A Survey of Techniques for the Co-Verification of Hardware/Software Co-Designed Systems

by
Thomas S. Hall & Kenneth B. Kent

TR07-183, June 15, 2007

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
www: http://www.cs.unb.ca

Abstract

This paper describes the process of designing and verifying a hardware/software co-designed system. This is done by going through a complete case study involving polygon clipping algorithms as applied to computer graphics. As is the case in many software and hardware/software design processes, verification of the software part of the system is done using test scenarios while the hardware partition is verified using the SystemC Verification Standard methodology. This case study carries the design process through to a partial integration of the hardware and software partitions using SystemC simulation.

1 Introduction

The use of hardware/software co-design techniques in the creation of systems that utilize a combination of hardware and software to perform their tasks is common. The actual design and implementation techniques used are well defined and there are many design tools available to assist in the process. A major area of research in both the hardware/software co-design field (as well as software and hardware) is the development of methodologies and tools to verify the correctness of a design prior to its implementation. There are techniques available in both hardware and software design to perform both functional and formal verification of hardware and software only designs respectively. In the area of hardware/software co-design, the primary method for verifying a design, known as co-verification, has been to perform functional verification of the entire design and then perform formal verification of the hardware part of the co-designed system. To some extent, this makes sense since the hardware part of a system is much more difficult to change once the design has been fabricated into a physical device; but with the growing size of hardware/software co-designed systems the need for formal verification of the entire design is also increasing [1–3]. A major issue with many of the formal verification techniques is the same as one faced by formal specifications of systems; the very mathematical syntax of the techniques make it difficult for them to be accepted in practice due to the large learning curve and high costs in learning how to use them [4].

What is verification? Somerville [5] provides a simple and very succinct answer to this question on page 516:

> Verification: Are we building the product right?

In other words, is the design being created going to perform the task defined by its specifications correctly. Functional verification can be considered to be the process of ensuring that the design performs the tasks it is intended to do correctly. Formal verification can be considered to be ensuring that these tasks will always be performed correctly (e.g. no deadlock occurs).

Section 2 presents an overview of research in the areas of verification and co-verification of co-designed systems. Section 3 then discusses how some of the tools available in industry perform functional and semi-formal verification and co-verification. Finally, Section 4 provides some conclusions about the state of co-verification techniques and the future direction of the techniques being developed to make it more useful in practice.

2 Co-Verification Research

This section describes some of the research carried out in an effort to provide effective formal verification and co-verification of hardware and hardware/software co-designed systems. Techniques for both hardware/software co-verification and hardware-only verification are discussed here as both are used in co-verification.

The properties of a design that are checked by formal verification are called *temporal properties*. These properties exist only for a finite period of time, in this case the length of time that the design is in an operating state. There are several properties that are commonly checked during verification. The most common of these are safety and liveness meaning that the design is deadlock free and will eventually terminate normally.

2.1 Predicate Abstraction

Predicate abstraction is a technique for generating formally verifiable abstract models of a concrete design. This model is then used to determine if specific temporal properties of the design are adhered to throughout the design [6]. It is particularly useful for determining whether conditions such as deadlock and infinite looping exist. There are a number of approaches that utilize predicate abstraction. This section describes one developed by Clarke *et al* [6]. This method uses both the Verilog source code and the bit-level representation of a design as its input. Another is discussed in Section 2.3.1 that applies specifically to designs created using C or C++ based tools. An alternative general approach can be found in [7].

The bit-level representation of a design contains the entire design in a very detailed and structured way since it is the basis for the hardware creation process. The control structure of a design is most likely to be the part of the design with problems in it and this is extremely difficult to extract from the bit-level representation. Alternatively, using the source code of the design along with the bit-level representation simplifies the extraction process [6]. In hardware designs, control structures are represented as finite state machines and the variables that represent the state of each of these are used as the basis for generating predicates in the model. There is a major problem with naively generating a predicate for each possible state of the design, there may be as many as 2^{2^n} predicates created, where n is the number of control variables in the design [6]. Except for very small designs this leads to an intractable number of predicates. Rather than generating every possible predicate for a design, a set of abstracted predicates is generated. Each abstract predicate represents a set of concrete predicates that have common control variable values. This permits the verification of the system to be performed much more quickly than using all of the possible predicates.

The set of abstract predicates for a design is built from the design using both its source and bit-level representation. As each predicate is created, it is checked to see if there is a valid set of predicates including the new one that leads to a successful completion of the abstracted design. If there is, then that predicate represents a valid execution path within the real design and it is kept as is, or combined with other similar predicates, thus further abstracting the design. If the new predicate leads to an unsuccessful condition in the abstract model, it either reveals an error in the design or is a spurious error indicating that the abstract model requires further refinement. The only way to determine if a valid error was found or not is to perform a simulation on the actual design using the path taken by the predicate that produced the error condition in reaching that condition. If the simulation fails, verification stops reporting an error condition, otherwise additional abstract predicates are added to the model to prevent this spurious error condition from being reported again. This process is repeated until the entire design has been successfully verified. Through the use of appropriate abstraction algorithms, the size of the abstract model can be kept small and, thus, the total time required to perform a verification can be minimized. The reported experimental results show significant (orders of magnitude) performance increases when compared to other verification techniques [6].

2.2 Bounded Model Checkers

Bounded model checkers are another formal co-verification technique that deals with the state explosion issue by limiting the number of predicates that are used during verification. Rather than abstracting the design into a manageable number of predicates as is done in the predicate abstraction technique (see Section 2.1), a pre-assigned limit is placed upon the length of execution paths through a design when liveness and safety properties are being checked [8]. Predicates that produce longer paths than this limit permits are not added to the model. The resulting predicates are then submitted to a theorem prover for checking. A major problem with this type of verification is determining a reasonable limit for the number of predicates to be used [6]. If the number is too small, inadequate checking is performed. Conversely, too many predicates can cause the verification process to take too long or even become intractable. This approach to bounded model checking is applicable only to designs with a finite number of states.

Another approach to bounded model checking is described in [8] that extends the technique to include designs with an infinite number of states. In this approach, rather than converting the design directly into predicates, it is first converted into a set of automata. Each automata can exhibit the characteristics of one of six properties: safety, liveness, obligation, persistence, recurrence and reactivity. These automata are then translated into a temporal logic called Presburger arithmetic after first being converted to safety or liveness properties if not already in that form. Presburger arithmetic deals with the problem of infinite state and permits the use of theorem provers to check the design [8].

2.3 C/C++-based design Co-Verification

C and C++ have become popular as tools for designing hardware/software co-designed and hardware-only systems. This has occurred for several reasons. First the cost of a C or C++ development and runtime environment is considerably less than a hardware description language simulator for evaluating a hardware design [9]. Another reason is that speed of simulation is much faster in a C or C++ prototype of the design than using a hardware description language simulator for the hardware part of the system [9]. Thirdly, having both hardware and software design teams working in the same development environment allows for shorter design times and better inter-team communications [2]. The main drawback of performing hardware partition design in C or C++ is that like any programming languages for software-only systems, these two languages do not provide sufficient timing facilities or the ability to describe system structure adequately. These deficiencies can be overcome using special tools that provide simulation facilities that include strict timing and structure definitions. These simulation facilities provide an abstraction layer above the existing C and C++ runtime environments [10–12]. The timing provided by these simulators is based on the simulator's internal clocking mechanism and not necessarily real time. Two popular C/C++ design tools are discussed in section 3. In this section, some techniques available for the formal co-verification of designs created with them are presented.

There are a number of languages based on the C and C++ software programming languages that are intended for the design of hardware systems. These C/C++ based languages include Handel-C, Hardware-C, SpecC, SystemC and Single Assignment C [2, 13]. Since these languages are based on software programming languages they are, in some cases also suitable for hardware/software co-designed systems. Two such languages are SpecC and SystemC (see Sections 3.1 and 3.3). This section describes some of the formal techniques available for the co-verification of systems developed using these languages.

2.3.1 Predicate Abstraction

A common problem faced when attempting to verify hardware, software or hardware/software co-designed systems is the large number of states that any system can be in [2, 14]. One approach that can be used to reduce the number of states that must be dealt with during verification is called predicate abstraction (see Section 2.1). A difficulty faced in this type of formal verification is how to convert the source code of the design into a set of predicates that can be used by a theorem prover application to show that the system is correct. One of the advantages of C and C++ is that the programmer is free to choose any one of the almost limitless ways to solve a problem that are functionally equivalent. For example, in C the *for*, *while*, *do…while* and *tag: if …goto tag* constructs can all be programmed to perform the same functionality. Normally the choice of which to use is based upon the semantics of the part of a system being coded. However, when it comes to formally verifying a system, having multiple ways to perform the same operation can lead to complications. As described in Section 2.1, when the bit-level representation of a design is available it can be used along with the source code to generate the predicates. The same is not true for the binary version of a software program simulating the hardware portion of a design. This is because the binary version of a simulation program is designed to run on the targeted simulation platform and contains additional instructions for the control of the simulation.

One approach to solving this issue is to manipulate the source code of a system before beginning to generate the abstracted predicates [2]. The exact changes made to the source code of the system depend upon the programming language being used. For example, if the system is written in SpecC, it is necssary to change all loop statements to have the format *tag: if…goto tag*. Other changes include replacing increment and decrement operations (e.g. i++) with standard assigment operations (e.g. i = i + 1) and replacing all of the *wait*, *waitfor* and *notify* statements with explicit lock and unlock operations. Function calls are replaced by inlining the corresponding function bodies [2].

Figure 1 shows a simple example of how the source code of a SpecC component is changed to prepare it for abstract predicate generation. Note that the *while* loop is replaced with a tagged *if* statement and a *goto* statement. The *notify* and *wait* functions are replaced with *lock* and *unlock*. The exact format of the predicates that are created from the modified version of the source code is dependent upon the theorem prover to be used.

Another method for using predicate abstraction to verify hardware/software co-designed systems is presented in [15]. In this method, a SystemC specification is automatically partitioned into hardware and software components during the abstraction process. Hardware components are assumed to be those parts of the system that are defined to be threads sensitive to all non-clocked input signals (i.e. combinational logic) or sensitive only to a clocked signal.

```
behavior producer(
    in  event read_evt,
    out event data_evt,
    out int data_item){
    int value = 0;
    void main(){
        while (value <= 20){
            value++;
            data_item = value;
            notify(data_evt);
            if (value <= 20)
                wait(read_evt);
        }
    }
};
```

```
behavior producer(
    in  event read_evt,
    out event data_evt,
    out int data_item){
    int value = 0;
    void main(){
        loop1:  if (value <= 20){
            lock data_evt;
            value = value + 1;
            data_item = value;
            unlock data_evt;
            if (value <= 20){
                lock read_evt;
                unlock read_evt;
            }
            goto loop1;
        }
    }
};
```

Figure 1: A simple example of a SpecC component before (left) and after (right) source code modification for abstract predicate generation.

All threads that have no sensitivity to input signals are assumed to be the software partition of the system [15]. Since the combinational logic threads contain no synchronization-related control structures they are discarded from the abstract model. Any object created with the *new* operator receives its own predicate in the abstract model along with a control flag to indicate when it is active (created) or inactive (deleted). Trials using this approach show significant improvement in the time required to perform the verification of a given system when compared to other verification techniques [15] .

2.3.2 Boolean Programs and Equivalence Checking

Another approach to formally verifying designs is to break the verification into two parts, execution path checking and equivalence between refinement steps [16]. As any design (software, hardware or hardware/software) executes portions of the design need to be processed in a specific order with respect to each other. This is especially true when determining whether parts of a design executing in different threads or processes are behaving correctly. This type of analysis can be achieved by defining a set of temporal properties that describe the expected execution characteristics of the design and then analyzing the source code of the design for adherence to these properties. This can be achieved by converting the design from its original source code into a boolean program [16, 17]. A boolean program generated from a program written in a common programming language has the same control flow structure as the original program with all of the functional statements removed from it and replaced by a skip operation. In addition, all of the decision variables in the control structures are replace with boolean variables. The boolean program is analyzed for execution paths through it that violate the properties defined for the original program. If a path that produces a violation is found, that path is then checked on the original program to verify that it is in fact a violation. If the path is not a violation of the rules in the original program or design, then the boolean program is modified by adding additional boolean variables to the control structure to prevent the incorrect path from being found again. This process is repeated until all paths through the design have been checked and found correct or a real violation of the properties is found [16].

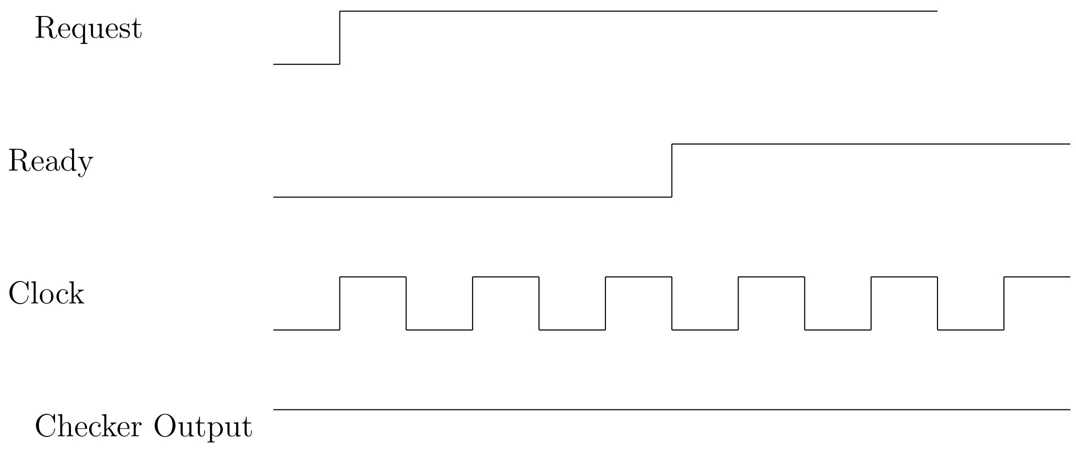
When using SystemC or SpecC to develop a system design, it is common to refine the design from a pure software one to its final hardware/software co-design or hardware only one in small increments [16]. This refinement, necessarily, changes the source code of the design to gradually contain more and more hardware synthesize-able code, but is the source code in each successive refinement equivalent to its immediate predecessor. There are two approaches to checking the equivalence of two successive increments of the same design, re-verify the entire design or check only the changed portions of the design and any portions of it that directly impact or are directly impacted by the changed portion. It is simple to find the changed portions of a design by using the Unix *diff* command on the source code of the two increments of the design. This produces a listing of all of the changes made to the design. From this listing select one of the variables involved and extract all of the source code that impacts the chosen variable from the designs and check them to make sure they are equivalent. Similarly, check all of the source code that is dependent upon the selected variable for equivalence [16]. If all of the changed and impacted portions are equivalent the new version of the design is equivalent to its predecessor. This is repeated for each variable in the difference listing. When testing only the changed part of the system and the parts directly related to it, it may not be possible to retain the timing relationships of the entire design during the testing process. This is because the timing of a design depends upon the entire design and not just upon the individual components of that design.

2.3.3 Model Checkers

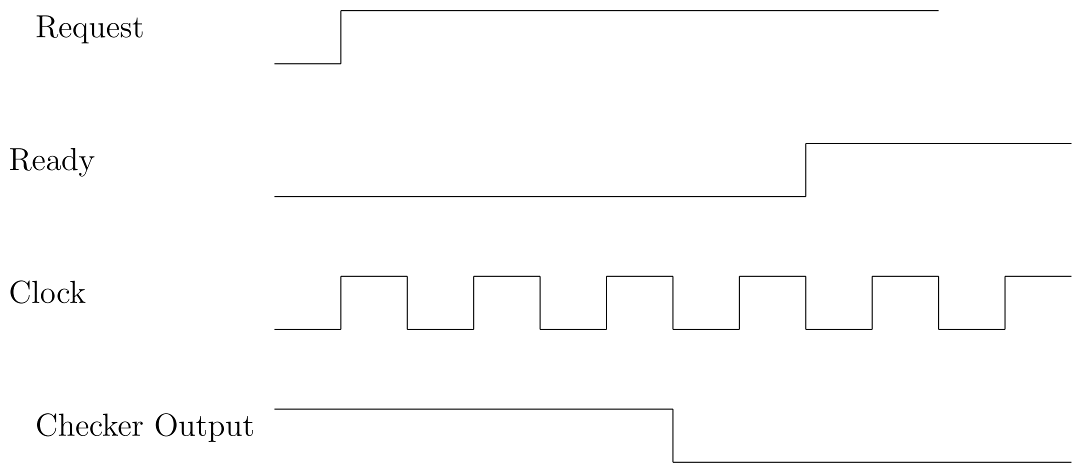
Model checkers[1] are blocks of code that are inserted into a design to check specific properties of that design [18]. These checkers can be used to check the temporal properties of the execution of the design during simulation and after full implementation. This is achieved by defining the checker in such a way that it can be implemented as a shift register. For example, consider a memory read operation. The part of the design requesting data from the memory places the address of the data on the memory address bus and then sets a flag to indicate to the memory module that it wants the data at the specified address. Figure 2 shows the timing for successful (a) and unsuccessful (b) operation of this design. Assume that the memory module is designed such that it set a data ready flag 3 clock cycles after the request flag is set. A temporal property for the design can be stated as: the memory module shall set the data ready flag 3 clock cycles after the request flag is set. To check this property, a checker with a 3 bit shift register is required along with an AND gate. One input of the AND gate is connected to the data ready flag and the other to the output of the third state of the shift register. The input to the shift register is the data request flag and it is clocked by the same clock as the memory module. As the clock runs, the data request flag is moved through the stages of the shift register until it reaches the output stage 3 clock cycles after it entered. At the same time as the data request flag reaches the output of the shift register and hence the input of the AND gate, the data ready flag should be set and applied to the other input of the AND gate. At this point, the AND gate is set if the timing is correct, otherwise it will not change state indicating that an error condition exists.

This type of checking is less formal than some of the other techniques

[1]This type of model checker is not the same as Bounded Model Checkers as described in Section 2.2.



(a) Successful memory access (Ready set after 3 clock cycles).



(b) Unsuccessful memory access (Ready set after 4 clock cycles).

Figure 2: Timing diagram for the model checker memory module example. Set is assumed to be high and unset low.

already described [18]. Its advantages are its simplicity and that it can be carried over into the actual implementation of a design. It is also applicable to any of the C/C++ based design techniques as well as other hardware description languages, although this is not stated in [18] which refers specifically to SystemC.

2.4 Polyhedral System Verification

It is possible to model co-designed systems using a polyhedral model. Such a model uses affine recurrence equations to describe the system being modelled as evidenced by the MALPHA description language [19]. The structure of the language and associated semi-automated refinement tools attempts to ensure that each design has the correct control structures for proper operation. This is not guaranteed since manual modification of a design are allowed. Verification of this type of design is performed using a mathematical approach to the generation and proving of abstract predicates than were described in the previous section. When an execution path through the system ends in a false result then an error is reported without the need to check for spurious error conditions.

2.5 Non-abstracting Verification

The issue of state explosion during the generation of predicates for formal verification have lead to techniques such as predicate abstraction to limit the number of predicates to be necessary (see Sections 2.1 and 2.3.1). These techniques normally only consider designs with control structures having a finite number of states and remove data variables which potentially have an infinite number of values [20].

In situations where it is not desirable to remove the data manipulation portion of the design during verification (e.g. a Mealy finite state machine that combines data path), abstraction techniques cannot be used. An approach that does provide a means to verify a complete design is the use of a theorem prover such as ACL2 and a modal υ-calculus. This permits the definition of a set of rules governing the generation of predicates in such a manner that there is no state explosion. There are a total of eighteen rules, described in [20].

2.6 CSP

Communicating Sequential Processes (CSP) is a formal modelling, technique that uses a process algebra to define the communication behavior of the processes in a system, whether hardware or software. Processes are defined by the events that occur to stimulate their operation rather than the operation itself [4]. This is because CSP is designed to model the communications between processes, not their internal workings. CSP can also be used in a hierarchical manner when one process is composed of one or more subprocesses. As a result of its formal nature, CSP models can be directly input to theorem provers for verification of the design at the interprocess communication level.

Two issues that arise from the formal nature of CSP are that it is not executable in its algebraic form and that it does not model the internal operations of the processes being modelled. These issues are addressed by Gardner in [4, 21]. CSP++ is a framework tool that converts the CSP algebra into a C++ program. The designer can then add functionality for the processes and execute the model to further check its validity.

3 Functional Verification in Industry Tools

This section presents three tools that are used in industry to design and perform functional verification of hardware and hardware/software co-designed systems. SpecC [12, 22] and SystemC [10, 23] are based on the C and C++ software development languages[2]. SystemVerilog is a superset of the Verilog hardware description language [24–26]. These three tools were chosen because they provide an overview of the various functional verification techniques available in the field and for their diverse overall design methodologies.

This section focuses primarily on the functional verification of designs as this is the verification technique currently supported by these tools either through the use of built-in functionality or external tools.

[2]Industry use of SpecC is limited, however, it is used extensively by researchers.

3.1 SpecC

SpecC is a design language based upon the ANSI standard version of the C programming language [12, 22]. SpecC is a true superset of the ANSI-C language supporting the design of both the hardware and software portions of a system. There is a procedure specified for SpecC that permits an entire system to be initially specified at a very high level and then gradually refined to the desired level [22]. This permits the hardware and software partitions of the system to be developed concurrently in a single design environment. The software partition is already in a software programming language so it only needs to be compiled for the target device once it has been refined. The hardware partition can be refined to the point where it can be directly translated into a netlist or other format that is directly accessible by hardware implementation tools. Throughout the entire design process the complete design is executable in software [12, 22].

The SpecC project team have shown that the design methodology associated with the SpecC language produces equivalent functionality when followed correctly [27]. SpecC, however, does not directly support any means to formally or functionally verify a design. SpecC does, though, have the ability within its simulation engine to detect the occurrence of deadlock.

3.2 SystemVerilog

SystemVerilog is an extension to the Verilog hardware description language [24,25]. The additional features contained within SystemVerilog are intended to aid in the verification of system designs, but neither the Verilog nor SystemVerilog language contain verification functionality. SystemVerilog provides the facilities to capture and output simulation traces in a format that can be directly input to verification tools, such as implementations of the OpenVera hardware verification language [25, 28][3].

SystemVerilog supports verification through the use of assertions [24]. Unlike the assertions found in languages such as C and C++, SystemVerilog assertions do not necessarily terminate the execution of a design's simulation.

[3]There are two versions of the OpenVera standard. The first, described in this document, generates direct output during simulation while the second one is intended for use with testbenches [29].

They can generate warning and information messages as well as the usual error messages. Two types of assertions are available, immediate and concurrent. Immediate assertions are similar to those found in other programming languages and execute only when they are encountered in the design during execution. Concurrent assertions use clock semantics to control their actions. That is, the activation of a concurrent assertion is time dependent rather than executing whenever it is encountered during execution. Concurrent assertions provide a snapshot of the state of one or more specified data items at specific times during the simulation of a design [24]. Concurrent assertions are particularly useful for generating output for formal verification tools since the output produced is time sequenced on clock transitions.

The immediate assertion has the BNF form (page 198 of the SystemVerilog Language Reference Manual [24]):

```
immediateassertstatement :=
        assert (expression) actionblock

actionblock: :=
        statement_or_null
        | [ statement ] else statement
```

where	assert	is the syntactic name of the immediate assertion;
	expression	is a valid SystemVerilog expression that evaluates to 0, X (undefined) or Z (high resistance) for false (expression failed) and any other value for true (expression succeeded); and
	actionblock	is the action to be performed when the expression succeeds. The actionblock can be empty or contain a single statement. It may also have an else clause that is executed if the expression fails.

When an assertion succeeds, the statement in the action block is executed (if there is one) and then the simulation continues after the action block. When the expression within the assertion fails, the else clause of the action block is executed. The severity of the failure can be set to fail (default), error, warning or information as desired [24]. This allows the behavior of the immediate assertion to be customized to suit the purpose for which it is being used. The output generated for each severity level can be determined by the system developer allowing for the output of data suitable for the type of analysis being performed.

Unlike immediate assertions which are evaluated each time the statement is encountered, the evaluation of the concurrent assertion depends upon a clock event specified by the designer. The values used in determining whether the assertion succeeds or fails are the samples of the data values taken at the time of the clock event and not those present at the time the assert statement is encountered. The clock used by a concurrent assertion is configurable for each assertion statement used in a design and need not be tied to the simulation clock [24]. The syntax of the concurrent assertion is given by the following BNF (page 247 of the SystemVerilog Language Reference Manual [24]).

```
concurrentassertionitem :=
    [ blockidentifier : ] concurrentassertionstatement

concurrentassertionstatement :=
     assert_property_statement
    |assume_property statement
    |cover_property statement

assert_property_statement: :=
    assert property (property spec ) actionblock

assume_property statement: :=
    assume property (property spec);

cover_property statement: :=
    cover property (property spec )statement_or_null
```

The concurrent assertion statements evaluates a property rather than an expression. This property describes the values that are to be examined by the assertion and, where appropriate, the timing of each value's evaluation [24].

The property of a concurrent assertion can be either a single boolean expression or a sequence of boolean expressions. A simple linear sequence is a sequence of expressions whose evaluation follows a sequential time ordering that matches their order of appearance in the sequence. Each expression within a linear sequence is evaluated on consecutive clock ticks. When more complex behavior is needed, linear sequences can be concatenated together. When the sequences are concatenated, the start time for each sequence is specified. This allows concurrent assertions to be evaluated over extended periods of time during a simulation. Times are specified as delays with respect to the completion of the previous linear sequence. The delay time for the evaluation of a sequence can be given as either a fixed number of clock ticks or as a range of clock ticks after the previous evaluation [24]. When a time range is given, the evaluation of the sequence can occur at any time during that range.

The *assert property* version of the current assertion is used to check the values specified within the given property. Success and fail processing occurs in a manner similar to the immediate assertion statement [24].

The *assume property* concurrent assertion statement is used to define values of the components of the property similar to a hypothesis to be proved. When this type of assertion is used, SystemVerilog tools are to constrain the generated design such that it conforms to this hypothesis. This is useful for setting conditions for formal verification although such verification does not need to check these assumed values. During simulation or verification, failure of any assumed property causes an appropriate report to be generated [24].

The *cover property* variant of the concurrent assertion is used to gather data about the items specified in its property so that the operation of the design can be checked after simulation is complete. This assertion only executes associated statements if the assertion succeeds [24].

An immediate assertion can be used only within a procedural block in the SystemVerilog source code. The concurrent assertion statement is more flexible. It can also be used within a module, interface or program component. When a concurrent assertion is declared outside a procedural block, it behaves like it's declaration was preceded by the *always* keyword. This causes the property to be continuously evaluated throughout the simulation of the design. When a concurrent assertion is placed within a procedural block, it's evaluation is affected by the characteristics of that block, in particular the manner in which that block is clocked [24].

A similar technique for verification using assertions is described in [30]. In this case the LISA instruction set architecture language is used rather than SystemVerilog and the assertions are coded using the OVA and SVA languages (OpenVera compatible). The assertions are defined at the highest level of abstraction during the design process for consistent verification throughout the design process.

3.3 SystemC

SystemC is a system specification and simulation tool based on the C++ programming language that includes a tool set for the functional verification of both hardware-only and hardware/software co-designed systems [10,23,31, 32]. Like SystemC itself, the verification tool set is a class library for C++.

The SystemC Verification Standard consists of a number of classes and templates that support the following operations [23]:

- *transaction-based verification* - Systems can be verified at any level of abstraction that is executable by defining the beginning and end of a transaction so that the results of input data can be compared to output values for all executions of the simulation at various levels of abstraction.

- *data introspection* - The ability to inspect, change and randomize data values within a SystemC model at runtime. This is applicable for any data type supported by SystemC. The inspection can recover type and size information as well as the actual data value (or values for a composite type).

- *transaction recording* - information regarding a transaction can be recorded for analysis. The actual format of the recorded information is left to the implementor. The open source reference distribution, for example, outputs plain text.

- *data randomization* - any data item can be randomized provided that it is of a SystemC supported type except strings. This randomization can be unbounded, constrained or weighted (not supported by the reference implementation).

- *HDL connection* - a means of connecting a SystemC model to a model in VHDL or Verilog permits the simulation of designs that include both SystemC and one of these hardware design languages.
- *assertions* - boolean expressions used to detect erroneous conditions within a design. This feature is not included in version 1.0e of the Standard.
- *exceptions* - The Verification Standard includes an exception handling facility to deal with unusual but not erroneous conditions [23, 31].

Two important terms used throughout the remainder of this section are transaction and transactor. The SystemC Verification Standard provides the following definitions for them on page 4 [23]:

> transaction: A set of information that represents some bounded activity within the execution of a design or testbench. A transaction has a begin-time, an end-time, and a set of attributes (name-value pairs).
>
> transactor: An adaptor between a transaction-level test and a design typically at a different level. It is also referred to as a transaction verification model (TVM).

Transaction model verification permits the development of a single testbench program in SystemC that can be used throughout the verification process of the system. By defining the testbench at the highest level of abstraction testing of the system at that level and all lower levels of abstraction can use the same test program with the addition of appropriate transactors as necessary. This enables consistency in testing to be achieved with minimal effort on the part of the test team. Transactors act as the bridge between the software-based transaction testbenches and the hardware structure of a system model.

```
#include "scv.h"
#ifndef __PATH_PT_H
#define __PATH_PT_H
 struct path_point {
    int     elevation;
    int     distance;
    sc_int<8>   obstruction;
    sc_int<16>  obs_height;
    sc_int<8>   terrain;
    long        freznels[3];
    string      desc;
};
#endif
```

Figure 3: Header file path_pt.h containing the basic description of a set of data items that are to be examined using introspection.

The key component of the entire SystemC Verification Standard is introspection. This set of classes, templates and macros provides the functionality to access and change data within the system. The ability to record transaction information and randomize data is possible because of this functionality. The basic introspection services can be implemented using the SCV_EXTENSIONS macro that creates a C++ class containing the introspection capabilities for the class with which it is associated. Additional macros and method definitions are used to set up portions of this class such as constructors and the handling of composite data types [23].

An example of the way in which simple introspection can be used is shown in Figures 3 through 6. Figure 3 shows a header file that defines a data structure using standard C and SystemC language constructs and types. Note the inclusion of the *scv.h* header file within this header file providing access to the SystemC and SystemC Verification Standard data types, macros and classes. This data structure is representative of one that could be used in an application that determines the availability of a microwave radio path between two points on the surface of the Earth.

Figure 4 contains the header file *path_pt_ext.h* that contains the definition of the extensions to the data structure shown in Figure 3. The code in this file uses template classes to create the introspection extensions in-

```
#ifndef __PATH_PT_EXT_H
#define __PATH_PT_EXT_H
#include "path_pt.h"
template<> class scv_extensions<path_point>: public
scv_extensions_base<path_point> {
public:
    scv_extensions<int >        elevation;
    scv_extensions<int >        distance;
    scv_extensions<sc_int<8> >  obstruction;
    scv_extensions<sc_int<16> > obs_height;
    scv_extensions<sc_int<8> >  terrain;
    scv_extensions<long [3] >   freznels;
    scv_extensions<string >     desc;
    SCV_EXTENSIONS_CTOR(path_point){
        SCV_FIELD(elevation);
        SCV_FIELD(distance);
        SCV_FIELD(obstruction);
        SCV_FIELD(obs_height);
        SCV_FIELD(terrain);
        SCV_FIELD(freznels);
        SCV_FIELD(desc);
    }
};
#endif
```

Figure 4: Header file path_pt_ext.h containing the definition of the introspection facilities for the path_point structure.

stead of the SCV_EXTEN-SIONS macro presented earlier. The template instantiation creates a new class named *scv_extensions* that contains definitions of the extensions to the individual elements of the path_point structure. These extensions are created by using the pre-defined extension classes for the data types of those elements. The SCV_EXTENSIONS_CTOR macro is then used to create a constructor for the new class. The constructor uses the SCV_FIELD macro to denote each of the elements as part of a structure known to the class.

Figure 5 presents the main program for the example. All SystemC programs must contain a function named *sc_main* which is the entry point for the program. In this simple example the s*c_main* function performs all of the

```
#include "path_pt_ext.h"
int sc_main(int argc, char **argv) {
    scv_smart_ptr<path_point> path_p("Point");
    path_p->elevation = 100;
    path_p->distance = 1000;
    path_p->obstruction = 10;
    path_p->obs_height = 20;
    path_p->terrain = 9;
    path_p->freznels[0] = 20;
    path_p->freznels[1] = 55;
    path_p->freznels[2] = 5;
    path_p->desc = "Test Path Point";
    scv_out << "Printing Test Object:  " << path_p->get_name() << endl;
    path_p->print();
    return 0;
}
```

Figure 5: Source file path_pt_test.cc containing the program for the example.

operations of the program. A smart pointer to an instance of the path_point class is created with name *Point*. The use of smart pointers provides a consistent way of accessing objects of different classes within the SystemC environment. Following the creation of the class instance, values are assigned to all of its data elements. Once the values are set, some of the introspection methods of the path_point class are used to print out the label assigned to the class instance (*get_name()*) and all of the data element values (*print()*). Figure 6 shows the output produced by this example.

The above example is very simple but shows some of the available information from the introspection facility. The *scv_extensions* template class provides access to many different types of information about a set of data elements. This information can be obtained as necessary for the type of analysis being performed on the data.

The SystemC Verification Standard supports the randomization of all data types except strings. This includes the C data types, the SystemC data types and user defined data types. SystemC uses its own randomization techniques and permits three different styles of randomization. One technique is similar to the standard C *rand()* function except that any supported data type can

```
             SystemC 2.0.1 --- Jan 31 2006 09:57:42
        Copyright (c) 1996-2002 by all Contributors
                    ALL RIGHTS RESERVED
Printing Test Object:  Point {
  elevation:100
  distance:1000
  obstruction:10
  obs_height:20
  terrain:9
  freznels {
    [0]:20
    [1]:55
    [2]:5
  }
  desc:Test Path Point
}
```

Figure 6: Example output.

be randomized over its entire range of values [33]. The resulting random sequence is uniformly distributed. The second technique is also uniformly distributed but allows an upper and lower bound to be set on the generated values. It also permits the exclusion of ranges of values within the bounded range. The third technique uses user-defined distribution to generate the random values. The distribution of values is defined in a data structure referred to as a *bag*. Each item placed in a bag comprises a pair $\{value, percentage\}$ or $\{value\_pair, percentage\}$. The first pair format specifies that discrete values are to occur with the probability represented by the percentage in the second part of the pair. The second format specifies that values within a specific range are to occur with the given probability [23, 31].

Figure 7 shows a modified version of the program shown in Figure 6 that uses the SystemC randomization facility. The first step in using randomization is to provide a seed value for the random generator. In SystemC this is a global setting [23]. A constant seed value should always be provided for the SystemC randomization facility to ensure repeatability of verification simulations. Not providing a seed, or using the system time as is common in software systems eliminates repeatability since the seed value is either unknown or guaranteed to be different each time the simulation is executed [23].

The next step is to set up the randomization method to be used for each data item. In the example, fields *elevation*, *obs_height* and *terrain* use bounded uniform distribution; *obstruction* uses user-defined distribution with discrete values; *freznels* and *desc* have randomization disabled; and *distance* uses unbounded uniform distribution by default. The *desc* field must have randomization disabled as it is of type string. Each time a new set of random values is required for the fields the *next()* method is called on the scv_extensions object. Figure 8 shows the output produced by the program.

Transactions in SystemC permit the user to group operations together in order to monitor the results of those operations as a whole. Transaction recording is performed by writing transaction details to a database. The reference version of the SystemC Verification Standard will only output to a text file, other implementations may use their own output formats as well [23]. Within the SystemC Verification Standard document and in the examples provided with the open source version of the reference implementation there is a pipelined read/write example [23,34]. Figure 9 shows some of the output written to a text database file. This sample output was produced using a computer with a 600 MHz Pentium processor and 256 MB of memory running Linux.

The first four blocks of information in the output shown in Figure 9 show the setting up of the transaction monitoring and control functionality defined for the program. Following that, are three complete transactions. Transactions 2 and 3 are nested within transaction 1. This permits the measurement of the amount of time taken for specific parts of transaction 1 (i.e. transactions 2 and 3) and the overall time taken to perform the entire operation (transaction 1). This format of transaction recording does not include specific data values; they can be gathered using normal program I/O techniques as included in the sample program. Transaction control is normally placed within a transactor module so that monitoring can be performed on the interface between hardware and software. This is especially important when the software is purely a testbench and will not appear in the final implementation of the design.

Constraints are another important feature of the SystemC Verification

```
#include "path_pt_ext.h"

int sc_main(int argc, char **argv) {
    scv_smart_ptr<path_point> path_p("Point");
    scv_random::set_global_seed(101);
    path_p->freznels[0] = 0;
    path_p->freznels[1] = 1;
    path_p->freznels[2] = 2;
    path_p->desc = "Test Path Point";
    path_p->elevation.keep_only(0,59);
    scv_bag<sc_int<8> > obs_dist("Obstruction_Dist");
    obs_dist.add(0,20);
    obs_dist.add(1,25);
    obs_dist.add(2,15);
    obs_dist.add(3.40);
    path_p->obstruction.set_mode(obs_dist);
    path_p->obs_height.keep_only(0,59);
    path_p->terrain.keep_only(1,64);
    path_p->freznels[0].disable_randomization();
    path_p->freznels[1].disable_randomization();
    path_p->freznels[2].disable_randomization();
    path_p->desc.disable_randomization();
    path_p->next();
    scv_out << "Printing Test Object:  " << path_p->get_name() << " (random)" << endl;
    path_p->print();
    return 0;
}
```

Figure 7: File path_pt_test.cc using random value generation for some of the data elements of path_point.

```
              SystemC 2.0.1 --- Jan 31 2006 09:57:42
         Copyright (c) 1996-2002 by all Contributors
                    ALL RIGHTS RESERVED
Printing Test Object:  Point (random) {
  elevation:46
  distance:1077938815
  obstruction:2
  obs_height:56
  terrain:2
  freznels {
    [0]:0
    [1]:1
    [2]:2
  }
  desc:Test Path Point
}
```

Figure 8: Sample output of the program in Figure 7.

Standard. In the earlier example on randomization (Figure 7), two forms of constraints were used although not discussed as such there. The *keep_only* method constrains the range of values that can be randomly generated for a data element. The *sc_bag* class can be used to constrain the generation of random values for a specific data element to certain discrete values or ranges of values using probabilities of occurrence.

Another use for constraints in the SystemC Verification Standard is to ensure that necessary relationships between certain data values or sets of data values within a design are maintained. This is achieved through the use of sub-classes of the *scv_contraint_base* class. Instances of these subclasses are associated with specific data elements and are used to check constraint relationships as necessary [23].

The HDL connection facility of the SystemC Verification Standard provides a basic set of functionality for interfacing with models written in VHDL or Verilog. This interface permits the use of SystemC testbenches with models written in one of these hardware description languages. There are limits on the level of access that a SystemC model and a HDL model can interact with each other. For example, a SystemC model cannot access memory

```
scv_tr_stream (ID 1, name "pipelined_stream", kind "transactor")
scv_tr_stream (ID 2, name "addr_stream", kind "transactor")
scv_tr_stream (ID 3, name "data_stream", kind "transactor")
scv_tr_generator (ID 4, name "read", scv_tr_stream 1,
begin_attribute (ID 0, name "addr", type "UNSIGNED")
end_attribute (ID 1, name "data", type "UNSIGNED")
)
scv_tr_generator (ID 5, name "write", scv_tr_stream 1,
begin_attribute (ID 0, name "addr", type "UNSIGNED")
end_attribute (ID 1, name "data", type "UNSIGNED")
)
scv_tr_generator (ID 6, name "addr", scv_tr_stream 2,
begin_attribute (ID 0, name "addr", type "UNSIGNED")
)
scv_tr_generator (ID 7, name "data", scv_tr_stream 3,
begin_attribute (ID 0, name "data", type "UNSIGNED")
)
tx_begin 1 4 0 s
a 0
tx_begin 2 6 0 s
a 0
tx_relation "addr_phase" 2 1
tx_end 2 6 100 ns
tx_begin 3 7 100 ns
a UNDEFINED
tx_relation "data_phase" 3 1
tx_end 3 7 260 ns
tx_end 1 4 260 ns
```

Figure 9: Portion of the transaction information recorded in the database file for the pipelined read/write example program [23].

blocks in a Verilog model directly [23].

While SystemC version 2.0.1 contains an exception handling facility, the designers of the SystemC Verification Standard found that it was inadequate for their purposes. They provided a new exception handling system which will replace the original SystemC method in a later release [23]. The exception handler in the SystemC Verification Standard includes multiple levels of severity,user configurable actions when an exception occurs and multiple process logging support. This provides a very flexible methodology for reporting bugs detected within an executing simulation of a design [23].

An example of the incorporation of the SystemC Verification Standard into commercial tools can be found in [32]. In this example, a popular embedded processor architecture is modeled in SystemC with SystemC Verification Standard facilities included in the model. The testbench programs to evaluate designs utilizing this model can thus use these facilites to gather data for verification.

An alternate approach for functional co-verification using SystemC uses the tracing facilities of SystemC without the need for the SystemC Verification Standard [35]. In this technique, the hardware and software parts of a co-designed system are both implemented in SystemC and the appropriate tracing functionality within SystemC is enabled to gather the necessary data for verifying the design. Depending upon the level of abstraction of the design at any given point during its development, timing accuracy may or may not be achieved. Software execution can be done in one of two ways, as a standard C++ program running within the SystemC environment or as target processor instructions running on an instruction set simulator. The first approach provides the fastest simulation while the second provides a cycle accurate timing simulation but takes longer to execute [35].

3.4 Using Multiple Co-verification Tools

There are some cases where the use of a single co-verification tool is not sufficient for the design being checked. An example of such a situation is described in [36] when dealing with a system incorporating a USB 2.0 communications link. In this case, the authors created a tool called SSDE that provides a functional verfication flow using all of the tools necessary to perform the co-verification. All of the tools they use are commercially available but none of them provides the fill set of services required [36].

4 Conclusion

This paper has given an overview of the state of formal and functional verification in research and practice. It has shown that while formal verification of hardware/software co-designed systems is desirable, there are very few tools available in industry to do so. Research in the area of formal verification is progressing and will provide the necessary tools in the near future.

References

[1] S. Shukla, T. Bultan, and C. Heitmeyer. Panel: given that hardware verification has been an uphill battle, what is the future of software verification? In *Proceedings. Second ACM and IEEE International Conference onFormal Methods and Models for Co-Design*, pages 157 – 158, June23-25 2004.

[2] H. Jain, D. Kroening, and E. Clarke. Verification of specc using predicate abstraction. In *Proceedings. Second ACM and IEEE International Conference onFormal Methods and Models for Co-Design*, pages 7 – 16, June23-25 2004.

[3] T. Bultan, C. Heitmeyer, and J. O'Leary. Panel on design for verication. In *Proceedings. Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 232 – 235, July11 - 14 2005.

[4] W. Gardner. *CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications.* PhD thesis, University of Victoria, 2000.

[5] I. Somerville. *Software Engineering, seventh Edition.* Addison-Wesley, 2004.

[6] E. Clarke, O. Grumberg, M. Talupur, and D. Wang. High level verification of control intensive systems using predicate abstraction. In *Proceedings, First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03*, pages 55 – 64, June24 - 26 2003.

[7] R. Bryant. System modeling and verification with uclid. In *Proceedings. Second ACM and IEEE International Conference onFormal Methods and Models for Co-Design*, pages 3 – 4, June23-25 2004.

[8] T. Schuele and K. Schneider. Bounded model checking of infinite state systems: exploiting the automata hierarchy. In *Proceedings. Second ACM and IEEE International Conference onFormal Methods and Models for Co-Design*, pages 17 – 26, June23-25 2004.

[9] P. Jain. Cost-effective co-verification using rtl-accurate c models. In *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, pages 460 – 463, May30 - June2 1999.

[10] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual Revision 1.0*, 2003. Cited 10 Jan 2006, Available at www.systemc.org.

[11] Open SystemC Initiative. *SystemC Version 2.0 Users Guide Update for SystemC 2.0.1*, 2002. Cited 10 Jan 2006, Available at www.systemc.org.

[12] R. Domer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual Version 2.0*, 2002. Cited 3 Apr 2005 , Available at www.specc.org.

[13] J. Hammes, B. Rinker, W. Bohm, W. Naijar, and B. Draper. Cameron: High level language compilation for reconfigurable systems. In *Proceedings, 1999 International-Conference on Architectures and Compilation Techniques*, pages 236 – 244, Newport Beach, CA, U.S.A., Oct.12 - 16 1999.

[14] A. Ray. Security check: A formal yet practical framework for secure software architecture. In *New Security Paradigms Workshop 2003*, pages 59 – 65, Ascona Switzerland, 2003.

[15] D. Kroening and N. Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *Proceedings. Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 101 – 110, July11 - 14 2005.

[16] M. Fujita. Formal verification of c language based vlsi designs. In *Proceedings. 17th International Conference on VLSI Design*, pages 93 – 100, 2004.

[17] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis (technical report 2004-14), 2004. Cited 18 March 2006, Available at http://research.microsoft.com/slam.

[18] D. Grosse and R. Drechsler. Checkers for systemc designs. In *Proceedings. Second ACM and IEEE International Conference onFormal Methods and Models for Co-Design*, pages 171 – 178, June23-25 2004.

[19] D. Cachera and K. Morin-Allory. Verification of control properties in the polyhedral model. In *Proceedings, First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03*, pages 265 – 274, June24 - 26 2003.

[20] M. Contensin and L. Pierre. Combining acl2 and a /spl nu/-calculus model-checker to verify system-level designs. In *Proceedings, First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03*, pages 75 – 84, June24 - 26 2003.

[21] W. Gardner. Bridging csp and c++ with selective formalism and executable specifications. In *Proceedings, First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03*, pages 237 – 245, June24 - 26 2003.

[22] A. Gerstlauer, R. Domer, J. Peng, and D. Gajski. *SYSTEM DESIGN A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[23] O. S. Initiative. *SystemC Verification Standard Specification Version 1.0e*, 2002. Cited 10 Jan 2006, Available at www.systemc.org.

[24] Accellera Orgainization, Inc. *SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog*, 2004. Cited 16 Dec. 2005, Available at www.accellera.org.

[25] S. Sutherland, S. Davidmann, and P. Flake. *SystemVerilag For Design, A Guide to Using SystemVerilog for Hardware Design and Modeling.* Kluwer Academic Publishers, 2004.

[26] D. Thomas and P. Moorby. *The Verilog Hardware Description Language, Fifth Edition.* Kluwer Academic Publishers, 2002.

[27] S. Abdi and D. Gajski. Formal verification of specification partitioning, Mar. 2003. Technical Report CECS-TR-03-06.

[28] Synopsys Inc. *OpenVera Lanuage Reference Manual: Assertions Version 1.4.1*, 2004. Cited 5 Feb. 2006, Available at www.open-vera.com.

[29] Synopsys Inc. *OpenVera Lanuage Reference Manual: Testbench Version 1.4.3*, 2006. Cited 26 March 2006, Available at www.open-vera.com.

[30] A. Chattopadhyay. Integrated verification approach during adl-driven processor design. In *to appear in Proceeding Seventeenth IEEE International Workshop on Rapid System Prototyping*, 2006.

[31] C. N. Ip and S. Swan. A tutorial introduction on the new systemc verification standard, 2003. Cited 26 January 2006, Available at www.systemc.org with the open source distribution of the SystemC Verification Standard.

[32] F. Carbognani, C. Lennard, C. Ip, A. Cochrane, and P. Bates. Qualifying precision of abstract systemc models using the systemc verification standard. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 88 – 94, 2003.

[33] S. Harbison and G. Steele. *C A Reference Manual, Third Edition.* Prentice Hall, 1991.

[34] O. S. Initiative. Systemc verification standard open source reference implementation, 2002. Cited 10 Jan 2006, Available at www.systemc.org.

[35] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in c/c++. In *Proceedings of the 2000 conference on Asia South Pacific design automation*, pages 405 – 408, Yokohama, Japan, June 2000.

[36] T.-F. Omnes, G. Postuma, J. VerHaegh, M. Boonen, and N. Gatherer. Using ssde for usb2.0 conformance co-verification. In *Proceedings, First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03*, pages 113 – 122, June24 - 26 2003.