UDC 519.686

M. GLUKHIKH[1,2], M. MOISEEV[2], H. RICHTER[1]

[1] *Clausthal University of Technology, Clausthal, Germany*
[2] *Saint-Petersburg State Polytechnical University, St. Petersburg, Russia*

A STATIC ANALYSIS APPROACH FOR FORMAL VERIFICATION OF SYSTEMC DESIGNS

A novel approach for formal verification of SystemC designs is presented which is based on static analysis and logical inference. It allows to specify and to verify properties of SystemC processes as functions over time. Part of that approach is the new "Aegis FDL" language for property specification. Furthermore, we wrote a plug-in for the gnu gcc compiler which represents the SystemC design internally by a control flow graph. A subsequent time course analysis is applied to obtain the process' states at all simulated points in time. Property checking is implemented by selective linear definite resolution and allows to check assertions and to identify inactive branches. The applicability of the approach is shown by an example.

Keywords: *SystemC, Static Analysis, Logical Inference, Formal Verification, Abstract Interpretation.*

1. Introduction

Digital hardware design relies mostly on synthesizable code for field-programmable gate arrays (FPGA). However, hardware changes are more tedious and time consuming compared to software updates which is why every error in a hardware design is expensive. The reason for that is that a FPGA does not support the designer by an operation system or a run-time system, it has no keyboard, no screen and no high-level debugger. Because of that, all hardware errors should preferably be found and eliminated in an early stage of the design. During the design process, it must be ensured that the design matches its specification, i.e. that it has all prescribed functional properties. One way for that is testing: a testbench is created on the FPGA in parallel to the design which asserts combinations of input signals and checks the output. However, because of the exponential growth of possible combinations with the number of input signals, testing can normally not be exhaustive. An other way is simulation. All hardware description language such as VHDL [1], Verilog [2] or SystemC [3] have more or less comprehensive simulation capabilities. These simulators allow to check some properties of a design in the prepared testbench before downloading the design into FPGA. A third way for ensuring the needed functionality is formal design verification by means of model checking [4,5], static analysis [5,6,7] or deductive verification [8], for example.

SystemC [3] was originally created as a hardware description language with built-in simulation features. With the advent of commercial and open source high-level synthesis (HLS) tools that can translate simulated SystemC into real VHDL or Verilog code that can be synthesized for a programmable logic gate array (FPGA) situation has changed significantly. Examples for such HLS tools are CTOS [9] from Cadence or Catapult C from Calypto [10]. One cause for the situation change is that SystemC has a higher level of abstraction, especially when its C++ features are used, compared to traditional hardware description language like VHDL and Verilog. An other reason for the situation change is the comprehensive simulation features of SystemC. Both together allow to shorten the development cycle for FPGA designs significantly.

In this paper, we propose a novel approach for formal verification of SystemC designs that is based on a combination of static analysis methods and logical inference methods [11,12]. This approach uses a special language for specification of the required functional properties. The rest of the paper is organized as follows. In chapter 2, an overview of the approach is given. In chapter 3, the used intermediate representation is briefly considered. In chapter 4, the functional description language is presented. Chapter 5 describes the developed verification method in detail. Application of the approach to a practical example is considered in chapter 6. The paper ends with a conclusion and a literature list as reference.

2. Overview of the Approach

The approach is based on the idea that the designer of an FPGA creates his design and writes-up in parallel what functional properties he expects from it. Both, the design and the functional properties have to be specified by the designer in a formal language. For the specification of the functional properties, we created a new language called "Aegis FDL". After the implementation of the design and the description of functional properties are completed, an

automated comparison of both is performed in order to detect mismatches. The automated comparison consists of the following three steps which are considered in the following chapters:

1. Creation of an internal representation of the analyzed design.
2. Parsing functional properties from Aegis FDL source.
3. Verification of functional properties with use of static analysis and logical inference methods.

2.1 Approach Restrictions

Together with the CTOS compiler from Cadence and in compliance with many other SystemC compilers, our approach has for practical reasons the following restrictions:

1. Event-based *wait/notify* statements and shared variables are not allowed.
2. Channels are limited to *sc_signal*. User-defined channels or *sc_fifo* are not handled.
3. Only synchronous designs with a single *sc_clock* are possible, clock-based waits are allowed.
4. Loops must be finite, and a number of iterations must be deducible from the source code for each loop. One infinite loop is allowed for a process, and a *wait()* or *wait(n)* statement must exist in all execution paths of the loop.

Interprocess communication is allowed in principle but not considered yet.

3. Intermediate Representation

The approach uses as intermediate representation of the examined design an enhanced control flow graph. We have developed the ssadump plugin [13] to the gcc compiler to create such a control flow. The plugin acquires objects from the gnu compiler after its static single-assignment pass [14] and saves them in a file in JSON [19] format. This file is used to build-up the control flow graph. After the compiler's pass for static single-assignment, the successor and predecessor of every statement is known, and most design statements were simplified which is both needed.

The intermediate representation is also extended by different design properties, such as the set of declared parallel processes, their main function and the used signals and ports. The properties are determined by abstract interpretation [15] of the *sc_main* function of the design (see Chapter 5). Additionally, we use a slightly modified SystemC source code which includes callbacks. For example, the *sc_port* constructor adds a new port to a process, and the *SC_CTHREAD* macro detects a new process, together with its main function.

4. Aegis Functional Properties Description

Language

We have developed a language called „Aegis FDL“ to describe the properties of a SystemC design as functions over time. Aegis FDL specifies the design properties by a set of conditions that are valid at a point **T** in time, together with subsequent assertions. FDL thus reflects the real-time behavior hardware designs have. The basic element of Aegis FDL is the following inference:

```
(T, <condition>) => (<time>, <assertion>)
```

It means: if `<condition>` is true at a point **T** in time then the assertion will become true at the same **T** or at a later point which is specified by `<time>`. Furthermore, in the approach the course of time is discretized and measured in "clock ticks". Each clock tick is the time quantum that the subsequent analyses will simulate. The Backus-Naur [16] notation of `<condition>` and `<assertion>` of the inference is as follows:

```
<condition> ::= <unary_operator> <condition> |
  <condition> <binary_operator> <condition> |
  <timed_element> <comparison_operator> <value>
<assertion> ::= <assertion>, <assertion> |
  <timed_element><comparison_operator><value>
<comparison_operator> ::= == | != | > | < |
                          => | <=
<binary_operator> ::= || | &&
<unary_operator> ::= !
<value> ::= <timed_element> | <literal>
<timed_element> ::= <element> | <element>(<time>)
<time> ::= T | T + <NUM>
<element> ::= <NAME> | <element>.<NAME> |
  <NAME>[<NUM>]
```

`<condition>` and `<assertion>` are defined by means of the symbol `<timed_element>` which denotes the value of a variable or of a signal at some time moment. The symbol `<time>`, as part of `<timed_element>`, will be evaluated either at **T** or at a later point in time **T+<NUM>** in order to check whether `<assertion>` is true. **NAME** is a terminal symbol and denotes the name of a signal or a variable. Furthermore, **NUM** is also a terminal symbol, which is a literal of type unsigned integer. Finally, we have added into Aegis FDL <NAME>[<NUM>] and <element>.<NAME> in order to support C++ arrays and structures.

5. Verification Method

Our verification method is based on abstract interpretation, which is well-known static analysis technique [15]. This technique works by standard interpretation of the design, but uses so-called **abstract semantic domains** to describe potential variable and signal values. This stands in contrast to standard interpretation which uses non-abstract, i.e. normal semantic domains. A **semantic domain** is the set of all concrete values a variable or signal can assume in a design. For example, an integer variable has a specific set of 32 bit integer values, and a

pointer variable has a set of addresses. In an abstract semantic domain however, values are not always exactly known, only intervals may exist. For example, an integer variable can be abstracted as **[min...max]**, and a pointer variables by the tuple **(object, offset)**. **Object** is that location to which the pointer points-to, and **offset** is an interval increment to that object. Additionally, an abstract semantic domain allows the value **any** which means there is no concrete value information available.

For abstract interpretation, variable and signal values are stored in a so-called **process state** which is a set of tuples (**element**, **value**) for any process in the design. A tuple **element** is either a variable of a simple type or it is part of complex type such as array or structure. The tuple **value** is taken from the abstract semantic domain of the tuple **element** and depends on its type. Each process state is valid only for a specific **program point** which is the space between two consecutive, sequential statements. So every statement in the process has at least two states associated to it. These are the **input state** immediately before a sequential statement, and the **output state** immediately after that statement. Conditions such as *if* or *switch* have one **input** state but more than one **output states**, depending on the number of *else/case* branches.

5.1. Analysis of the Time Course of the Process

The target of this analysis is the main function of the process under consideration. In the main function, the control flow of the process and its FDL property are examined. This is done by simulating each clock tick in one iteration which is performed by **time step analysis** and **predicate analysis**. As a result, the time course analysis will answer whether a FDL property is matched by a design or not.

Let's consider the point **T** in time as the moment when a FDL condition becomes true. Then, the time course analysis has to start already $\mathbf{N_{back}}$ clock ticks earlier at $\mathbf{T-N_{back}}$ in order to reach full precision. $\mathbf{N_{back}}$ is the maximum number of **wait()** statements of all execution paths. It is also possible to use less than $\mathbf{N_{back}}$ clock ticks but then then the maximum precision will not be reached. In case of an infinite loop, an iteration is made for every loop cycle until time **T** is reached.

Additionally, we define "**initial points"** as the program points where the process can start with a first iteration. The set of all initial points will include those program points which are located immediately **after** the wait() statements of the process, if there are any. In the initial points, the **process state** contains tuples only with **any** as element value.

In the following iterations, time step analysis and predicate analysis are executed hand in hand in order to simulate one clock tick. The analysis ends at so-called **final points** which are those program points that lie immediately **before** wait() statements. For every next run, **final points** of iteration *i* will become the **initial points** of iteration *i+1* because the wait() statement is the border for each iteration. Furthermore, also process states evolve before the run, because signal values should be updated according to the SystemC specification [3]. So, each signal value is internally represented by a pair of (**R-value, W-value),** where **R-value** is the value of the signal before an iteration and **W-value** is the value afterwards. For the developer of FDL properties, it is sufficient to specify the signal R-value of every condition and assertion.

After $\mathbf{N_{back}}$ iterations, the time course analysis of the process reaches **T**, and the FDL property of the process that was specified to become true at time **T** or later has to be checked. To accomplish this, information about the property's condition is added to the process state at time **T**. From time **T** on, $\mathbf{N_{forw}}$ additional iterations have to be executed because in the coming interval $\mathbf{[T...T+N_{forw}]}$ all FDL assertions must match with the design. At the end of an iteration, the assertion that is prescribed for that clock tick is tested with the help of predicate analysis. In the case that an assertion does not match with the design, the analysis terminates with a negative result, otherwise it continues until $\mathbf{T+N_{forw}}$ and ends with a positive result.

5.2. Time Step Analysis

The purpose of the time-step analysis is to determine signal and variable values in every clock tick. The analysis starts in **initial points** which were determined by time course analysis. It stops in **final points**. All possible execution paths from initial points to final points are traced. For each sequential statement in a path, the analysis considers the **input state** of the statement and updates it to the **output state**. For example, the statement *b=b+1* modifies *b* value from **[min...max]** in the input state to **[min+1...max+1]** in the output state.

For a conditional statement such as *if*, the **output states** of all *then…else* branches are calculated. For instance, the statement "*if (a>0)"* will result in a positive value for *a* in the *true* branch and in a non-positive value for the *false* branch as output state. For an input state of the *if* statement that is defined by the interval **[-N...P**] for *a*, the output states of the *if* statements will be defined by the two subintervals **[-N...0]** in case of false and **[1...P]** in case of true. If a finite loop is encountered, the loop's body is iterated n times, where n is the loop's cycle limit, provided that n could be determined as a static value from the design.

5.3. Predicate Analysis

Predicate analysis examines the interdependencies

between elements. An element is either a variable of a simple type such as float or integer or it is part of a complex type such as array or structure. The interdependencies are described in predicate logic of second order. This logic type is built-up of terms of first-order logic and of so-called quantifiable relation variables. For predicate checking, each process state is extended by a set of predicates which will become true for those program points for which they were defined.

Because of the fact that the value of an element can change over time and between program points as well, the predicate checking must follow. For example, the statement *b=b+1* may give a predicate *"b(T, next) = b(T, prev)+1"*, where *prev* and *next* are the program points immediately before and after the statement.

After predicate checking of a statement, new information is created and stored in the process state. For example, the information *"a>b"* is created in case of an *"if (a>b)"* statement for the *true* branch, and *"a≤b"* is created for the *false* branch. It happens also that information is removed from the process state if a variable is no longer used after some clock ticks.

Finally, predicate checking examines also whether a FDL condition is true for all possible values (=provable) or whether it is true only for some possible values (=satisfiable). There are two situations where the distinction between provable and satisfiable is important: The first situation occurs if an *„if"* or *„switch"* statement is examined. In this case, a branch is classified as active if is satisfiable. It is classified as inactive if the negation is provable. The second situation occurs during property check in order to find a design error in source code. In this case, a property can be verified if its assertion is provable, or it can be disproved if the negation of its assertion is satisfiable.

In order to find-out whether an assertion is provable or satisfiable, we are using Horn clauses and selective linear definite resolution (SLD resolution) [11]. In general, a Horn clause is a disjunction of literals where there is at most one positive literal. For inference rules representation, so-called definite clauses with exactly one positive literal are used. They can be written in implication form: $p_1 \wedge p_2, ..., \wedge p_n \rightarrow q$. For assertions representation, goal clauses are used which contain no positive literals and can be written in conjunction form: $q_1 \wedge q_2, ..., \wedge q_n$. The resolution of a goal clause with a definite clause into a new goal clause is the basis of the SLD resolution inference rule. It is also used to implement logic programming and the programming language Prolog. SLD resolution proves sequentially that $p_1, p_2, ..., p_n$ are true to make the conclusion that q is true.

Although SLD resolution is simple and thus fast it is powerful enough for predicate analysis. We have also considered to use commercial tools, such as the Microsoft Z3 Solver [17] or the HOL theorem prover [18], for example, but it turns out that it is very time-consuming to convert data between analyzer internal format and the tool's format.

6. Example Application

As an example of the approach, a single storage element is considered that is depicted in fig.1. It allows for push and pop operations, and provides for full and empty flags. Its implementation in SystemC is given below.

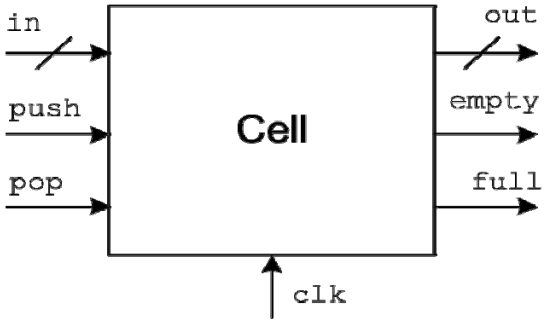

Fig. 1. Stack-like memory cell and its ports.

```
1 :template <class T> SC_MODULE(hwcell) {
2 :  T data;
3 :public: // ports omitted, see fig. 1
4 :  SC_CTOR(hwcell) {
5:     SC_CTHREAD(run, clock.pos());
6:   }
7:   void run() {
8:     bool isFull = full = false;
9:     empty = true;
10:    data = out = 0;
11:    wait();
12:    while(true) {
13:      bool pushed=push && (!isFull || pop);
14:      bool popped = (pop && isFull);
15:      if (popped) isFull = false;
16:      if (pushed) data = in, isFull = true;
17:      full = isFull;
18:      empty = !isFull;
19:      out = data;
20:      wait();
21:    }
22:  }
23:};
```

For this design, we define a property in Aegis FDL as:

```
(T, (push == true) && (empty == true)) =>
     (T+1, full == true, out==in(T))
```

This property specifies that if at time **T** the cell is empty and *push* is true, then the value at the *in* port is stored. One clock tick later, that value is output to the *out* port and *full* is true.

The approach is performed in the following steps. Below, the signal R-values are recorded as "signal-R", while the **W-values** are recorded as "signal-W".

1. First, the control flow graph is built-up, and *run()* is identified as the main function of the process; *in*, *push*, *pop*, *out*, *empty* and *full* are found as process ports.

2. The verification starts at clock tick *T-1*, because the infinite loop of the process contains only one wait. The only initial point that can be found is located after the **wait()** statement at line 20. The initial value is **any** for all ports and variables.

3. At lines 17 and 18, at clock tick *T-1*, "*full-W(T-1) =isFull(T-1,17)"* and "*empty-W(T-1)=not isFull(T-1,17)"* predicates are extracted by predicate checking.

4. At the line 20, 2nd iteration starts, and clock tick *T* is reached. The predicates *"full-R(T)=full-W(T-1)"* and *"empty-R(T)=empty-W(T-1)"* are added to the process state. Additionally, time course analysis adds FDL conditions *push-R(T)=true* and *empty-R(T)=true* as new variable tuples *(push-R(T), true)*, *(empty-R(T), true)* to the process state.

5. At the line 13, predicate analysis infers that "*pushed(T,13)=true"* holds because of *"push(T)=true"* and because of "*isFull(T-1,17)=false"*. To infer the value of *isFull*, predicate checking uses predicates "*empty-W(T-1)= not isFull(T-1,17)", "empty-R(T)=empty-W(T-1)"* and "*empty-R(T)=true,* together with the "inverse symmetry rule" (*A=not B* ← *B=not A*) and the "equal transitivity rule" (*A=C* ←*A=B* ∧ *B=C*).

6. Because of "*pushed(T,13)=true"* at line 16 only the true branch is active, so only the predicate "*data(T,16)=in-R(T)"* and the tuple *(isFull(T,16), true)* are added to the process state.

7. At line 17, the tuple (*full-W(T), true)* is added because *isFull(T,16)* is true.

8. At line 19, the predicate "*out-W(T)=data(T,16)"* is added. Then, the final point at time **T** is reached, and the predicate *"out-R(T+1)=out-W(T)"* and the variable tuple *(full-R(T+1), true)* are added to the state.

9. As a result, predicate analysis concludes that "full-R(T+1)==true" holds and that *"out-R(T+1)=in-R(T)"* holds. The latter is true because of equal transitivity and the predicates *"data(T,16)=in-R(T)", "out-W(T)=data(T,16)", "out-R(T+1)=out-W(T)"*.Because of that, both assertions *full(T+1)==true* and *out(T+1)==in(T)* are inferred, and the property given in Aegis FDL is proved.

Conclusion

We have developed an approach which allows to verify the behavior of SystemC designs as functions over time. For that purpose, we developed a language which can describe the design behavior by properties in second order logic. The approach is based on static analysis and logical inference. Its applicability was demonstrated by an example. In the future, we will use the approach for industrial applications. Additionally, it is planned to generate Aegis FDL specifications automatically out of SystemC designs using static analysis and to employ them for the analysis of higher-level modules created from lower-level components.

References

1. *EDA-STDS.ORG Page,* http://www.vhdl.org/.
2. *Verilog Resources Page,* http://www.verilog.com/.
3. *Accelera Systems Initiative,* http://www.accellera.org/home/.
4. *N. Razavi, R. Behjati, H. Sabouri, E. Khamespanah, A. Shali, and M. Sirjani, "Sysfier: Actor-based formal verification of systemc," ACM Trans. Embed. Comput. Syst., 2011.*
5. *N. Blanc and D. Kroening, "Race analysis for systemc using model checking," ACM Trans. Des. Autom. Electron. Syst., vol. 15, no. 3, pp. 21:1–21:32, Jun. 2010.*
6. *Moiseev M., Zakharov A., Klotchkov I., and Salishev S. Static Analysis Method for Deadlock Detection in SystemC Designs // International Symposium on System-on-Chip 2011.*
7. *F. Nielson, H.R. Nielson, C. Hankin. Principles of Program Analysis. Springer (Corrected 2nd printing, 452 pages,* ISBN 3-540-65410-0*), 2005.*
8. *Using SMT Solvers for Deductive Verification of C and Java programs,* http://research.microsoft.com /en-us/um/redmond/events/smt08/filliatre.pdf.
9. *C-to-Silicon Compiler,* http://www.cadence.com /products/sd/silicon_compiler/pages/default.aspx
10. *Catapult: Product Family Overview,* http://calypto.com/en/products/catapult/overview
11. *Jean H. Gallier. Logic for Computer Science. Foundation of Theorem Proving // Harper & Row Publishers, Philadelphia, 2003.*
12. *Glukhikh M., Itsykson V., Tsesko V. Using Dependencies to Improve Precision of Code Analysis // Automatic Control and Computer Sciences, Vol. 46, No. 7, 2012. - pp. 338-344.*
13. *Static Analysis Framework,* http://www.digiteklabs.ru/en/aegis/platform.
14. *Static Single Assignment Pass,* http://gcc.gnu.org/ onlinedocs/gccint/SSA.html.
15. *Cousot, P., Abstract Interpretation, ACM Comput. Surveys , 1996, vol. 28, no. 2, pp. 324–328.*
16. *Backus-Naur Form (BNF) Semantics,* http:// infolab.stanford.edu/~burback/dadl/node8.html.
17. *Z3 Solver,* http://z3.codeplex.com/.
18. *HOL4 Kananaskis 8, http://hol.sourceforge.net/.*
19. *JSON, http://www.json.org/.*

ПОДХОД К ФОРМАЛЬНОЙ ВЕРИФИКАЦИИ SYSTEMC-ПРОЕКТОВ НА ОСНОВЕ СТАТИЧЕСКОГО АНАЛИЗА

М. Глухих, М. Моисеев, Х. Рихтер

Разработан подход к формальной верификации программ на языке SystemC. Подход основывается на использовании методов статического анализа и логического вывода и позволяет верифицировать функциональные свойства процессов SystemC. Для описания функциональных свойств разработан специальный язык “Aegis FDL”. Для внутреннего представления SystemC программ используется граф потока управления, который строится с помощью плагина к компилятору gcc. Затем осуществляется построение возможных состояний процесса для различных моментов симуляции с помощью методов статического анализа. Методы логического вывода используются для определения мертвых ветвей в ходе анализа. а также для проверки утверждений функциональных свойств. В качестве базиса логического вывода, используются дизъюнкты Хорна и правило резолюций. Работоспособность подхода продемонстрирована на примере.

Ключевые слова: SystemC, статический анализ, логический вывод, формальная верификация, абстрактная интерпретация.

Глухих Михаил – канд. техн. наук, доцент, Технологический Университет Клаусталя, Германия, и Санкт-Петербургский Государственный Политехнический Университет, Россия, e-mail: mikhail.glukhikh@gmail.com.

Моисеев Михаил – канд. техн. наук, доцент, Санкт-Петербургский Государственный Политехнический Университет, Россия, e-mail: mikhail.moiseev@gmail.com.

Рихтер Харальд – д-р техн. наук, профессор, Технологический Университет Клаусталя, Германия, e-mail: hri@tu-clausthal.de.