ADAPTIVE TEST PROGRAM GENERATION: PLANNING FOR THE UNPLANNED

Allon Adir, IBM Research Laboratory in Haifa, adir@il.ibm.com
Roy Emek, IBM Research Laboratory in Haifa, emek@il.ibm.com
Eitan Marcus, IBM Research Laboratory in Haifa, marcus@il.ibm.com

Abstract

Simulation of automatically-generated test programs is the primary means for verifying complex hardware designs and random test program generators therefore play a major role in the verification process of micro-processors. The input for a test program generator is typically an abstract specification—a template—of the tests to be generated. Due to randomness, generators often encounter situations that were not anticipated when the test specification was written. In this paper, we introduce the concept of adaptive test program generation, which is designed to handle these unforeseen situations. We propose a technique that defines unexpected events together with their alternative program specifications. When an event is detected, its corresponding alternative specification is injected into the test program.

Introduction

The goal of processor functional verification is to ensure functional conformance of a processor design to its architectural and micro-architectural specifications. One way to achieve this goal is to write test programs that check the functionality of the processor and run them on the design or its simulator [6, 9]. These tests typically contain at least a sequence of instructions to be executed, and possibly initial values and expected result values for architectural resources used by the instructions. The test programs are run by setting the initial values of the design's resources and executing the instructions on a design simulator.

The current practice in the processor verification industry is to generate most test programs automatically with a pseudo-random test program generator [2, 3, 7, 8, 5]. The input to such a test program generator is a user-defined sequence of partially specified instructions outlining the scenario that should occur when the test program is run. This outline acts as a template for the generated test program. The test program generator generates the test according to the template, by filling in the missing details with valid random values. Random choices are often biased to improve the quality of the test program.

The first generation of test program generators were *static* in that they had no knowledge of the state of the verified design during the generation process. Later, *dynamic* test program generators were introduced; these use an architectural *reference-model* to track the state of the processor during the generation process. The reference-model provides feedback to the test program generator regarding the current value of architectural resources used in the test. The generator can then use this information to create more incisive test patterns that are more likely to expose design flaws (see Figure 1).

A significant consequence of randomness is that it can lead to situations that are difficult or impossible for users to anticipate when defining the input test program template. When such situations occur, users would like the generator to use an alternative template that exploits or reacts to the unanticipated situation. The process of detecting such situations and injecting alternative sequences of instructions into the test program is termed *adaptive test program generation*.

Examples for the use of adaptive test program generation include:

1. Handling unexpected interrupts caused by a generated instruction.
2. Avoiding unwanted situations from occurring in the test program. These can be detected in advance, and prevented or corrected after having occurred. An example of an unwanted situation is one in which a test program reaches a state that is not supported by the design in an early verification stage.
3. Exploiting rare situations that occur by chance during the test program run.

Previous techniques for handling unexpected interrupts or undesired situations were either to stop generation once the situation occurs or to handle them with special-purpose mechanisms built into the generator (as in the generators described in [5, 2]). The former method prevents the generation of long and complex test programs. The latter lacks the flexibility required to handle frequent design changes. In addition, both methods provide very limited user control over the generated test program, and are incapable of handling the case of rare situations exploitation.

We introduce a novel mechanism for adaptive test program generation that is more general and more flexible than existing approaches. This mechanism is based on the definition of a language construct — called an *Event* — as part

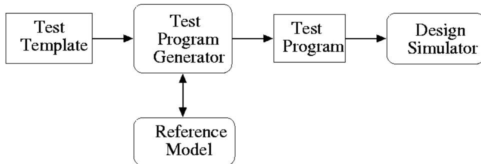

Figure 1. Test Program Generation Scheme

of the test program template language. An Event consists of a *condition* and a *response-template*. The generator refers to the condition in order to identify that the Event has occurred. When an Event's condition is satisfied, the generator interrupts normal operations, generates the Event's response template, and returns to the original test program template at the interruption point.

The Events can be defined separately from the specification of the scenario in the main test program template. This achieves a separation of concerns between the planning of the scenario to be tested and the handling of the unexpected situations that may occur in the scenario.

Adaptive test program generation is an integral part of *Genesys-Pro*, a state-of-the-art random test program generator developed at the IBM Research Laboratory in Haifa and used extensively by IBM for the verification of high-end processors.

The rest of this paper is organized as follows: The following section explains the relevant test program generation scheme. Then we expand on the event-response technique and provide examples of adaptive test program generation. The final section describes how adaptive test program generation is implemented in the Genesys-Pro generator.

Test Program Generation Scheme

The test program generation scheme assumed in this paper is shown in Figure 1. The input to the test program generator is a *test program template*—an abstract specification of tests to be generated. This template outlines the scenario that should occur in the test program. Typically, test program templates are incomplete in that various details of each instruction, such as the specific resources to be used (i.e., which registers or memory locations), the data for these resources, and even the exact instruction to be generated, may be left unspecified. The test program generator then generates a complete test by filling in the missing information of each instruction with random values. The choice of random values is often biased so as to increase the chance of finding bugs in the design.

The generator is assumed to generate the test program instructions sequentially in their program order[1]. In the dynamic test program generation scheme shown in Figure 1, the generator uses an architectural reference-model to track the state of the processor during the generation process. The reference-model provides feedback to the test program generator on the current value of the resources used in the test. The test program generator can then use this information to create more incisive test patterns, which are more likely to expose design flaws.

Finally, the generated test program is passed to a design simulator which runs the test program and looks for violations of the specification.

[1]This assumption can be relaxed (as seen in [1]) but is taken here to simplify the discussion.

Table 1. Test Template and Test

Test Program Template	Test Program
Instr: `Add R1+R2` $\longrightarrow *$	*Resource Initial Values:*
select target data = 0	`R1 = 4, R2 = -4`
Instr: `Store`	*Instructions:*
Instr: `Load`	`Add R1+R2` $\rightarrow$ `R5`
	`Store R5` $\rightarrow$ `1000`
	`Load R1` $\leftarrow$ `1000`

Test Program Templates

A simple test program template language may only allow a list of partially specified instructions. More sophisticated languages may include constructs for designing the structure of the test program and various types of constraints on properties of the test program's instructions and resources [4].

In the following discussion, we use the term *statement* to refer to a basic construct of the template language. Examples can include a request for a specific instruction, a sequence or repetition of partially specified instructions, and so forth. A test program template is formed by combining several such statements.

Consider the example of a test program template in Table 1. This template includes three instruction statements. The first statement specifies an `Add` instruction that uses `R1` and `R2` as input. The target register is left unspecified. The generator chose `R5` as the target register, as can be seen in the final test program shown in the table. The template also specifies that the result of the `Add` should be zero. The generator fulfilled this request by initializing `R1` with 4 and `R2` with -4. The other two instruction statements specify a `Store` instruction and a `Load` instruction. The generator chose the same address (i.e., 1000) for the Store and Load because it biases the test program towards memory reuse.

Event Statement

The main object of this paper is the introduction of a technique termed Adaptive Test Program Generation, which is carried out by means of a novel test program template language construct called an Event.

An Event consists of a *condition* and a *response-template*. After each instruction is generated, the conditions of all the defined Events are checked. Whenever the condition of an Event is satisfied, its response-template is used for generation. After the response-template has been generated,

Table 2. Test Program Template with an Event

Test Program Template	Test Program
Test Program Template Body:	*Resource Initial Values:*
Repeat 10 times	R2=12,R3=-7,R4=1
Instr: Add * + * → $R1$	*Instructions:*
Instr: Div * / R1 → *	Add R4 + 3 → R1
	Div R2 / R1 → R2
Event: Avoid-Zero-Divide	Add R3 + 7 → R1
condition: R1=0	**Move-To 5 → R1**
response :	Div R1 / R1 → R4
Instr: Move-To 5 → R1	Add R4 + 7 → R3
	:
	Div R3 / R2 → R2

the generator returns to the original test program template. This process is recursive, since the Event statement involves the generation of instructions which, in turn, may cause the condition of further *events* to be satisfied and the generation of their respective response-templates.

The *response-template* may be any statement of the test program template language. An Event *condition* may refer to:

- The *Machine state* as given by the reference-model used by the generator. The machine state may include the architectural or micro-architectural resources maintained by the reference-model, including for example, the current values of processor registers, memory, caches, etc.

- *Generation state and information*: Examples include the number and types of instructions generated so far, the registers that were used in the last generated instruction, or a reference to different settings that were made in the test program template.

Table 2 shows a simple example of a test program template with an Event. The main body of the template asks for 10 pairs of an Add instruction followed by a Div instruction. The Add instruction adds a register and an immediate value (left unspecified) into register R1. The Div is a divide instruction that should use R1 as the divisor. The template includes an Event named Avoid-Zero-Divide, which is to be activated whenever R1 becomes 0. Its response is to set R1 to 5 with a Move_To instruction. As the generated test program shows, the second Add puts the value 0 into R1. This caused the Event to be invoked, which caused the generator to add the Move_To instruction to the test program, setting R1 to 5. The instructions added by the event are shown in bold in this table and in the following tables given in the paper. Following this, the test program continues with the Div instruction of the interrupted pair, and then continues with the remaining 8 pairs of Add and Div instructions.

Uses of Adaptive Test Program Generation

This section gives some motivation for adaptive test generation by showing examples of situations that may occur during test program generation and can be aided by our proposed technique.

Architectural Interrupts

Users often want to generate an alternative set of instructions when an unexpected interrupt occurs. In some cases, the user may want to avoid a test program with an interrupt (perhaps because the verified design does not yet handle this case correctly) or to verify that the design behaves correctly when an interrupt occurs. This can be handled with an Event whose condition checks whether an interrupt occurred, and whose response-template includes statements for dealing with the interrupt.

In Table 3, the main template body asks for 10 Store instructions. Suppose that a Store instruction may cause an Illegal_Access interrupt, which in turn sets the program-counter to 0x300. The request for the Stores was given with no further specification, so that the random generation of a Store may lead to the interrupt.

With no adaptive test program generation, if one of the Stores causes an Illegal_Access interrupt, the generator will continue to generate the rest of the Stores from address 300. The Event appearing in Table 3 specifies that when an exception occurs and the program-counter equals 0x300 the generator should generate an Xor instruction, followed by a Return_From_Interrupt instruction, instead of Stores. This, in effect, specifies the instructions to generate for the Illegal_Access interrupt handler. After the handler is generated, the generator returns to generate Stores from the main template body. The test program in the table also gives the memory locations in which the generated instructions are placed by the generator, assuming that instruction opcodes take up 4 bytes.

With the Event response-templates, users writing test program templates have complete control over the manner in which different interrupt handlers are generated. There is also added flexibility in the use of a general Event condition relating to current resource values. This enables the generation of *relocatable interrupts* (i.e., interrupts whose handlers do not reside in pre-specified memory locations).

Avoiding Problem Situations

Events can be used to avoid or to recover from unwanted situations that may occur during generation. To avoid an undesired situation, the Event condition should identify that the unwanted state is *about to* occur and the response-template should include an "escape" action. The Event in Table 2 is an example of this type of Event. To recover from an unwanted state, the Event condition should identify that the unwanted state *has* occurred and the response-template should include a corrective action. If the generator has the capability to undo the last generated instruction, a typical corrective action would indicate to the generator that the instruction that caused the invocation of the Event should be

Table 3. Handling exceptions with an Event

Test Program Template	Test Program
Test Program Template Body:	*Instructions:*
Repeat 10 times	1000: `Store`
Instr: `Store`	1004: `Store`
	1008: `Store`
Event: Handle_Illegal_Access_Interrupt	300: **Xor**
condition:	304: **Return_From_Interrupt**
exception occurred and `PC=300`	100C: `Store`
response :	1010: `Store`
Instr: `Xor`	:
Instr: `Return_From_Interrupt`	1024: `Store`

Table 4. Escaping from unfetchable memory areas

Test Program Template	Test Program
Test Program Template Body:	*Instructions:*
Repeat 10 times	1000: `Store`
Instr: `Store`	1004: `Store`
	1008: `Store`
Event: Escape_Unfetchable_Memory	100C: **Branch** → **2000**
condition:	2000: `Store`
location `PC+4` is unfetchable	2004: `Store`
response :	:
Instr: `Branch to fetchable location`	2018: `Store`

undone.

The following list gives some important examples of cases where adaptive generation with Events can be used to avoid problematic states:

- **Avoid states unsupported by the design**: In early verification stages, test program generators must avoid situations not yet supported by the design. For example, the template in Table 2 can be used for a design that does not yet support the `Divide_By_Zero` interrupt.

- **Avoid instruction fetches from illegal locations**: Some memory locations are unsuitable for instruction fetches. Examples may include data-only segments or reserved memory areas. Another example that may occur during test program generation is that the generator cannot place a new instruction in a memory location already occupied by a previously placed instruction.

  When the generator generates the instructions from the template in program order, it normally places the generated instructions one after the other in adjacent memory locations. An exception is when a generated instruction causes a branch (either an explicit branch instruction or an implicit branch, such as an interrupt). In this case, the generator must start placing the generated instructions at the branch target location.

  As the generator generates instructions and places them in memory, it may reach an unsuitable location for fetches. The Event in Table 4 identifies that this situation is about to occur and avoids it by generating a branch to some valid location. The example assumes again that an instruction opcode takes up 4 bytes. Thus, PC+4 (PC being the program-counter register) points to the location where the PC is going to end up after the generation of the next instruction. If this points to an unfetchable location, the generator should generate the escaping branch before continuing with the generation of instructions from the main template body.

- **Avoid using up all available resources**: As part of an instruction's generation, the generator must select values for the resources used by the instruction. This way, the generator controls the behavior of the instruction. Registers are a limited resource—most architectures define several dozen general purpose registers. Thus, for the first instructions of the test program, the generator can easily find registers that have yet not been assigned values and use them in the new instructions, with whatever values the generator sees fit. Later on in the test program, if all the registers have already been used, the generator is limited to the values that currently exist in the registers. This may critically disable the generator's ability to control the behavior of the test's instructions or even to generate instructions that behave legally. The problem naturally becomes more acute as test programs grow longer.

  In [1], the authors describe a mechanism that deals with this problem implemented in the Genesys-Pro generator. The technique, based on the adaptive test program generation approach described here, basically

defines a 'reloading' Event. The condition of this Event is satisfied whenever there are not enough registers available. The response-template increases the number of free registers.

Exploiting Rare Situations
Some states that may be desired by the user may be difficult for the generator to cause deliberately, but can be easy to identify once they have occurred.

For example: the user may want to verify a `Branch_On_Overflow` instruction that follows some specific type of multiplication overflow that is difficult to generate analytically [2].

In this case, it may be unreasonably time-consuming to attempt to generate the specific overflow analytically. However, while the generator is used for massive random test program generation, the Event may occur by chance (perhaps with the aid of some testing knowledge in the tool). Adaptive test program generation can then be used with an Event statement in which the condition identifies that the rare overflow has occurred, and the response-template inserts the desired `Branch_On_Overflow` instruction.

Adaptive Test Program Generation in Genesys-Pro

Genesys-Pro, developed at the IBM Research Laboratory in Haifa (see Figure 2), is the main test program generator used for processor verification within IBM. It has been used for the verification of nine processor designs during the last few years. One of the major advancements made with Genesys-Pro was in a rich test program template language that enables the description of more complex scenarios. The language is hierarchical, and includes programming style constructs (e.g., variables, expressions and control constructs such as loops, if-else, etc.). The language provides the full range of expressibility between a precise description of the required scenario and a more event-driven approach, in which the user gives only the general characteristics of the scenario and allows the generator to fill in the details.

Adaptive test program generation with Event statements has been implemented as part of Genesys-Pro. Some of the adaptive test program generation cases described in the previous section (e.g., interrupts handling and escaping non-fetchable memory) were handled by the previous generation test program generator, named *Genesys*. However, there they were implemented as separate special-purpose built-in mechanisms of the generator. The introduction of Event statements gives the user direct control over the adaptive test program generation. The user can now handle his own variations of event management by simply making changes to his own template files, without needing to change the generator code.

[2] There are actually many examples of interesting constraints on operands of floating point operations for which only heuristic search solutions are currently known

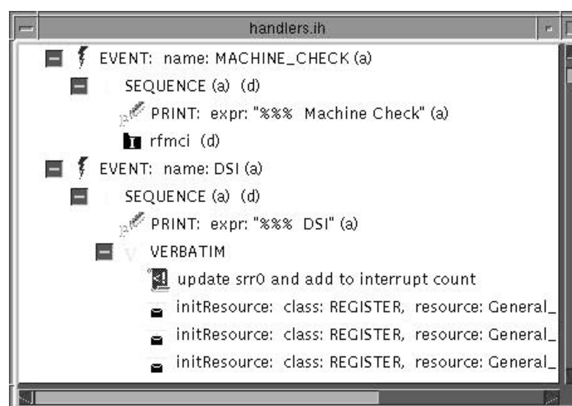

Figure 2. Genesys-Pro GUI

In addition, the usage of Events makes adaptive test program generation in Genesys-Pro more unified and simple to use and maintain. The richness of Genesys-Pro's template language, together with the flexibility of the Event statement enables the support for many new cases of adaptive test program generation (such as relocatable and asynchronous interrupts and Preemptive-Reloading [1]).

Summary

In this paper we introduced the concept of *Adaptive Test Program Generation* that uses a new language construct for test program template languages — namely an Event statement. Adaptive test program generation is required for handling situations that occur unexpectedly during generation. In this method, the generation of instructions for the main body of the test program template is interrupted when the Event's condition is satisfied, and the situation is handled by generating the Event's response-template.

In existing approaches to the problem, some unexpected situations could cause the generation to stop and some specific situations are identified and handled with special-purpose mechanisms. With adaptive test program generation, users have control over the definition of situations that require adaptation and the specific responses to these situations. All adaptive test program generation cases are handled uniformly using the same Event handling generator mechanism. This simplifies both the usage and maintenance of the tool. In addition, the specification of the way unexpected situations are handled (through Events) is made separate from the specification of the main scenario planned for the test program. In fact, Genesys-Pro maintains a separate file with default Event statements which users may include in their his own test program templates. This separation of concerns relieves the writer of the test program template from worrying about interferences of unexpected situations when designing the template. In addition, the management of unexpected situations may be altered independently without needing to change the test program templates.

Adaptive test program generation with Events has been implemented in the Genesys-Pro test generator and has been used for several years in the verification of IBM designs.

References

[1] A. Adir, E. Marcus, M. Rimon, and A. Voskoboynik. Improving test quality through resource reallocation. In *Sixth Anual IEEE International Workshop on High Level Design Validation and Test*, pages 64–69, 2001.

[2] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *the 32nd Design Automation Conference*, pages 279–285, 1995.

[3] A. M. Ahi, G. D. Burroughs, A. B. Gore, S. W. LaMar, C. R. Lin, and A. L. Wiemann. Design verification of the HP 9000 series 700 PA-RISC workstations. *Hewlett-Packard Journal*, 14(8), August 1992.

[4] Eyal Bin, Roy Emek, Gil Shurek, and Avi Ziv. Using constraint satisfaction formulations and solution techniques for random test program generation. *IBM System Journal, Special issue on AI, to be published on Aug. 2002*.

[5] A. Chandra and V. Iyengar. Constraint solving for test case generation — a technique of high level design verification. In *IEEE International Conference on Computer Design (ICCD)*, 1992.

[6] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test program generator - application to the x86 microprocessors family. In *DATE99, Munchen*, pages 434–441, 1999.

[7] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkas, K. Holtz, A. Long, D. King, and S. Barret. A methodology for the verification of a “system on a chip”. In *the 36th Design Automation Conference*, pages 574–579, 1999.

[8] A. Hosseini, D. Mavroidis, and P. Konas. Code generation and analysis for the functional verification of microprocessors. In *the 33rd Design Automation Conference*, pages 305–310, 1996.

[9] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional verification of a multiple-issue, out-of-order, superscalar alpha processor - the DEC Alpha 21264 microprocessor. In *the 35th Design Automation Conference*, pages 638–643, 1998.