How an ”Incoherent Behavior” inside generic hardware component characterizes functional errors

Bruno Monsuez
ENSTA - UEI
bruno.monsuez@ensta.fr

Franck Védrine
CEA, LIST, LMEASI
franck.vedrine@cea.fr

Micaela Mayero
Université Paris 13 - LIPN - LCR
micaela.mayero@lipn.univ-paris13.fr

Nicolas Vallée
ENSTA - UEI
nicolas.vallee@ensta.fr

Abstract: Detecting functional errors on generic hardware components is often a complex task. This task becomes more complex in a componentwise approach when analyzing components without their embedded context that is the entire system description. In this paper, we propose a methodology that successfully detects just from the component's description a pure functional error that neither extensive tests nor formal methods could find.

Key–Words: static analysis, hardware functional error, logical formulae, formal inference

1 Introduction

Conception and design of components are changing. Keywords are now reusability, genericity, scalability and modularity as well as time to market.

Perfect tools and hardware description languages should allow expressing generic components, allow efficient specialization and multiple levels of abstraction, support the assemblies of modular components. In addition, these tools should provide fast and efficient simulation, an insurance that the combined components can work together. In a really perfect world, each component should be individually certified and the assembly process should guarantee that the final instantiated component is efficient and error-free.

1.1 New challenges for formal verification

The dream of any SoC designer would be to tailor the complexity, the offered functionalities as well as the performance of the component by specializing generic components (1), and assembling them (2).

Part of the challenges are addressed by using new Hardware Design Language like SystemC, CowareC, SystemVerilog that address multiple abstraction levels, modularity and genericity. However a new HDL will solve some problems and introduces new ones. If we concentrate on verification, the first challenge is to analyze a single component before being assembled. To make verification more difficult, formal parameters are used to describe for instance the size of the bus, the endianness of data, the cache associative memory. The second challenge consists in analyzing the component with respect to those formal parameters. The third and last challenge is to be able to combine the results obtained by each single analysis and refine the inferred properties when embedded in a bigger system. We call this approach ”*debug as design*” since the debug process occurs during all the design process. By contrast, we call the more classical approach ”*design and debug*” since the debug process occurs after the design having been completed.

1.2 Context of the conducted case-study

In year 2001, the SystemC standard was emerging. SystemC looks very different from HDL languages like Verilog or VHDL introducing all the complex and powerful expressivity of C++ to HDL. Obviously hardware verification of SystemC developed components could not be done with the tools that were available at this time. And tools for debugging SystemC design are 7 years later still missing. Some of the authors had good expertise in hardware and software verification, and were working on technologies to analyze C++ components. The idea of reusing the developed techniques had gained and a collaboration with STMicroelectronics began, to evaluate if it were meaningful with respect to hardware verification.

The first step was to evaluate the technologies we developed. We first verify components that connect to a bus. The components have been designed using SystemC. The main work was to first use our techniques to verify each component individually. We then connect the components to the bus and terminates the verification of the complete system. The components were designed at BACA abstraction level. This case study was conducted to verify the viability of the ”*debug as design*” approach and to gain experience in SystemC verification before going into the implementation of a full-featured abstract debugger[2].

We chose a novel and rich analysis framework

where components are described by automata and where the action on the arrows of the automata are automata as well. Within this framework, we are able to combine abstract interpretation, type inference and theorem proving like techniques. We hope to be able in the near future to add support for model checking as well as for advanced theorem proving.

1.3 Errors detected during this case-study

The case-study has convinced us from the viability of the presented approach before starting with the full implementation of the formal debugger (1) as well as to determine which kind of errors could be detected during each phase of the formal analysis (2).

Despite having a good expertise in software and hardware verification, we thought that the first phase of our analysis – a formal verification of a single component without the context – may only detect local design errors or basic local violation of specifications.

If we verified that protocol violation introduced in the code are correctly detected, we were however really surprised that the methodology we used make it possible to detect ”incoherent behaviors” of individual components and to be able to classify definitely those ”incoherent behaviors” as ”functional system errors”.

We do not hear about any other approach that is able to proceed with the analysis of small component like a size-converter and is able to detect that there is no consistency in the way the messages get sent or received, despite the fact that all the single actions are perfectly valid and do not violate the properties or constraints that the single component should verify.

We suppose that at least one of those errors was introduced by an hazardous copy-paste in the SystemC code that were obtained by retro engineering the RTL. However, this error was never detected by the tests that were conducted during the validation of the platform written in SystemC.

We think that the gained results are not specific to our current formal debugger and that the approach could be also be used by other analyzers (resp. formal verification techniques) to detect functional errors.

In this paper, we first characterize one of those functional errors that have been detected by the presence of an incoherent behavior. We then give a way of detecting and correcting such errors. We finally end the presentation with some words on the formal debugger we are developping and how such a tool can help non-expert users to validate their components.

2 Functional error in size-converter

Detecting functional errors is difficult by using formal verification techniques. The most obvious way is to write a functional specification of the system and to verify that the implemented system verifies it. However, since the functional specification is quite redundant with the implemented code, it may also contain the same error. Therefore, functional errors are mostly detected by performing execution tests and by verifying the results of those tests.

However, as we mentioned above, we discovered functional errors by performing an analysis of a hardware component without any functional specification.

In this section, we present on one specific case how an incoherent behavior allow us to automatically detect a functional error and to localize its origin.

2.1 A small component: a size-converter

During the conducted case-study, we first proceed with the analysis of different components used to implement a full functional system. Among other components, a size converter were used to connect two buses of different sizes – the sizes of the two buses are parameters of the generic component.

The size converter receives requests from an initiator. It first converts a request message into a sequence of smaller (resp. bigger) words, it then sends this converted request to the target and waits for an answer from the target. It then translates the answer message into a sequence of bigger (resp. smaller) words. In the following, we suppose that the initiator bus is smaller than the target bus. For instance, the size converter translates a four words request message from the initiator into a two words request message. It translates a two words answer coming from the target into a four words message.

The size converter also support some QoS. Words refused by the initiator bus are stored inside a buffer.

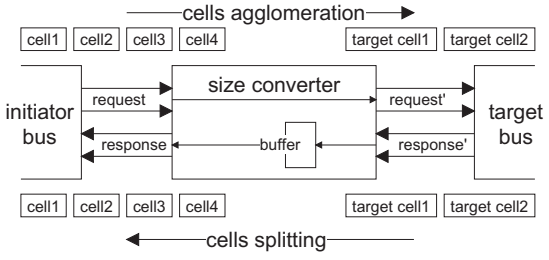


Figure 1: Size converter connections.

Messages are identified by a unique id. Each packet carries a message id and the last packet of a

message also carries a flag ”end of message”. Packets can interleave and the size converter receives and reconstructs the full message before proceeding with the conversion.

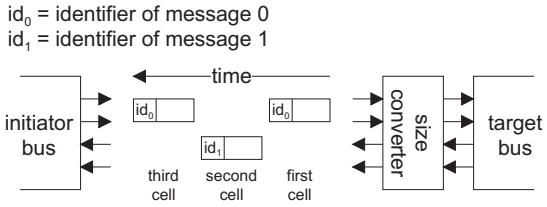


Figure 2: Messages’ scheduling.

2.2 The error and its manifestation

When performing a static analysis of the size converter, the only potential errors we expected to find were in the decision procedure that gives grants. The size converter gives grants as long as memory is available in the internal buffer. The decision procedure of the component is error-prone since many internal signals have been introduced to optimize transmission delays. However, despite the inherent complexity of the decision procedure, no errors were found inside it.

The analysis nevertheless detected an error when handling with ”unaligned data”. This error was not located in the decision procedure. More surprisingly, neither the specification, nor the tests and their associated oracles could capture this error. This error was purely functional.

When proceeding with the conversion of messages sent by the target – ie. an answer to a specific request – in some specific cases, the size converter may handle unaligned data. Decision to handle it only depends on the structure of the message sent by the emitter (in our case the target). However, the structure of messages sent by the receiver (in our case the initiator) were also used to decide whether we are in presence of unaligned data.

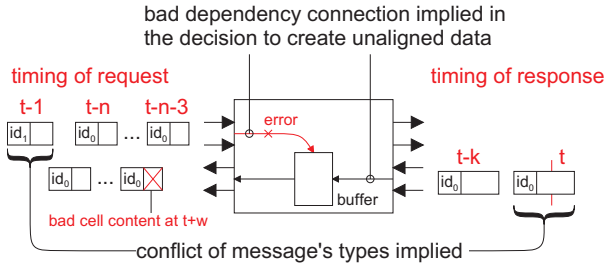


Figure 3: Unaligned data bug.

Creating unaligned data does not violate the protocol: messages remain well structured, the ordering of the packets is preserved, the message ids remain also intact. However, the content of the message get corrupted. The semantic content of the transmitted message after conversion will be altered w.r.t. the answer message that the target sent.

2.3 Why is detecting such errors so difficult ?

Conceptually, such a data corruption may be detected when testing the complete system in presence of components, that requires data should be aligned. However, the data corruption would be detected later inside the component that will handle the data and not inside the size converter. Once the error is detected – and if it is detected – the designer has to find its origin. We should not forget that the presence of many cache associative memory make its localization much more difficult.

Regarding this specific case, the error has not been detected by the tests performed on the SystemC code. The unitary tests are correct since the messages are not long enough, their chaining makes the correct decision or the oracles are too robust with respect to the data to detect the error. The integration tests are not fine enough to make such verification.

Pure formal methods have also difficulties to find such errors. There is no protocol violation and no invalid state. Basically, the only way is to add a more detailed specification that goes over the protocol and that specifies for each type of message passing through the size converter the valid conversions not altering the meaning of the message.

Yet in this case, a bus does not care about the types of transmitted messages. Thus the specification breaks the component modularity and is not adequate.

The last way consists in specifying all the dependencies that allow to handle with unaligned data. With such information, model checkers should detect a violation. However, specifying all those dependencies requires a considerable effort that requires updates each time the component evolves.

2.4 How to analyze the behavior of a generic and parametrized component ?

In a modular, bottom-up approach wih reusable components, we first decided to analyze the size converter as an independent IP component. To verify the expressed properties, we had annotated the SystemC code of the size converter with assertions. The job was ”to automatically verify that the assertions were correct if the initiator and target bus verify the bus protocol as well as the size converter protocol”.

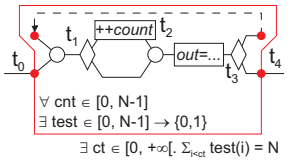
Our static analyzer performs an abstract but symbolic simulation of the main process. It symbolically

executes the code and builds (1) a support for the execution traces, (2) the current trace, (3) the current formal values of registers, signals, path condition and time as specified below.

signal	$\longrightarrow$	*trace*	$\longrightarrow$	*symbolic value*
register	$\longrightarrow$	*trace*	$\longrightarrow$	*symbolic value*
sc_time	$\longrightarrow$	*trace*	$\longrightarrow$	*symbolic value*
condition	$\longrightarrow$	*trace*	$\longrightarrow$	*symbolic value*

As an example, on the SystemC code

```
while (count < N) {
   wait(clock);
   if (msg.bit)
      ++count;
   out = count%10;
};
```



Traces' support

t_2 is the trace that has done $cnt < N$ loops and that has finally answered `true` to the test `msg.bit`. The analysis progressively produces this final formal result.

count	t_0	0
	t_1	$\sum_{i<cnt} test(i)$
	t_2	$1 + \sum_{i<cnt} test(i)$
	t_3	N
msg.bit	t_2	1
	t_3	test(cnt)
out	t_3	$(\sum_{i\leq cnt} test(i))\%10$
	t_4	N % 10
sc_time	t_3	cnt
	t_4	ct

Organized by traces, the result is a set of logical formulae that describes the evolution of data. As for all static analyzers, if we want a verdict in finite time, we have to reconcile the following choices :

- we may lose information. But losing information may quickly mix the contents of the packets sent. Consequently false warnings will be issued and a lot of work is required to distinguish between false warnings and real errors.
- we may infer a minimal set of properties that may be refined later if precisions are needed. But inferring a minimal set of properties does only allow to verify the specification we want to verify.
- we may try to keep exact and formal information as long as long the amount of information to be handled with does not explode. But keeping exact and formal information requires complex heuristics and abstraction techniques when the amount of information starts growing exponentially.

For the size converter, we decide to keep the maximum of formal information in order to precisely solve further assertions. Several heuristics avoid the inherent explosion of the amount of information. Among other heuristics, the extraction of what we call the *component protocol* is the most significant one. The extracted *component protocol* represents *a posteriori* the sequence of events that a given component is waiting for. It looks like the traces' support with formal conditions to take transitions.

On the size converter, the analyzer extracts the following *component protocol*.

```
Initialization through a reset - event e0
Do n times
      Receive a first packet - event e1
      Do p times
           Receive a non final packet - event e2
      Receive a final packet - event e3
Optionnaly receive a first packet - event e1
Do p times
      Receive a non final packet - event e2
```

To find the *component protocol*, the analysis performs abstract interpretation iterations where each iteration step adds new edges, new nodes to the traces' support and new symbolic values to the data descriptions. When loops are encountered, a formal loop counter is automatically introduced, that helps to simplify the symbolic values of the data in the loop. Symbolic values are forced to converge on logical, stable and combinatorial formulae valid for all values of the loop counter. Note that sequential formal values are expressed as combinatorial formulae w.r.t the *component protocol*. That is why the *component protocol*, shared by the sequential values, has got its structure obtained through a vote between the different formal values. The analysis ends when convergence is met, a standard process within fixpoint iterations. This kind of analysis is an adaptation of [6, 7] for systems.

On figure 4, we describe the steps that participate to the construction of the adequate protocol that is the valid sequence of packets received by the size converter. Without any analysis, the protocol is simply an unstructured sequence of events e_1, e_2, e_3, represented by the regular expression $(e_1 \vee e_2 \vee e_3)*$. Given that some registers have got cleared since the processing of an event e_3, the data descriptions naturally merge after this kind of event. Hence the analyzer starts with transforming the initial unordered sequence of events $(e_1 \vee e_2 \vee e_3)*$ into the ordered sequence of events $((e_1 \vee e_2)*.e_3)^n.(e_1 \vee e_2)^p$ and goes on as documented on figure 4.

2.5 What look the inferred properties like ?

In addition to verifying that the assertions are not violated, we discovered a relation between the size of

step 1 e_1

step 2 $e_1.(e_2 \vee e_3)$

step 3 $e_1.e_2.(e_2 \vee e_3) \vee e_1.e_3.e_1$

step 4 $e_1.e_2^2.(e_2 \vee e_3) \vee e_1.e_2.e_3.e_1 \vee e_1.e_3.e_1.(e_2 \vee e_3)$
$= (e_1.e_2^2 \vee e_1.e_3.e_1).e_3 \vee e_1.e_2^3 \vee e_1.e_3.e_1.e_2$
$\vee\ e_1.e_2.e_3.e_1$
Same data description implies merge of traces

step 5 $((e_1.e_2^r.e_3)^n.e_1.e_2^q).e_3$
$\vee\ (e_1.e_2^2 \vee e_1.e_3.e_1).e_3.e_1 \vee ((e_1.e_2^s.e_3)^m.e_1.e_2^p$
Data description with same structure for formulae implies merge of traces with loop counters

step 6 $((e_1.e_2^r.e_3)^n.e_1.e_2^q).e_3$
$\vee\ ((e_1.e_2^t.e_3)^l.e_1.e_2^u).e_3.e_1 \vee ((e_1.e_2^s.e_3)^m.e_1.e_2^p$
Stable component protocol

step 7 $((e_1.e_2^r.e_3)^n.e_1.e_2^q).e_3$
$\vee\ ((e_1.e_2^t.e_3)^l.e_1.e_2^u).e_3.e_1 \vee ((e_1.e_2^s.e_3)^m.e_1.e_2^p$
with $n, l <$ buffer_size, $m \leq$ buffer_size
Constraint resolution

step 8 $((e_1.e_2^r.e_3)^n.e_1.e_2^q).e_3$
$\vee\ ((e_1.e_2^t.e_3)^l.e_1.e_2^u).e_3.e_1 \vee ((e_1.e_2^s.e_3)^m.e_1.e_2^p$
$\vee((e_1.e_2^r.e_3)^{buffer\_size+a}.e_1.e_2^q).e_3$
$\vee\ ((e_1.e_2^t.e_3)^{buffer\_size+b}.e_1.e_2^u).e_3.e_1$
$\vee\ ((e_1.e_2^s.e_3)^{buffer\_size+c}.e_1.e_2^p$
with $n, l <$ buffer_size, $m \leq$ buffer_size,
and $a, b \geq -1$, $c \geq 0$. **Stable, no more behavior**

Figure 4: Abstract simulation and structure on message receptions

internal buffers – since the size converter grants all inputs until internal buffers may be filled – and the number of messages the size converter may handle without any delay. Such an information is useful *per se* to determine the size of buffers when instantiating a small component in a bigger system.

As we have seen before, the selected approach tries to compute a set of stable logical formulae. However, in some cases, the analyzer fails in finding stable formulae. We must distinguish between different cases:

1. the size of the computed logical formulae is growing exponentially. If this happens, we have to use approximation techniques to reduce the size of the logical formulae and to compute an upper-approximation of the set of all the reachable states. Such approximation techniques have already been used in some model-checkers [3].

2. the functionalities implemented by the component heavily relies on functionalities provided by other components. This case happens when the component is not reusable out of the given system. In this case, the analysis of the component may only continue when we will connect this component in the system.

3. a meta-stable generalized logical formulae may be inferred. It so relies on the complete history of the component, different from the *component protocol*. Such a behavior points to a potential functional error.

During the analysis of the size-converter, the analysis has found for each intern subcomponent an adequate formula defining its evolution. The only exception came from the conditional that handles with unaligned data. The analyzer was not able to find a stable logical formula. It has generated a recursive formula depending on the whole history of the component since the last reset. This erratic behavior may suspect a functional error. We finally found the source of the error thanks to a review from the start of the explosion.

3 A new characterization of functional errors

As a consequence of the conducted case-study, we propose to suspect errors when we observe a formal explosion that does not simplify, except with a loss of precision or with the whole history.

3.1 Why is it an error ?

The notions of simplicity, modularity, focus on one task, are inherent and essential for hardware and software components. Focused on one task, a non trivial component should do something that does not arbitrarily mix different behaviors. The functional error we point out violates this implicit rule.

3.2 Why detecting functional errors during early design stage is difficult ?

Functional errors often appear mixed with other implementation errors, they may only be detected at the end of the following refinement process :

- During the first tests of a new design, many assertions fail. The simulation identifies local implementation errors and failed assertions.

- During the first debugging stages, only the trivial simulations are correctly done: the other tests fail or return aberrant states. The user extracts scenarios that fails as non-regression tests. It contains the environment description, the formal stimuli and the traces of simulation.
- During intensive debugging, simple simulations are correct and the most complex simulations fail. Errors do not question the global architecture, but indicate some difference between specification and implementation. The user progressively refines the complex simulations scenarios to focus on the error. During the refinement process, he can replace a transactional description by a BACA or a RTL description.
- After the validation report, all the simulations seem to be correct, all expressed assertions have the expected behaviour. The existence of some residual bug cannot be excluded though, since we never take into account implicit or purely functional specifications.

If the standard test process may detect the vast majority of errors, the following errors may be not captured by simulation and occurs in the following cases:

- simple events never tested since they occur with very low frequency.
- simple events never tested because they are quickly implemented to add a non-forecast functionality in the existent range of tests.
- succession of simple events setting more complex events, such as the buffer filling up.
- error set in motion from an unlikely combination of these different cases.

The difficulty of detection increases with the functional nature of the errors, since assertions do not cover the whole specification. And simulation is not the only verification technique having problems with detecting functional errors.

3.3 What are the limitations of standard formal verification techniques in this case ?

In this section we review the most widely-used techniques, their scopes and limitations with respect to the given problem and also discuss how to extend the technique to make functional error detection possible.

- Model checking is certainly the most effective technique to formally detect errors. It can handle with simple sequences of events as well as complex ones. However, model checking requires a lot of effort to translate the properties into logical expressions that model checkers can understand. Thus, the detected errors are only the errors capable of inferring a violation of the protocol. In the present case, since the specifications do not cover the contents of the message, the error in the size-converter could not have been detected, We can improve model checkers with a dynamic refinement of properties[8], by taking explicitly into account the implementation of the component. The states should then merge according to high level or user's criteria to avoid explosion. If the user cannot avoid it, the first states that explode may indicate the location of the error.
- Theorem provers only deal with simple events. This limitation is not theoretical, but practical. For instance, our formal debugger uses some technology like local simplifications from theorem provers. The succession of simple events commonly requires a user's assistance. To find the functional error, the user should express the evolution of each register and signal according to a protocol in construction. If it fails, the user can suspect an error. To follow this process, more automation should be introduced in theorem provers, especially for the protocol construction. The protocol is naturally big and it is built with many heterogeneous notions. Theorem provers should then introduce some merge heuristics present in model checkers. Techniques of theorem provers are compatible with the ones of model checkers and they closely work through PVS [1].
- Static analysis like abstract interpretation will set warnings for the errors resulting from simple events and from the successions of simple events. It allows to give automatic verdicts regarding the successions of simple events. However, this technique performs an analysis of a complete system [4]. Therefore, abstract interpretation could only be used at the end of the development and not during the various development stages. Since abstract interpretation does not support the notion of specification, there is no way to explain to an analyzer based on this technique that such a functional error should be detected. Consequently, no analyzer based on abstract interpretation will produce no warning at all.

3.4 Methodology to detect a functional error using a formal debugger

Our result suggests to apply the following test methodology for the verification of IPs. For each individual component, a static analyzer should extract a formal description of the behavior of the component, The user should assist the static analyzer providing

SystemC assertions and suppressing impossible scenarios introduced for exhaustiveness. The static analyzer should be able to find a set of stable logic formulae that describes the behavior of the system.

The set of stable logic formulae may be used as an abstraction for model checking, as a coverage objective for test case generators and as a base description for theorem provers (based on the Hoare Logic for instance).

If no stable logic formulae can be inferred, we must distinguish between three cases :

- the size of the logical formulae is exploding because of the complexity of the component,
- the behavior of the component depends on functionalities provided by other component,
- the logic formulae describes an incoherent behavior with respect to the component parameters. Here the origin of the error is at the point where the first decision procedure does not stabilize.

In fact this methodology is really applicable as soon as the component development covers all functionalities and either before passing the first tests or just after. It proves the validity of the version and authorizes the automatic generation of test cases for non-regression of future versions. Since this methodology implies some work – it is not a push button methodology – it must be used for releases candidate. But it gives confidence in the component, in its reuse and in its maintainability: all that is the essence of IPs.

Errors that can be detected using this methodology are: (1) a violation of the protocol or the specification, (2) a violation of internal and external assertions, (3) an incorrect access to data structures like an out of bound access on internal buffers, (4) information used before their definitions, (5) information stored but never used, (6) aberrant dependencies, (7) internal deadlocks in a component or possible live locks with extern components, (8) incoherent implemented component functionalities.

4 Summary and Conclusions

In this article, we develop a methodology to achieve exhaustive verification even on behaviours not covered by the specification. We motivate the importance of constructing the protocol that describes the formal behavior attached to each component. This protocol allows simplifications of all internal decisions, except the erroneous ones.

In this methodology, the protocol construction is the most complex task. We propose a process that will be integrated in a “formal debugger”. The protocol construction interacts with the simplifications. Heuristics tends to maximize the number of decisions involved. Based on user’s assisted scenarios and on the protocol, the formal debugger should provide a precious help to track the large kinds of errors until exhaustive verification.

The “formal debugger” should assist the code’s review but may intervene very early during design, which we call the ”*debug as design*” design approach. The protocol verifies that each component is dedicated to a single task in an often complex but abstract environment. Hence it enables an automatic verification of assumptions when components are connected. The verification specializes each components protocol and automatically propagates on each internal decision without any additional work. Hence the methodology should be scalable, as soon as it will be fully supported by tools.

References:

[1] T. Arons, “Using Timestamping and History Variables to Verify Sequential Consistency”, in LNCS 2102, pp. 423, 2001

[2] F. Bourdoncle, “Abstract debugging of higher-order imperative languages” *Proc. of the ACM SIGPLAN PLDI 1993* Albuquerque, New Mexico, pp. 46 - 55, 1993

[3] E. Clarke, O. Grumberg, D. E. Long, “Model Checking and Abstraction”, *ACM Transactions on Programming Languages and Systems*, Vol. 16, pp. 1512–1542, 1994

[4] P. & R.Cousot, “Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation” *Proc. of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet*, SSGRR 2001, Compact disk, L’Aquila, Roma, Italy, 612 August, 2001.

[5] D. Geist and al. “A methodology for the verification of a system on chip” in *Proc. of the 36th ACM/IEEE conference on Design automation*, New Orleans, Louisiana, United State, pp. 574 - 579, 1999

[6] L. Gonnord and N. Halbwachs “Combining Widening and Acceleration in Linear Relation Analysis” in *Proc. of the 13th Static Analysis Symposium*, 2006

[7] D. Gopan and T. W. Reps “Lookahead Widening” in *Proc. of the 18th Conference on Computer Aided Verification*, Seattle, WA, USA, pp. 452 - 4466, 2006

[8] Shaz Qadeer, “Algorithms and Methodology for Scalable Model Checking,” *Ph.D. thesis, University of California at Berkeley*, October 1999.