae with fractal hypergraphs; Section V describes how to convert a fractal hypergraph describing the implementation of a SystemC component into a fractal hypergraph representing a logical formula that describes the behaviour of this SystemC component; Section VI finally describes the sizeable design we have analyzed: the SystemC Simple Bus, and it consequently shows the result of our analysis.

II. Related works

SystemC modelizations and analysis are various. First there are a few semantics for SystemC, that approximately respect the real behavior of its scheduler [5], [6], but they are especially useful for its simulation.

Dynamic verifications with monitoring [5] are also used, whose execution traces, among other elements, could be statically analyzed. For instance, Vardi et al. create an extendable framework for temporal logics [7], yet it might miss some details due to its big-step semantics, therefor excluding soundness.Furthermore, there is an approach based on *symbolic model-checking* that captures the reactive features of SystemC [8], [9], enabling the analysis of component communication, transactional memory, etc.

Another approach, that is closer to our work, consists in formalizing a subset of SystemC, called SystemC$^{\mathbb{FL}}$, by using *Algebra of Communication Processes* (ACP) and *A Timed Process Algebra for Specifying Real-Time Systems* (ATP) [10]. SystemC designs are then translated into an adhoc representation for model-checkers.

III. Preliminary notions

A. Fractal hypergraphs

In [12], we proposed a new mathematical structure that is well suited to capture and represent the object-oriented nature of SystemC components. Hypergraphs [13] are a mathematical extension of graphs. A hypergraph is a graph whose hyperedges may connect two or more vertices. Fractal hypergraphs as presented in [12] is a special category of hypergraphs where the hyperedges between the vertices may be defined by fractal hypergraphs.

We proposed in [12] a formal representation of SystemC components based on fractal hypergraphs. For instance, the fractal that corresponds to the function given in the Figure 2 is represented in the Figure 4. Nodes represent all the separators of the system description source. Arrows (hyperedges) represent either instructions, or transitions for the sequence of declaration blocks. Frames represent a hierarchical level in fractal hypergraphs, that may embed a static scope, a function body, a block of instructions, etc.

Finally, we also proposed in [12] a formal semantics of SystemC based on the transformation of the fractal hypergraphs. The proposed semantics is a trace based semantics that can be used to perform abstract interpretation or symbolic execution. A very nice feature is that the execution traces are also represented by fractal hypergraphs.

In this paper, we use the formal model as well as the formal semantics of SystemC based on fractal hypergraphs.

This model is used at the beginning of our analysis in order to translate the CFG into a hierarchical model.

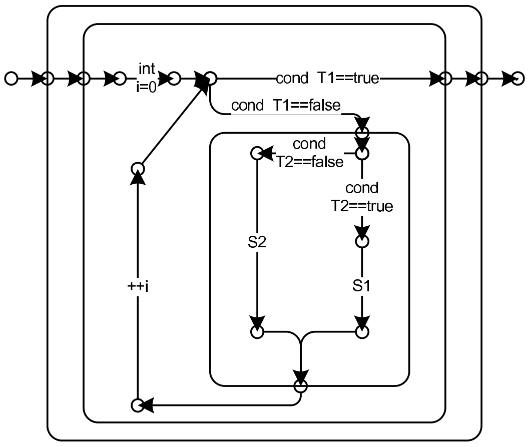

Fig. 4. Fractal hypergraph of simple_bus::clear_clocks

B. Abstract interpretation

All possible behaviors of a system S can be defined by a transition system F_S. Properties of the system S are analyzed when studying F_S. Unfortunately it may be too complex to handle. Abstraction can construct a computable over-approximation of F_S. Abstract interpretation [14], [11] assumes that semantics can be expressed as fixpoints of monotic functions in partially ordered domains. It classically works in an abstract domain related to a concrete domain through a Galois connection.

Definition 1 (Galois connection): Given two partially ordered sets $\langle C, \leq \rangle$ and $\langle \bar{C}, \sqsubseteq \rangle$, two monotone functions $\alpha : C \to \bar{C}$ and $\gamma : \bar{C} \to C$ provide a Galois connection iff:

$$\forall X \in C, \forall \bar{X} \in \bar{C}, \alpha(X) \sqsubseteq \bar{X} \Leftrightarrow X \leq \gamma(\bar{X})$$

denoted $\langle C, \leq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{C}, \sqsubseteq \rangle$

Given a transition system F in the concrete domain, a transition system $F_\sharp$ defines a *sound abstraction* for F with respect to the Galois connection $\langle C, \leq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{C}, \sqsubseteq \rangle$ iff:

$$\forall \bar{X} \in \bar{C}, (\gamma \circ F_\sharp)(\bar{X}) \leq (F \circ \gamma)(\bar{X})$$

Among all transition systems, we only consider the transfer function describing the symbolic execution. It corresponds to the concrete domain of the set of execution traces partially ordered by inclusion $\langle 2^\Sigma, \subseteq \rangle$. The soundness of the analysis is implied by the fact that we always keep an over-approximation of the initial state, *i.e.* all the possible behaviors are still in the abstract state.

C. Symbolic execution as Abstract Interpretation

We choose to present the symbolic execution as a transfer function in the framework of Abstract Interpretation. The use of an operational semantics enables us to construct execution traces. As oposed to a normal simulation, the symbolic execution represents all the parameters as symbolic values. Therefore we construct execution traces throughout all the possible execution paths, in a compact manner with the help of symbols and the Path Conditions that represent constraints on an abstract path. As described in [12], there exists an operational semantics of SystemC based on fractal hypergraphs. High level features as templates and class are managed. In this section we will only focus on the execution rules in order to ensure the safety of symbolic execution.

For the assignment, the new symbolic expression of variables consist just in updating the abstract environment. Conditional branchings present two cases. In the first case we can statically determine the value of the conditional test. Symbolic execution knows which branching is to be taken. We can also simplify the trace fractal hypergraph in order just to represent this path – see Figure 5.

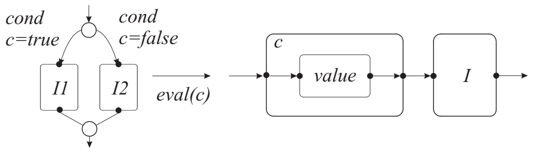

Fig. 5. Traversal of a dispatch point – the original fractal hypergraph is modified: all the unused branchings are deleted. the final fractal hypergraph also goes on representing the execution trace.

Proposition 1: The soundness of symbolic execution of statically determined conditional branchings is verified.

$$\gamma \circ F_{S\sharp} = S.(cond = true) = F_S \circ \gamma$$

In the second case, the conditional entry cannot be statically resolved. Symbolic execution has to build also a disjunction of all possible branchings – see Figure 6.

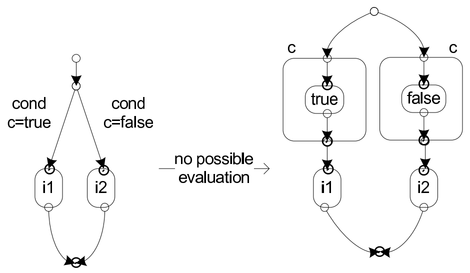

Fig. 6. Traversal of a dispatch point – the original fractal hypergraph is modified: no condition value can be determined, so no branching is unused.

Proposition 2: The soundness of symbolic execution of conditional branchings is verified.

$$F \circ \gamma(S) \subseteq \gamma \circ F_\sharp(S) = S.(cond = true) \cup S.(cond = false)$$

Conditional loops

When entering the loop block, the frame that denotes the condition gets replaced by the logical expression denoting the condition being evaluated. If the condition evaluates to "false", the exit edge is active, in which case we simply have a returning transition to the upper level. If the condition evaluates to "true", the transformation continues with the frame b that denotes the body of the loop. When the transformation reaches the exit node of the frame b, the loop is unrolled to pursue the transformation – see Figure 7.

Proposition 3: The soundness of symbolic execution of conditional loops is verified.

$$F \circ \gamma(S) \subseteq \gamma \circ F_\sharp(S).$$

Handling loops implies symbolically iterating the loop body. However, the lack of information about the iteration number or having to many iterations may render it impossible. To solve this problem we have to find a sound

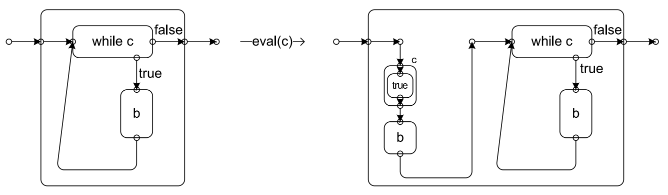

Fig. 7. Conditional loop unrolling

overapproximation of all execution traces of this loop. This generalization will be described more precisely in Section V-A.

D. Symbolic Execution and Fractal Hypergraphs

The third step of our analysis consists in the transformation of the code fractal hypergraph into a trace fractal hypergraph by means of symbolic execution.

Fractal hypergraphs can represent the SystemC components as well as the execution traces generated by the simulation of SystemC components.

Definition 2 (System state): A *system state* consists in:

- t : the current time ;
- $\mathscr{H}_e$: the hypergraph that denotes the history of the system execution ;
- $\mathscr{H}_s$: the hypergraph representing the whole system.

We define a transition function that maps a system state to another system state.

A trace-based semantics [15], [16] manipulates paths representing the execution traces $\mathscr{H}_e$. The traces contains the complete history of the symbolic execution or more generally the complete history of any static analysis ; it totally defines the current context. Such a trace based semantics has been presented in [12].

IV. Representing logical formulae with fractal hypergraphs

In order to extract logical property when statically analysing a component defined in SystemC, we want to extend the *system state* with a set of additional logical formulae that completely describes the behavior of the components.

Definition 3 (Extended system state): A *extended system state* is represented by :

- t : the current time ;
- $\mathscr{H}_e$: the hypergraph that denotes the history of the system execution ;
- $\mathscr{H}_l$: the hypergraph that denotes the inferred logical formulae ;
- $\mathscr{H}_s$: the hypergraph representing the whole system.

We decide to represent the logical formulae using fractal hypergraph. Further on, we show how we can express first-order logical formulae using fractal hypergraphs.

Definition 4 (Logical formula): Considering a programming language $\mathscr{L}$, the set of all possible identifiers $\mathscr{A}$, the set of variables $\chi = \{x; x_i | x \in \mathscr{A} \wedge i \in \mathbb{N}\}$. We define:

- constants: $\top$ defines the boolean `true` and $\bot$ defines the boolean `false`;
- logical atoms: polynomials on χ with coefficients in $\mathbb{Z}$ that are constrained by a comparison with 0.

$$\mathbb{Z}[\chi] \diamond 0 \text{ where } \diamond \in \{=, <, >, \leq, \geq\}$$

- terms: constants or logical atoms;
- predicates: terms or logical connections $P \wedge Q$, $P \vee Q$ or $\neg P$ such that P and Q are predicates;
- logical propositions: predicates or logical quantifications $\forall x, P$ or $\exists x, P$ such that P is a logical proposition and that x is a variable.

A. Terms

Constant atoms are defined as simple transitions. Logical *atoms* are also expressed by encapsulating the atoms into a frame. The variables defined in $\mathbb{Z}[\chi]$ are present in the environment associated to the fractal hypergraph that denotes the execution.

Native values and constraints on atoms are associated to fractal hypergraphs through the environment that is shared among the hypergraph $\mathscr{H}_e$ representing the execution paths and the hypergraph $\mathscr{H}_l$ representing the logical formulae.

B. Predicates

Conjunctions and disjunctions

A conjunction of two logical terms means that verifying this conjunction implies verifying both logical terms. A sequence of logical hyperedges defines the conjunction of the logical terms associated to these hyperedges – see Figure 8.

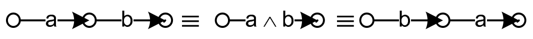

Fig. 8. Conjunction of logical terms

A disjunction of two logical terms means that verifying this disjunction implies verifying at least one logical term. Verifying at least one logical term is translated into two parallel transitions. Two logical hyperedges, that have the same origin and destination, are also used to define the disjunction of the logical terms associated to these hyperedges – see Figure 9.

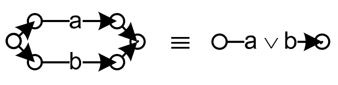

Fig. 9. Disjunction of logical terms

Negations

A logical negation is transformed by applying De Morgan's rules only to the concerned terms and not to more complex predicates. We then use the following rule, in order to obtain a disjunctive logical fractal hypergraph:

$$\forall a \in \mathbb{Z}[\chi], \forall [i; j] \in \mathbb{I},$$
$$[\ \neg(a \in [i; j]) \Leftrightarrow a \in [-\infty; i-1] \vee a \in [j+1; +\infty]\ ]$$

C. Logical propositions

Logical propositions associate a logical term to a set of free variables, similarly to functions in programming language. Logical predicates are also represented by a hyperedge embedding a fractal hypergraph that represents its parameters and the logical formula. Each parameter of the logical predicate is represented by an empty hyperedge – see Figure 10.

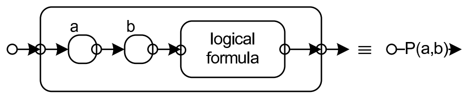

Fig. 10. Logical proposition

D. Universal and existential quantifications

A formula can be verified for a set of values. We can abstract the set of all formulae verified for each value to a formula that denotes the set of all formulae. This transformation creates a first-order abstraction, called a $\forall - term$ – see Figure 11.

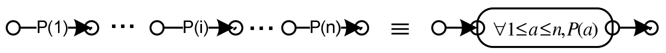

Fig. 11. Universal quantifier

A formula may be verified for at least one value in a given set. We can abstract this formula to a first-order abstraction, called an $\exists - term$ – see Figure 12. Such a quantification ensures that a property is verified for at least one element of a container, such that a find function does not fail.

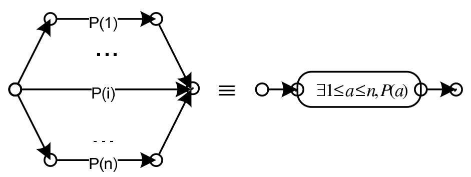

Fig. 12. Existantial quantifier

E. The place of logical fractal hypergraphs in our formal debugger

A SystemC design models the behavior of a circuit. However, it corresponds to an executable description of this circuit, that may describe irrelevant implementation details. Another way to describe its behavior is by inferring relations between the inputs and outputs as well as the internal states of this circuit.

Our formal debugger [3] semi-automatically extracts logical fractal hypergraphs from trace fractal hypergraphs in order to represent an abstract behavior of the corresponding system. Execution traces have been produced by symbolic execution of the concrete semantics as detailed in [12]. Those execution traces are then iteratively abstracted to logical fractal hypergraphs that represent relations between the initial states and the final states of the execution traces.

V. Converting a trace fractal hypergraph into a logical fractal hypergraph

The last step of our analysis consists in the behaviour extraction, from the traces annotated with path conditions, that were previously generated by symbolic execution, in the form of logical formulae.

This section presents the relevant transformations to extract a logical fractal hypergraph from a function body. We then show how the basic logical fractal hypergraphs are aggregated when calling sub-methods.

A. Intraprocedural analysis

Variable declarations, variable assignments, conditional branchings and loops qualify as intraprocedural actions. Variable declarations and assignments are managed by the environment while conditional branchings are converted through symbolic execution. Handling loops is more difficult: we test if the entry condition is verified, in which case we unroll some iterations. A precise result cannot always be guaranteed with this method therefore we often have to proceed to a generalization.

Conditional branchings

The `if-then-else` conditional branching is a simple pattern - the *dispatch point*. A hyperedge labelled by an entry condition precedes all possible paths. When entering one of those paths, the trace-based semantics must record the branching condition.

If the conditional entry can be statically resolved, the symbolic execution knows which branching is to be taken. In this case, we can simplify the fractal hypergraph in order to represent the logical formula:

$$condition = true \ \wedge \ branch\_formula$$

The $cond = false$ case is symmetrical so we do not describe it.

If the conditional entry cannot be statically resolved, the symbolic execution has to build a disjunction of the logical formulae of all possible branchings:

$$\begin{array}{c} cond = true \ \wedge \ branch_{true}\_formula \vee \\ cond = false \ \wedge \ branch_{false}\_formula \end{array}$$

Application to simple_bus:clear_locks

With previous transformations, we can now convert the loop body of the method `simple_bus::clear_:locks`. Its associated logical fractal hypergraph represents the following logical formula (where $SB = SIMPLE\_BUS$):

$$\begin{array}{c} m\_requests[i] \rightarrow lock^{(n)} = SB\_LOCK\_GRANTED \wedge \\ m\_requests[i] \rightarrow lock^{(n+1)} = SB\_LOCK\_SET \\ \vee \\ \neg(m\_requests[i] \rightarrow lock^{(n)} = SB\_LOCK\_GRANTED) \wedge \\ m\_requests[i] \rightarrow lock^{(n+1)} = SB\_LOCK\_NO \end{array}$$

Conditional loops

Handling loops may imply symbolically iterating the loop body. However, the lack of information about the iteration number or having to many iterations may render the unrolling of the loop impossible. To solve this problem we unroll for a fixed number of iterations all the possible execution paths and we then proceed to a generalization in order to infer an abstract formula that abstracts all the possible situations.

Generalization

When symbolic execution may not be able to statically determine a logical formula or when it may determine too complex formulae, we must try to generalize the inferred logical formulae.

To illustrate the generalization step, an additive block is analyzed – see Figure 13.

The generalization step consists in finding an invariant in order to represent loop iterations, *i.e.* a set of execution traces

```
int fact(int n)
{
    int res = 1;
    int i;
    for (i=0 ; i<n ; i++)
       res += 1;
    return res;
}
```

Fig. 13.   Additive block

will be collected into a logical formulae representing the invariant of the loop. At this moment we leave the area of second order analysis (path based) for the area of first order analysis (state based), as described in [16].

We use classical abstract interpretation techniques, such as widenings or convergence accelerators [17], to generate a relation between all the variables used inside the loop body and the loop condition as well as the loop counter. For instance, if we choose the numerical abstract domain of integer polyhedra [18], the behaviour of the loop presented in Figure 13 can be summed up into the following invariant:

$$res = 1 + counter \ \wedge \ i = counter$$

Note: In this paper, the previous exposed techniques are enough to limit the loss of precision. However more advanced abstraction techniques, as described in [19], can manage more complicated designs.

B. Managing method calls

In this section we describe how we handle sub-programs[1] in a compositional way. During the analysis, each time we encouter a method call two cases are considered. If the method M has never been analyzed we then extract a logical fractal hypergraph H_M associated to the *out of context* behaviour of M. Otherwise we specialize H_M by adding the constraints on the input parameters of M, depending on the context call.

As stated in Section III-C, the soundness of the analysis of method calls relies on the soundness of symbolic execution.

Remark: The method body may contain loop instructions. As a consequence the extraction of the out of context behaviour may be too coarse due to the generalization step (see Section V-A for more details). In that case, our strategy consists in analyzing the function calls by inlining the method body in order to reduce the loss of precision.

VI. Experimental results

A. SystemC design of Simple Bus

We will analyze the SystemC Simple Bus that is available in the SystemC standard library. It is a basic component described at transactional level. Simple Bus is a synchronous bus – *i.e.* all subcomponents are synchronized through a unique clock. There is no limit for concurrent connection number. An *arbiter* manages priority and access grants to a unique provider. Each interval of message addresses is mapped to a unique consumer.

[1]We do not consider recursive methods since they cannot occur in electronic designs. If we want to handle mutual recursive functions, we can use a generalization in a similar way as when handling loops.

There are two available modes:

- a blocking mode: the provider calls the method `write_burst`. It then waits until the message is read. In the same way, the consumer waits for a message when `read_burst` is called.
- a non-blocking mode: the provider calls the method `write` and immediately returns. Likewise, the consumer does not wait when no message is available – by using the method `read`.

B. Abstract behavior of Simple Bus

Scalability of our analysis has been tested on the whole `Simple Bus`. In Figure 15, you can see a logical formula describing the behaviour of the method `simple_bus_arbiter::arbitrate`. In this method, there are conditional branchings, loops and method calls. Notice that we have deleted all verbose mode lines.

```
simple_bus_request *
simple_bus_arbiter::arbitrate(const
       simple_bus_request_vec &requests)
{
   int i;
   simple_bus_request *best_request = requests[0];

   for (i = 0; i < requests.size(); ++i) // selection P1
   {
      simple_bus_request *request = requests[i];
      if ((request->status == SIMPLE_BUS_WAIT) &&
          (request->lock == SIMPLE_BUS_LOCK_SET))
         return request;
   }

   for (i = 0; i < requests.size(); ++i) // selection P2
      if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
         return requests[i];

   for (i = 1; i < requests.size(); ++i) // selection P3
   {
      sc_assert(requests[i]->priority != best_request->
          priority);
      if (requests[i]->priority < best_request->priority)
         best_request = requests[i];
   }

   if (best_request->lock != SIMPLE_BUS_LOCK_NO)
    best_request->lock = SIMPLE_BUS_LOCK_GRANTED;

   return best_request;
}
```

Fig. 14.   simple_bus_arbiter::arbitrate: code

The method `simple_bus_arbiter::arbitrate` takes the list of all requests and selects the best request if possible – *i.e. if an adequate request exists*. To choose the request we browse the list of all requests looking for the first request that verifies a given condition. If any request is found, the method returns an error.

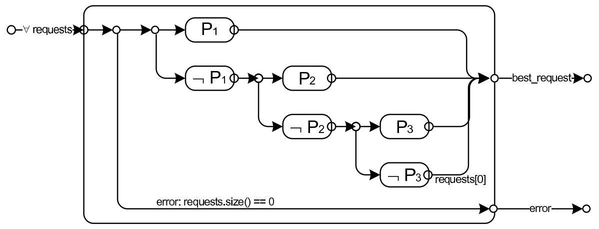

Fig. 15.   simple_bus_arbiter::arbitrate: logical fractal hypergraph

C. Interpretation of the experimental results

Our process extracted the logical fractal hypergraph described in Figure 15. The three selective browsings used to select the best request have been found and are represented by the logical formulae P_1, P_2 and P_3.

$test_1(req) : req \rightarrow status = SB\_WAIT \wedge req \rightarrow lock = SB\_LOCK\_SET$
$test_1$ verifies if the given request is currently waiting for being executed and whether its lock is set.
$test_2(req) : req \rightarrow lock = SB\_LOCK\_GRANTED$
$test_2$ verifies whether the lock of the given request has been granted.
$P_1 : \exists i, \forall j, 0 \leq i \leq requests.size() \wedge test_1(requests[i]) \wedge 0 \leq j \leq i \wedge \neg test_1(requests[j])$
P_1 selects the request that verifies the $test_1$ and that has the lowest index.
$P_2 : \exists i, \forall j, 0 \leq i \leq requests.size() \wedge test_2(requests[i]) \wedge 0 \leq j \leq i \wedge \neg test_2(requests[j])$
P_2 selects the request that verifies the $test_2$ and that has the lowest index.
$P_3 : \exists i, \forall j, 0 \leq i \leq requests.size() \wedge 0 \leq j \leq requests.size()$
$\wedge\, i == j \vee$
$requests[i] \rightarrow priority < requests[j] \rightarrow priority \wedge$
$requests[i] \rightarrow lock \neq SB\_LOCK\_NO \wedge requests[i] \rightarrow lock = SB\_LOCK\_GRANTED$
P_3 selects the request that has the lowest priority and grants its lock if it was unset.

The structure of the fractal hypergraph shows the successive use of these selective browings. If P_1 finds any request, the method returns the respective request. Otherwise, P_2 is used in a similar way, then P_3. If none of these three selective browsing finds any request, the method returns the default choice, the first element of the request collection. At the beginning of the method, we test whether this collection is empty when selecting its first element. If this selection fails, the method fails and returns an error – represented by a special exit hypernode.

VII. Conclusion

In this paper, we propose an analysis techniques for TLM-designed SystemC components that manages to be compositional and intends to reduce the information loss of abstractions. This method combines a symbolic execution of the SystemC code to infer logical formulae representing its behavior and a generalization phase of the inferred logical properties in order to manage loops, parametric code, function abstraction, etc.

The extraction process does not only make consecutive steps but also proceeds to on-the-fly generalization of the inferred formulae. Inferred logical properties representing a sound abstraction of the system behavior are still general enough to be translated into an ad-hoc representation for a wide range of tools in order to verify whether it respects a specification. These tools can be either classical model-checkers, or theorem provers.

Although we only analyze functional properties with our analysis methodology, we hope to use it for non-functional properties as estimating *Worst Case Execution Times*. Futhermore, we will define a general framework for the generalization techniques that could be used in our analysis.

Acknowledgements

The authors would like to thank Alexandre Chapoutot for his helpful comments in improving the quality of this paper.

References

[1] F. Ghenassia, *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[2] M. Y. Vardi, "Formal techniques for systemc verification; position paper," in *DAC*, 2007, pp. 188–192.

[3] B. Monsuez, F. Védrine, and N. Vallée, "On the design of a formal debugger for system architecture," in *ICC'08: Proceedings of the 12th WSEAS international conference on Circuits*. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 462–467.

[4] P. Cousot and R. Cousot, "Modular static program analysis, invited paper," April 6—14 2002.

[5] W. M. Lab, W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl, "The simulation semantics of systemc," in *In Proc. of DATE 2001. IEEE CS*. Press, 2001, pp. 64–70.

[6] B. Monsuez, Y. ZHANG, and F. Védrine, "Systemc waiting-state automata," in *VECoS'07*, Algiers, Algeria, May 2007.

[7] D. Tabakov, M. Y. Vardi, G. Kamhi, and E. Singerman, "A temporal language for systemc," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 22:1–22:9.

[8] R. Shyamasundar, F. Doucet, R. Gupta, and I. Krger, "Compositional reactive semantics of systemc and verification with rulebase," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and P. Sampath, Eds. Springer Netherlands, 2007, pp. 227–243.

[9] F. Doucet, R. Shyamasundar, I. Krüger, S. Joshi, and R. Gupta, "Reactivity in systemc transaction-level models," in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science, K. Yorav, Ed. Springer Berlin / Heidelberg, 2008, vol. 4899, pp. 34–50.

[10] K. L. Man, A. Fedeli, M. Mercaldi, M. Boubekeur, and M. Schellekens, "Sc2scfl: automated systemc to systemcfltranslation," in *Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation*, ser. SAMOS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 34–45.

[11] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[12] N. Vallée and B. Monsuez, "A formal model of systemc components using fractal hypergraphs," in *Design, Analysis and Tools for Integrated Circuits and Systems – International MultiConference of Engineers and Computer Scientists*, 2010.

[13] C. Berge, *Graphs and Hypergraphs*. Elsevier Science Ltd, 1985.

[14] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130.

[15] C. Colby and P. Lee, "Trace-based program analysis," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 195–207.

[16] D. A. Schmidt, "Trace-based abstract interpretation of operational semantics," *Lisp Symb. Comput.*, vol. 10, no. 3, pp. 237–271, 1998.

[17] L. Gonnord, N. Halbwachs, and G. France, "Combining widening and acceleration in linear relation analysis," in *In SAS*, 2006, pp. 144–160.

[18] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '78. New York, NY, USA: ACM, 1978, pp. 84–96.

[19] F. Védrine, "Binding-time analysis and strictness analysis by abstract interpretation," in *SAS '95: Proceedings of the Second International Symposium on Static Analysis*. London, UK: Springer-Verlag, 1995, pp. 400–417.