Cross-Product Functional Coverage Measurement with Temporal Properties-based Assertions

Avi Ziv
IBM Haifa Research Laboratory
Haifa University Campus
Haifa, 31905
Israel
email: aziv@il.ibm.com

Abstract

Temporal specification languages provide an efficient way to express events comprised of complex temporal scenarios. Assertions based on these languages are used to detect violations of the specification and monitor coverage events. In this paper, we propose to extend temporal specification languages, and assertions based on these languages with auxiliary variables. We attach these variables to sub-expressions and assign them values when the sub-expressions are evaluated. The use of auxiliary variables enables the implementation of large cross-product coverage models, using small number of assertions. This simplifies the definition and implementation of coverage models and helps reduce the simulation overhead caused by assertions, thus increasing the efficiency of simulation resources.

1 Introduction

Functional verification is widely acknowledged as the bottleneck of the hardware design cycle [4]. In current designs, up to 70% of the design development time and resources are spent on functional verification. In current industrial practice, simulation is the main vehicle for functional verification. Two of the main issues verification teams face are: how to check that the design behaves according to its specification and how to make sure the testing has been thorough.

Assertions (or checkers) are a traditional part of simulation environments. They are used to monitor the behavior of the design under test, and detect when its behavior deviates from the design's specification. Assertions assist in the detection of errors by monitoring the internal behavior of the design. They also help in the analysis of errors by reporting them as soon as they occur.

In recent years, assertion-based verification techniques that use temporal specification languages, such as Sugar [3] and ForSpec [2], have become popular. These languages, which are based on temporal logic [5], allow users to specify assertions for complex temporal scenarios. For example, a command sent to the storage control unit is completed after three to five cycles, unless it is an illegal command that completes after one cycle. There are several tools that implement assertions from properties specified in temporal specification languages [1, 9]. These tools translate properties into routines and data structures (usually state-machines) that are integrated into the simulation environment, monitor the simulation on a cycle-by-cycle basis, and report when a property is violated.

Coverage [7] is a recognized technique for checking and showing that the testing has been thorough. The idea of coverage is to create, in a systematic fashion, a large and comprehensive list of tasks and check that each task was covered during the testing phase. Assertions are commonly used to detect coverage events [1]. In this case, the assertions are responsible for detecting interesting legal events. Whenever an assertion detects such an event, it records the event in the coverage database.

In many cases, instead of coverage of a single event, we are interested in coverage of a coverage model, which comprises a family of events, specified as parameters of a basic event. In these cases, we are usually interested in coverage of the cross-product of the parameters [6]. For example, we may want to see that all possible commands have been sent to the storage control unit and that for each command all possible responses have been received.

One possible way to implement a cross-product coverage model with assertions is to build an assertion for each possible coverage task. The problem with this approach is that it requires many assertions. For example, if there are 50 possible commands, with 20 possible responses, then 1000

1530-1591/03 $17.00

assertions are needed. Therefore, this solution can significantly slow down the simulation.

In this paper, we describe a technique that implements cross-product functional coverage with a small number of assertions. In this new technique, we specify the parameters of the cross-product coverage as auxiliary variables that are attached to sub-expressions in the temporal expression. The auxiliary variables are assigned values when the sub-expressions to which they are attached are evaluated as true. When the full temporal expression becomes true, the values of the auxiliary variables are reported as a covered task. This technique allows us to use a single assertion or a small number of assertions for a coverage model with many coverage tasks, instead of using a single assertion for each coverage task. For example, to implement the command response model above, we attach an auxiliary variable that stores the type of command to the sub-expression that detects a new command and another auxiliary variable that stores the response type to the sub-expression that detects the corresponding response. When a command response pair is detected during simulation, the values stored in the auxiliary variables are reported as a covered task.

In addition to the auxiliary variables, we also introduce a new coverage operator *C*. This new operator detects multiple occurrences of an event and reports them as separate coverage tasks. Note, Sugar 2.0 proposes a `coverage` directive. The *C* operator described here provides a possible implementation of this directive.

We added the auxiliary variables and the new coverage operator to a checking tool that is based on temporal logic from the University of Tuebingen [8] and used it to measure coverage for several coverage models of a storage subsystem of a multiprocessor. We compared the simulation overhead of a large set of simple assertions, one for each coverage task, to the overhead of a small number of assertions with auxiliary variables. The results show that the use of auxiliary variables can reduce the checking overhead by an average factor of 10, significantly increasing the efficiency of the simulation.

The rest of the paper is organized as follows. In Section 2, we provide a short background on assertions that are based on temporal specification languages and cross-product functional coverage. In Section 3, we show how assertions can be used as coverage monitors. Section 4 introduces the auxiliary variables and explains how they can be used to enhance the capabilities of assertions for cross-product coverage. In Section 5, we describe how we implemented the auxiliary variables and present some experimental results. Finally, Section 6 concludes the paper.

2 Background

2.1 Assertions Based on Temporal Properties

Temporal specification languages [3] allow users to specify properties that involve complex temporal scenarios. Checking tools, such as [1] and [9], translate these properties into assertions, which are integrated into the simulation environment; the assertions monitor the simulation on a cycle-by-cycle basis and report when a property is violated.

This paper refers to basic *Linear-time Temporal Logic* (LTL) [5], which is the basis for most temporal specification languages, rather than a particular specification language. This is done to simplify the description of the temporal logic and highlight our new ideas, which are not unique to a specific language.

Temporal logic provides operators that define temporal relations between operands. LTL has four basic temporal operators:

X: the next operator $X(expr)$ is true if *expr* is true in the following cycle.

F: the eventually operator $F(expr)$ is true if *expr* becomes true sometime in the future.

G: the always operator $G(expr)$ is true if *expr* is true in all cycles (current and future).

U: the until operator $U(expr_1, expr_2)$ is true if $expr_2$ becomes true sometime in the future and $expr_1$ is true until that time.

There are finite interval versions of these operators that limit the scope of the future [8]. These finite operators are shorthand for many *X* operations, but they provide a convenient way to specify time bounded events. For example, specifying that a `command` must be followed by a `response` within one to five cycles, can be very easily specified using the formula $\texttt{Cmd} \rightarrow F[1,5](\texttt{Res})$.

Since LTL expressions refer to the infinite future, they cannot always be evaluated during simulation that takes a finite time. For example, the temporal expression $G(expr)$, which means that *expr* is always true, cannot become true during simulation. Even if *expr* is true in every cycle during the simulation, there is no guaranty that it will not become false after the simulation ends. On the other hand, if *expr* becomes false for a single cycle, $G(expr)$ becomes false at the same time. In a similar way, $F(expr)$ can never become false in finite simulation, since *expr* can become true after the simulation ends, but $F(expr)$ can become true if *expr* becomes true during the simulation.

Therefore, assertions based on temporal specification languages can be used to detect violations of safety properties that are expressed as $G(expr)$ expressions. They can

also be used to detect interesting events that are expressed as $F(expr)$ expressions. On the other hand, they cannot be used to detect violations of liveness properties.

2.2 Functional Coverage

Coverage is the main technique for checking and showing that testing has been thorough [7]. Coverage, in general, can be divided into two types: code-based and functional [10]. Functional coverage focuses on the functionality of the design. It is used to check that all important aspects of the design's functionality have been tested. Functional coverage models can be built using a list of single events or from a family of events with some common denominator. Often, models of the latter type are defined as the *cross-product* of the values of a given set of attributes.

A cross-product functional coverage model can be constructed in the following manner [6]. We start by creating a semantic description (story) of the model, to describe the type of events we want to cover. The story contains a set of attributes that become the attributes of the cross-product coverage model. For each attribute, we define the set of all possible values it can receive. Finally, we define a list of restrictions that describe the legal combinations in the cross-product of the attribute values.

The storage subsystem of a multiprocessor, described in Figure 1, illustrates how cross-product coverage models are used. We used a model of the same subsystem to measure the performance of our checking tool (see Section 5.1). The storage subsystem is built of a *Storage Control Unit* (SCU) that is connected to four processors, CP0 – CP3, and a main memory. Each CP can send commands to the SCU. The SCU handles these commands, either locally or, using the main memory. When the handling of a command is completed, the SCU sends a response to the requesting CP. Each CP has four internal sources of commands; each of them generates its own commands independently. To distinguish between commands from different sources in the CP, the command line from the CP to the SCU contains two bits that identify the source of the command. The response from the SCU to the CP also contains the same source bits. Note that a CP can have several commands pending in the SCU, however, an internal source cannot send new commands before it receives a response to the last command it sent.

An example of a coverage model for the storage subsystem is the *Command-Response* model that looks at all the command response combinations for each of the CPs in the system and at each internal source in each CP. The attributes of this model and the possible values for each attribute are shown in Table 1. A CP can send one of four commands to the storage control unit, Instruction fetch (IF), Data Fetch (DF), Data Store (DS), and Read-Modify-Write (RMW). It can also send an illegal command (ILL). In response, the storage control unit sends the requesting CP an ACK to indicate that the command was executed, a NACK to indicate that it was not executed, or an ERROR to indicate that an error occurred. The number of tasks in this cross-product coverage model is $4 \times 4 \times 5 \times 3 = 240$ (number of CPs $\times$ number of sources in each CP $\times$ number of possible commands $\times$ number of possible responses). However, not all the tasks in the model are legal. For example, the response for an illegal command must be ERROR, while the response to a legal command cannot be ERROR. Considering this and other restrictions, the number of legal tasks in the model is 112.

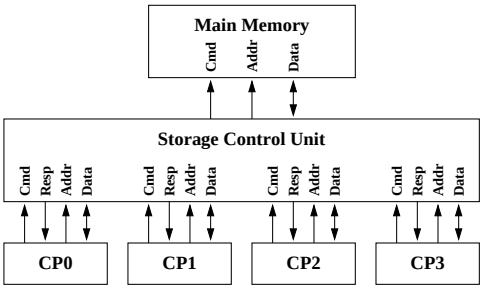

Figure 1. Storage subsystem structure

Attribute	Values
CP	0, 1, 2, 3
Source	0, 1, 2, 3
Command	IF, DF, DS, RMW, ILL
Response	ACK, NACK, ERROR

Table 1. Attributes of the Command-Response model

3 Using Assertions for Coverage Measurement

The goal of coverage monitors is to detect coverage tasks that occur during the simulation. Therefore, to specify a coverage task e as an assertion, we can use the expression $F(e)$. For example, if we want to cover the event that a command was answered after 1 – 5 cycles with a NACK response, the coverage event is

$$\texttt{Cmd} \wedge F[1,5](\texttt{Res} \wedge \texttt{ResType} = \texttt{NACK}).$$

We can use the assertion

$$F(\texttt{Cmd} \wedge F[1,5](\texttt{Res} \wedge \texttt{ResType} = \texttt{NACK}))$$

and mark the coverage task as covered if the expression becomes true during simulation.

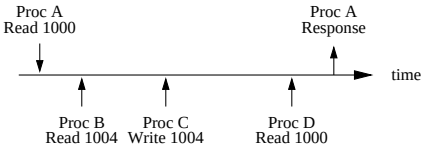

Figure 2. Ambiguous counting example

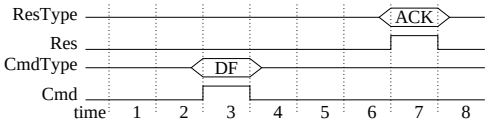

Figure 3. Example of a simple timing diagram

In many cases, coverage monitors need to count how many times a coverage task occurred during simulation, not just detect whether or not it occurred. In this case, using the F operator for the coverage monitor is not sufficient. The F operator detects the first occurrence of the event, reports it to the user, and stops. To overcome this problem we introduce a new coverage operator C for assertions. $C(e)$ detects all the occurrences of the event e during simulation and counts how many times the event happened. Note, unlike other temporal operators that evaluate to a Boolean value (or a pending value if they cannot yet be evaluated), the C operator evaluates to an integer value. Therefore, the C operator can be used only as the top operator in the parse tree of a temporal expression.

Implementing the C operator can be a complicated issue. Counting the number of events that occur during simulation requires a much more accurate definition of the event. For example, consider the event that consists of a request sent from Processor A to a common bus, where the bus is interrupted by a request from another processor before it could send a response to Processor A. During simulation we observed the sequence in Figure 2. One possible interpretation is that each interruption corresponds to a different coverage event, and therefore the sequence represents three separate events. Another possible interpretation is to associate the request from processor A with all the interruptions. In this case, the sequence represents a single event.

To simplify the implementation of the C operator, we assume that in each cycle, at most one instance of an event can start. If overlapping events are not possible, then the C operator can be implemented by restarting the assertion (with the F expression) after the assertion becomes true. This solution will not work correctly if overlap between events is possible. When overlapping events can occur, we start an assertion for the coverage expression (without the C operator) every cycle. This solution seems to be very wasteful, but in practically all cases, the expression would fail immediately (in the example above, if `Cmd` is not set), therefore the resource burden is small.

4 Auxiliary Variables

When cross-product functional coverage is used, the number of coverage tasks that need to be covered can become very large. For example, the Command-Response coverage model in Figure 1 contains 240 tasks (of which only 112 are legal). Using a separate assertion for each coverage task can significantly slow down the simulation. If we examine the coverage model closely, we see that it consists of two parts: (1) a basic event that specifies that a command is sent and few cycles later a response is sent back; (2) parameters that are attached to the basic event, such as the type of command and the type of response. Therefore, instead of using many assertions, we can use a single assertion that detects the basic event and collects the parameters while the assertion operates. We specify the parameters of the cross-product coverage as auxiliary variables that are attached to sub-expressions in the temporal expression.

When a sub-expression is evaluated to true, all the auxiliary variables that are attached to the sub-expression are assigned values. When the full temporal expression becomes true, the values of the auxiliary variables are reported as a covered task. For example, if we consider a simple version of the Command-Response model that looks at a single CP and assumes that only a single internal source exists, we can use a single assertion to detect all the command-response pairs that occurred during simulation with the following coverage expression

$$C(\texttt{Cmd}\{\$CT \leftarrow \texttt{CmdType}\} \wedge F(\texttt{Res}\{\$RT \leftarrow \texttt{ResType}\})),$$

where $\$CT$ and $\$RT$ are auxiliary variables that are attached to `Cmd` and `Res`, respectively.

Figure 3 presents a simulation example. In the third cycle, the CP raised the `Cmd` signal and set `CmdType` to `DF`. This caused the sub expression `Cmd` to become true and the value in `CmdType` to be assigned to the auxiliary variable $\$CT$. Next, the checking tool waits for the second part of the expression, $F(\texttt{Res}\{\$RT \leftarrow \texttt{ResType}\})$, to become true as well. In the seventh cycle, the SCU raises `Res` and sets `ResType` to `ACK` to indicate that the command finished successfully. This causes the `Res` sub-expression to become true and therefore the `ACK` is stored in $\$RT$. This also causes the F sub-expression to become true, and thus, the full coverage expression is true. When the coverage expression becomes true, the checking tool reports the values of $\$CT$ and $\$RT$ (`DF` and `ACK`) to a coverage collection tool as a coverage task that was covered.

Another important role for auxiliary variables, besides collecting coverage parameters, is to connect between sub-expressions in the coverage expression. For example, in the

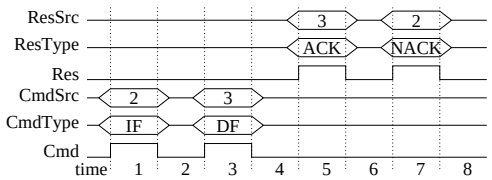

Figure 4. Example of overlapping events

storage subsystem of Figure 1, a CP can send new commands to the SCU before the last command is answered, and responses may arrive out of order. The CP uses the source bits to associate the command with its response. In this case, we can modify our coverage expression to do the same thing. That is, to look for the matching source bits in the response using an auxiliary variable. The expression will appear as follows:

$$C(\texttt{Cmd}\{\$CT \leftarrow \texttt{CmdType}, \$S \leftarrow \texttt{CmdSrc}\} \wedge F((\texttt{Res} \wedge \$S = \texttt{ResSrc})\{\$RT \leftarrow \texttt{ResType}\})). \quad (1)$$

Note, the second time $\$S$ appears, it is used as part of the expression, not as an auxiliary variable that has to be stored. Figure 4 shows a simulation example with overlapping events. The implementation of the C operator described earlier, means that each cycle a new assertion for the internal expression in (1) is started. The assertion that starts in the first cycle detects that the CP raised the `Cmd` signal and saves `IF` and 2 in the auxiliary variables $\$CT$ and $\$S$, respectively. The assertion that starts in the second cycle stops immediately, since the `Cmd` signal is not set. The assertion that starts in the third cycle detects that `Cmd` is set and saves `DF` and 3 in its auxiliary variables. During the fifth cycle both active assertions detect that `Res` is active. The first assertion ignores this response, since the value in `ResSrc` does not match the value stored in its $\$S$ auxiliary variable. The value in `ResSrc` does match $\$S$ of the second assertion, and therefore, its coverage expression becomes true and it reports `(DF, ACK, 3)` as a covered task. In the seventh cycle, `Res` is set again and this time `ResSrc` matches $\$S$ of the first assertion, which reports that `(IF, NACK, 2)` is covered.

5 Implementation of Auxiliary Variables

We implemented the coverage operator and the auxiliary variables as an extension to an LTL-based checking tool from the University of Tuebingen [8]. This tool is based on SystemC [11], a C++ class library for hardware description. The tool is implemented as a SystemC module that evaluates all pending assertions in every cycle during simulation and reports to the user each time an assertion becomes true or false. The algorithm used by the tool to evaluate temporal expressions is based on the recursive evaluation of the expression's sub-expressions. Since some sub-expressions may not be immediately evaluated to true or false, each expression maintains a list of sub-expressions with pending values that are evaluated in future cycles.

Since auxiliary variables are attached to sub-expressions, it is simple to support them in the tool. When a sub-expression becomes true, the tool assigns values to all the attached auxiliary variables. These values propagate to the parent sub-expression. When the parent becomes true, these values continue to propagate up until they reach the root of the expression parse tree. When the coverage operator is used as the root of the tree, the values of the auxiliary variables are reported to the coverage tool when the coverage expression becomes true. The coverage operator C is implemented by starting an assertion for the internal expression in each cycle and reporting the values of the auxiliary variables attached to the assertion when it becomes true. Assertions that are evaluated to false are ignored.

5.1 Experimental Results

We implemented three coverage models for the interface between the CPs and the Storage Control Unit in the storage subsystem described in Figure 1. We used these models to check how the checking tool operates with auxiliary variables. We also wanted to compare the performance of coverage monitors that use temporal expressions with auxiliary variables to coverage monitors that use a separate assertion for each coverage task. The three coverage models that we implemented are as follows:

Commands - checks that all combinations of commands from all four CPs were sent at the same cycle. Note, this coverage model is not temporal. The number of possible tasks in this model is $6^4 = 1296$, since every CP can send one of the five possible commands (IF, DF, DS, RMW, ILL) to the SCU, or not send any command. Since 1296 assertions slowed down the simulation considerably, we grouped the two fetch commands together and the two commands that modify the main memory together. This left us with $4^4 = 256$ tasks.

Command-Response - as described in Section 2.2. The model checks that all the possible command-response pairs appear during simulation for all the CPs in the system and for all sources in the CP. The total number of tasks in the model is $4 \times 4 \times 5 \times 3 = 240$ tasks, out of which only 112 are legal.

Out-of-Order - checks for all possible out-of-order executions of commands from the same CP. That is, a command that was sent from source s_i after a command that was sent from source s_j in the same CP is responded

Model Name	# Tasks (Legal)	Sim Overhead [sec]		Ratio
		Simple	Auxiliary	
Commands	256	206	4	51.5
Command-Response	240 (112)	225 (50)	9	25 (5.6)
Out-of-Order	576 (192)	711 (125)	21	33.9 (6)

Table 2. Performance comparison

before the earlier command. Figure 4 illustrates an example of an out-of-order event. After the same grouping described in the Commands model, the total number of tasks in the model is $4 \times 4^2 \times 3^2 = 576$, out of which only 192 are legal.

We used two alternative implementations for each model; one used a simple assertion for each coverage task in the model and the other used a small number of assertions with auxiliary variables. For all the models, the two implementations detected the same coverage tasks. We also compared the performance of the two alternative implementations in terms of the overhead in simulation time. For the second and third model, the measurements for the separate simple assertions were done twice — once when we used an assertion for each task in the model and once when we used assertions only for the legal tasks.

Table 2 shows the performance comparison information. For each model, the table shows the number of tasks in the model and the overhead in seconds per 10,000 cycles of simulation for the simple separate assertions and for small number of assertions that utilize the auxiliary variables. The numbers in parenthesis indicate the cases when only assertions for legal tasks are used. As the table shows, utilizing the auxiliary variables and reducing the number of assertions can reduce the simulation overhead by a factor of more than 5, and up to 50. In fact, using the auxiliary variables can make coverage model whose implementation is impractical (e.g., the Commands model without the grouping), and turn it into a coverage model that can be measured with a small overhead.

6 Conclusions

In this paper, we showed how assertions that are based on temporal specification language expressions can be used as monitors for the collection of functional coverage information. We also showed how the addition of auxiliary variables can significantly reduce the number of assertions used to collect coverage information for cross-product coverage models. Experimental measurements of the performance of the proposed assertions show that the use of auxiliary variables can lead to a large reduction in the simulation time overhead introduced by the assertions.

We are currently investigating several methods to enhance the capabilities of our checking tool and improve its performance. Specifically, we are working on an AR-automata based implementation of our checking tool. We are also trying to incorporate the auxiliary variables into Sugar [3], the Accellera EDA Standards Organization selection for a standard property language. This includes extensions to the language to provide improved ways to express the exact semantics of temporal expressions and allow accurate counting of events.

References

[1] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs - automatic generation of simulation checkers from formal specification. In *Proceedings of the 1999 Computer Aided Verification Conference*, July 1999.

[2] R. Armoni et al. The ForSpec temporal logic: A new temporal property-specification language. In J.-P. Katoen and P. Stevens, editors, *8th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2280, pages 296–311. Springer-Verlag, April 2002.

[3] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13^{th} International Conference on Computer Aided Verification (CAV)*, LNCS 2102, pages 363–368. Springer-Verlag, July 2001.

[4] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[5] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072. Elsevier Science Publishers, 1990.

[6] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Proceedings of the 35th Design Automation Conference*, pages 158–165, June 1998.

[7] B. Marick. *The Craft of Software Testing, Subsystem testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1985.

[8] J. Ruf, D. W. Hoffman, T. Kropf, and W. Rosenstiel. Checking temporal properties under simulation of executable system description. In *Proceedings of the International High Level Design Validation and Test Workshop*, pages 161–166, November 2000.

[9] J. Ruf, D. W. Hoffman, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued ar-automata. In *Proceedings of the 2001 Design, Automation and Test in Europe Conference (DATE)*, pages 742–748, March 2001.

[10] S. Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *proceedings of STAR98: the 7th international conference on software testing analysis and review*, May 1998.

[11] SystemC 2.0 user's guide. http://www.SystemC.org/.