Hardware/software covalidation

I.G. Harris

Abstract: Hardware/software systems are embedded in devices used to enable all manner of tasks in society today. The increasing use of hardware/software systems in cost-critical and life-critical applications has led to the heightened significance of design correctness of these systems. A summary is presented of research in hardware/software covalidation. The general covalidation problem involves the verification of design correctness using simulation-based techniques. The focus is on the test generation process, the fault models and fault coverage analysis techniques, and the test response analysis techniques employed in covalidation. The current state of research in the field is summarised and future areas for research are identified.

1 Introduction

A hardware/software system can be defined as one in which hardware and software must be designed together, and must interact to properly implement system functionality Hardware/software systems are built from a wide range of hardware and software components which are associated with different trade-offs in design characteristics, such as performance, area and reliability. Typical hardware components include application-specific integrated circuits (ASICs), application-specific instruction processors (ASIPs), general purpose microprocessors and field-programmable gate arrays (FPGAs). Software components generally vary in the abstraction of the programming language used, including compiled language components (C, C + +), interpreted language components (Java, Visual Basic) and script language components (Perl, Javascript, csh). A system design containing multiple subordinate behaviours can be designed most efficiently by mapping the subordinate behaviours to hardware and software components appropriately. For example, an internet appliance might require complex signal processing for real-time video coding, as well as a graphic user interface which can interact with other components via the internet. Such a system could be designed efficiently by using an ASIC to perform the complex signal processing in realtime, while using a Java program to implement the control logic and user interface, allowing interaction through the Internet. By using hardware and software together, it is possible to satisfy varied design constraints which could not be met using either technology separately. It is for this reason that the use of hardware/software systems is so common today, and is expected to increase in the future.

The widespread use of these systems in cost-critical and life-critical applications motivates the need for a systematic approach to verify functionality. Several obstacles to the verification of hardware/software systems make this a challenging problem, necessitating a major research effort. One issue is the high complexity of hardware/software systems which derives from both the size and the heterogeneous nature of the designs. Hardware verification complexity has increased to the point that it dominates the cost of design. In order to manage the complexity of the problem, many researchers are investigating *covalidation* techniques, in which functionality is verified by simulating (or emulating) a system description with a given test input sequence. In contrast, formal verification techniques have been explored which verify functionality by using formal techniques (i.e. model checking, equivalence checking, automatic theorem proving) to precisely evaluate properties of the design. The tractability of covalidation makes it the only practical solution for many real designs.

Figure 1 shows how covalidation fits into a generic hardware/software codesign flow. The codesign flow shown in Fig. 1 closely matches the flow proposed in [1], but covalidation is integrated in a similar manner into any codesign flow. The flow shown in Fig. 1 starts with a high-level specification and produces a partially refined design which can be completed by software compilation and behavioural hardware synthesis. Covalidation is performed after each design refinement step to guarantee that synthesis has produced a correct design. If the design is correct then the synthesis process continues, otherwise the previous synthesis step must be modified to correct any problems.

An outline of the steps involved in the covalidation process is shown in Fig. 2. Covalidation involves three major steps: test generation, cosimulation, and test response evaluation. The test generation process typically involves a loop in which the test sequence is progressively evaluated and refined until coverage goals are met. Cosimulation (or emulation) is then performed using the resulting test sequence, and the cosimulation test responses are evaluated for correctness. A key component of test generation is the covalidation fault model, which abstractly describes the expected faulty behaviours. The fault model is needed to provide fault detection goals for the automatic test generation process, and the fault model enables the fault detection qualities of a test sequence to be evaluated. Test response evaluation is also a bottleneck because it typically requires manual computation of correct responses for all test stimuli.

Several features of the hardware/software problem make it unique and difficult. Each covalidation technique addresses these issues to different degrees.

IEE Proceedings online no. 20045095

doi: 10.1049/ip-cdt:20045095

Paper first received 20th July and in revised form 12th November 2004

The author is with the Department of Computer Science, University of California Irvine, Irvine, CA 92697, USA

E-mail: iharris@uci.edu

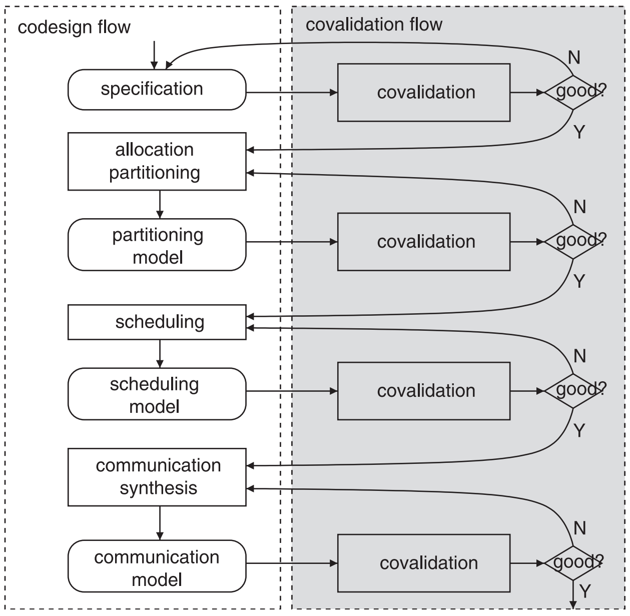


Fig. 1 *Hardware/software codesign/covalidation flow*

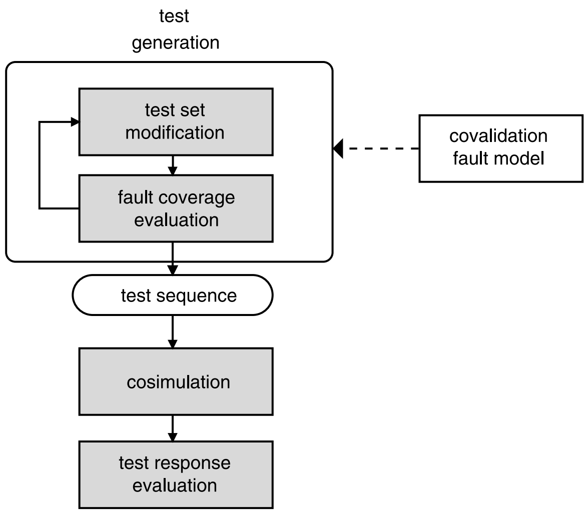


Fig. 2 *Hardware/software covalidation process*

• *Globally asynchronous locally synchronous (GALS) systems*: The complexity of hardware/software systems results in implementations which are composed of several globally asynchronous clock domains, implemented on one chip or across several chips. Sophisticated communication between domains requires an emphasis on interface design and validation.
• *Component reuse*: Component reuse is an established design paradigm in the hardware domain (i.e. core logic blocks) and in the software domain (i.e. library functions/classes, design patterns). The use of predesigned and prevalidated components reduces the need for unit testing, while increasing the relative importance of interface testing, focusing on the interactions between components.
• *Varied design styles*: Hardware/software systems are built from a wide range of different types of components, each representing different design quality trade-offs and requiring different design styles. For example, a software designer may use an object-oriented design style which abstracts detail, a hardware designer may mix top-down design with bottom-up. The nature of the design defects will depend on the design style and fault models may need to be developed to match the design style.
• *Varied design abstraction levels:* Hardware/software components are described at a variety of abstraction levels to match the need for design quality and flexibility. Although multiple abstraction levels are used, the behavioural abstraction is most often used to describe the total system specification because it is common to both hardware and software. Mixed-level simulation techniques are needed to characterise the behaviour of these complex systems. Fault models and test pattern generation algorithms must be applicable to designs represented at different abstraction levels.

In this survey we summarise research in the stages of covalidation involved with test generation, test response evaluation and the fault models which support the covalidation process.

2 Cosimulation techniques

Cosimulation is the process of simulating disparate design models together as a single system. The term 'cosimulation' has been used very broadly to encompass the simulation of not only electrical systems but also of mechanical and even biochemical systems. In this paper we will limit our definition to electrical hardware/software systems.

The challenge of cosimulation is the efficient and accurate management of the interaction between components which are described with very different computational models. Excellent simulation techniques exist for each type of component in isolation but fundamental differences in abstraction level make the simulation techniques difficult to use together. For instance, digital hardware may be simulated at a relatively low level of abstraction using an event-driven simulator with picosecond accuracy. However, software may be written at a high level of abstraction (possibly in an interpreted language like Java) and a real-time operating system (RTOS) might be used to abstract performance issues from the programmer. Software 'simulation' could be performed by running the code on any processor which supports the RTOS and the interpreter for the language.

Cosimulation tools typically model three basic types of components: software, hardware, and processor. Each hardware component can be modelled at a range of abstraction levels which can be simulated with different degrees of timing detail.

• Picosecond accurate simulation: This type of model has the highest accuracy and the lowest performance.
• Cycle-accurate simulation: This model provides accurate register contents at each clock cycle boundary.
• Transaction-level simulation: In a transaction-level model (TLM), the details of communication among computation components are separated from the details of computation components. Unnecessary details of communication and computation are hidden in a TLM and may be added later. TLMs speed up simulation and allow exploring and validating design alternatives at the higher level of abstraction.

Software simulation is accomplished by compiling the software for a target processor and simulating the processor using a model. The processor is hardware and so any of the modelling techniques described above may be used. The processor is also predesigned and usually prefabricated intellectual property (IP). In order to preserve the confidentiality of IP design, detailed information required for simulation may not be provided. The following techniques are often used to model processors in a way which does not reveal design detail.

- Instruction set simulation: The contents of memory elements are correctly modelled at machine is at instruction boundaries. Cycle-to-cycle timing effects such as pipeline stalls are ignored.
- Host processor: Rather than model the target processor, software can be compiled to a host processor and simulation is performed by executing the software as processes, which communicate with hardware simulator processes. No processor model is needed but timing accuracy suffers because the software timing is not related to the timing of the actual target processor.
- Bus functional model: A bus functional model does not model the complete behaviour of the processor, only the different bus cycles that the processor can execute. For this reason it cannot be used to simulate and debug software components. Bus functional models are used to debug hardware and its interactions with the processor by replicating the processor's bus interactions.
- Hardware modeller: This describes the use of a real processor part as the hardware model. This technique can be applied to model any prefabricated hardware including processors as well as application-specific integrated circuits (ASICs).

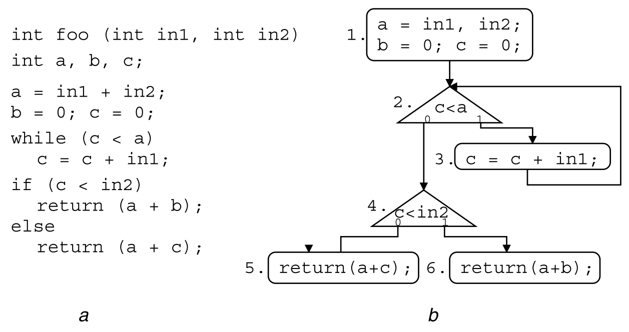
Subsets of the cosimulation problem are well studied and a number of industrial tools exist which enable the cosimulation of a variety of system types. Managing the difficult trade-off between performance and timing accuracy is still a problem for large systems. Further information on existing cosimulation techniques can be found in [2].

3 Fault models and coverage evaluation

A design error is a difference between the designer's intent and an executable specification of the design The designer's intent is most commonly expressed as a natural language specification. An executable specification is a precise description of the design which can be simulated. Executable specifications are often expressed using highlevel hardware/software languages. Design errors may range from simple syntactical errors confined to a single line of a design description, to a fundamental misunderstanding of the design specification which may impact a large segment of the description. The number of potential design errors is too large to be managed either automatically or manually, so a method is needed to reduce complexity without sacrificing accuracy. A design fault describes the behaviour of a set of design errors, allowing a large set of design errors to be modelled by a small set of design faults. A covalidation fault model describes the definition of a set of faults for an arbitrary design. A covalidation fault model allows the concise representation of the set of all design errors for an arbitrary design. Covalidation fault models can be evaluated by their accuracy in terms of modelling design errors, and their efficiency in terms of the time required to evaluate fault coverage.

The majority of hardware/software codesign systems are based on a top-down design methodology, which begins with a behavioural system description. As a result, the majority of covalidation fault models are behavioural-level fault models. Existing covalidation fault models can be classified by the style of behavioural description upon which the models are based. System behaviours are originally specified in textual languages, such as VHDL and ESTEREL, and are converted into an internal behavioural format for use in codesign and cosimulation. Many different internal behavioural formats are possible [1].

Many of the covalidation fault models currently applied to hardware/software designs have their origins in either the hardware [3] or the software [4] domains. Because the nature of errors in the hardware domain differs from that in the software domain, fault models designed for only one domain will be insufficient in isolation. The differences between errors in the hardware and software domains are understood by examining the differences between hardware description languages and software programming languages. Hardware description languages invariably include formalisms to describe **concurrency** because hardware design is often performed structurally, combining several concurrent components. In many hardware description languages concurrency is described using a *process* statement, or an equivalent statement, to define a hardware block which executes concurrently with other process blocks. Many software languages support concurrent execution, but the use of concurrency is assumed in common hardware design, while it is less well studied in software design. Hardware description languages also model event **timing** because timing constraints are essential to the hardware design process. The large majority of software does not satisfy hard timing constraints, so timing has not been thoroughly studied in the software domain. More recently, timing constraints have become important in embedded software performing real-time control tasks, but testing of real-time software is immature. Since both concurrency and timing have



```
int foo (int in1, int in2)
int a, b, c;
a = in1 + in2;
b = 0; c = 0;
while (c < a)
  c = c + in1;
if (c < in2)
  return (a + b);
else
  return (a + c);
```

Fig. 3 *Behavioural descriptions*
a Textual description
b Control-dataflow graph (CDFG)

Table 1: Taxonomy of covalidation fault models

Model class	Name of fault model
Textual	Mutation analysis
	Statement coverage
Control-dataflow	Branch coverage
	Path coverage
	Domain coverage
	OCCOM
State machine	State coverage
	Transition coverage
Gate-level	Stuck-at coverage
	Toggle (bit-flip) coverage
Application-specific	Microprocessor fault models
	User-defined
Interface	Communication faults
	Timing-induced faults

traditionally been less important in the software domain, software fault models do not provide good coverage of these errors.

As a tool to describe covalidation fault models we will use the simple system example shown in Fig. 3. Figure 3*a* shows simple behaviour, and Fig. 3*b* shows the corresponding control-dataflow graph (CDFG). The example in Fig. 3 is limited because it is composed of only a single process and it contains no signals that are used to model real time in most hardware description languages. In spite of these limitations, the example is sufficient to describe the relevant features of many covalidation fault models.

Table 1 presents a taxonomy of covalidation fault models classified according to the abstraction level of the behaviour which they operate on. Each class of fault models is described in the following Sections.

3.1 Textual fault models

A textual fault model is one which is applied directly to the original textual behavioural description. The simplest textual fault model is the statement coverage metric introduced in software testing [4], which associates a potential fault with each line of code, and requires that each statement in the description be executed during testing. This model is very efficient since the number of potential faults is equal to the number of lines of code. Fault coverage evaluation for this model is very low complexity, requiring only that an array be updated after each statement is executed. However, this coverage metric is accepted as having limited accuracy in part because fault effect observation is ignored. In spite of its limitations, statement coverage is well used in practice as a minimal testing goal.

Mutation analysis is a textual fault model which was originally developed in the field of software test [5, 6], but has also been applied to hardware validation [7]. In mutation analysis terminology, a mutant is a version of a behavioural description which differs from the original by a single potential design error. A mutation operator is a function which is applied to the original program to generate a mutant. A set of mutation operators describes all expected design errors, and therefore defines the functional error model. Since behavioural hardware descriptions share many features in common with procedural software programs, previous researchers [7] have used a subset of the software mutation operations presented in [5]. A typical mutation operation is arithmetic operator replacement (AOR), which replaces each arithmetic operator with another operator. For example in Fig. 3*a*, the line $a = in1 + in2$; would be replaced with $a = in1 - in2$, $a = in1 * in2$, and $a = in1/in2$.

Mutation fault coverage evaluation can be time consuming because it requires that each mutant be executed so that their results can be compared with the known correct results. The theoretical efficiency of this metric is good because the number of mutants in a description is $O(s * m)$, where s is the size of the behavioural description and m is the number of mutation operations applied to an individual line of code. In practice, however, the complexity of coverage evaluation can be large because multiple mutation operations may be applied to a single line of code. For example, researchers in [6] applied 22 mutation operations to an 11-line Fortran program to perform bubble sort and generated 338 mutants. Researchers [6] have investigated approaches to identify a smaller representative set of mutants in order to alleviate the time complexity problem. In addition to potential time complexity problems, the accuracy of this approach has not been demonstrated. Researchers in [5] state that their 22 mutation operations capture the errors that Fortran programmers typically make, but no empirical evidence is provided. Also, the local nature of the mutation operations may limit its ability to describe a large set of design errors.

3.2 Control-dataflow fault models

A number of covalidation fault models are based on the traversal of paths through the CDFG representing the system behaviour. In order to apply these fault models to a hardware/software design, both hardware and software components must be converted into a CDFG description. Applying these fault models to the CDFG representing a single process is a well understood task. The application of CDFG fault models to the behaviour of an entire hardware/software system would require that all component CDFGs be merged into one. A weakness of CDFG fault models is that the process of merging multiple concurrent CDFGs has not been well studied. For this reason, CDFG fault models are currently restricted to the testing of single processes. The earliest control-dataflow fault models include branch coverage and path coverage [4] models used in software testing.

The branch coverage metric associates potential faults with each direction of each conditional in the CDFG. Branch coverage requires that the set of all CDFG paths covered during covalidation include both directions of all binary-valued conditionals. Branch coverage is commonly used for hardware validation and software testing, but it is also accepted to be insufficient to guarantee correctness alone. The efficiency of the branch coverage metric is high because it can be computed by analysing a single cosimulation output trace. Branch coverage evaluation is performed by recording the direction of each branch as it is taken during simulation. The branch coverage metric has been used for behavioural validation by several researchers for coverage evaluation and test generation [8–10]. The accuracy of branch coverage has been studied to determine its ability to cover design errors [8, 9]. In [8] researchers found that branch coverage, together with toggle coverage, was sufficient to ensure the detection of 25 of 26 total design errors in a five-stage pipelined microprocessor example.

The path coverage metric is a more demanding metric than the branch coverage metric because path coverage reflects the number of control-flow paths taken. The assumption is that an error is associated with some path through the control flow graph and, therefore, all control paths must be executed to guarantee fault detection. The number of control paths can be infinite when the CDFG contains a loop as in Fig. 3*b*, so the path coverage metric may be used with a limit on path length [11]. Since the total number of control-flowpaths grows exponentially with the number of conditional statements, several researchers have attempted to select a subset of all control-flow paths which are sufficient for testing. One path selection criterion is presented in [12] (based on work in software test [13]) and identifies a basis set of paths, a subset of paths which are linearly independent and can be composed to form any other path. Previous work in software test [14–18] has investigated dataflow testing criteria for path selection. In dataflow testing, each variable occurrence is classified as either a definition occurrence or a use occurrence. Paths are selected which connect a definition occurrence to a use occurrence of the same variable. For example in Fig. 3*b*, node 1 contains a definition of signal a and nodes 2, 5 and 6 contain uses of signal a. In this example, paths 1, 2, 4, 5 and 1, 2, 4, 6 must be executed in order to cover both of these definition–use pairs. The dataflow testing criteria have also been applied to behavioural hardware descriptions [19].

The complexity of fault coverage evaluation for path coverage metrics is largely determined by the number of control-flow paths considered by the metric. If all paths are considered then the computation of coverage requires that all control-flow paths be enumerated. Since the number of paths is exponential in the number of control branches, the time and space required to enumerate all paths can be large. In practice however, the number of paths in a single process is relatively low because designers commonly limit complexity to improve understandability and maintainability.

The majority of control-dataflow fault models consider the control-flow paths traversed without overconstraining the values of variables and signals. For example in Fig. 3*b*, in order to traverse path 1, 2, 3, the value of c must be minimally constrained to be less than a, but no additional constraints are required. This can be contrasted with variable/signal-oriented fault models which place more stringent constraints on signal values to ensure fault detection. The domain analysis technique in software test [4, 20] considers not only the control-flow path traversed, but also the variable and signal values during execution. A domain is a subset of the input space of a program in which every element causes the program to follow a common control path. A domain fault causes program execution to switch to an incorrect domain. Domain faults may be stimulated by test points anywhere in the input space, but they are most likely to be stimulated by inputs which cause the program to be in a state which is 'near' a domain boundary. An example of this property can be seen in Fig. 3*b* in the traversal of path 1, 2, 3. The only constraint required is that $c < a$, but if the difference between c and a is small, then there is a greater likelihood that a small change in the value of c will cause the incorrect path to be traversed. Researchers have applied this idea to develop a domain coverage fault model which can be applied to hardware and software descriptions [21].

Many control-dataflow fault models consider the requirements for fault activation without explicitly considering fault effect observability. Researchers have developed observability-based behavioural fault models [22–25] to alleviate this weakness. The OCCOM fault model has been applied for hardware validation [22, 23] and for software validation [24]. The OCCOM approach inserts faults called *tags* at each variable assignment which represent a positive or negative offset from the correct signal value. The sign of the error is known but the magnitude is not. Observability analysis along a control-flowpath is done probabilistically by using the algebraic properties of the operations along the path and simulation data. As an example, in Fig. 3 we will assume that a positive tag is inserted on the value of variable c and we must determine if the tag is propagated through the condition $c < in2$ in node 4 of Fig. 3*b*. Since the tag is positive, it is possible that the conditional statement will execute incorrectly in the presence of the tag, so the OCCOM approach optimistically assumes tag propagation in this case. Notice that a negative tag could not affect the execution of the conditional statement. While the approach presented in [22–24] determines observability in a probabilistic fashion, other researchers have developed a precise technique [25]. Work in [25] injects stuck-at faults on internal variables and determines fault effect propagation behaviourally. Because the observability analysis is precise, the computational complexity is increased.

3.3 State machine fault models

Finite state machines (FSMs) are the classic method of describing the behaviour of a sequential system and fault models have been defined to be applied to state machines. The commonly used fault models [26–28] are the state coverage model which requires that all states be reached, and transition coverage which requires that all transitions be traversed. Calculation of coverage using these models requires the update of a table containing all states and transitions in the behaviour. The act of updating these tables is not time consuming, but the size of these tables will be large for realistic state machines. These fault models have also been refined to differentiate faults in the output function from faults in the next state function [29]. State machine transition tours, paths covering each transition of the machine, are applied to microprocessor validation [30]. A user-refined transition coverage model has been proposed [31], which selects only transitions which affect state variables which are identified by the user as being important for test. The problems associated with state machine testing are understood from classical switching theory [32] and are summarised in a thorough survey of state machine testing [33].

The most significant problem with the use of state machine fault models is the complexity resulting from the state space size of typical systems. Several efforts have been made to alleviate this problem by identifying a subset of the state machine which is critical for validation. The extended finite state machine (EFSM) [34] and the extracted control flow machine (ECFM) [27] models create a reduced state machine by partitioning the state bits between control and data bits. These techniques can be quite effective for systems with a large datapath which does not significantly impact control flow. For example, the ECFM technique [27] reduces a 32-bit microprocessor design with 251 flip-flops to a state machine with only 17 reachable states. In [35] a reduced state machine is generated by projecting the original state machine onto a set of states which are identified as being interesting for validation purposes. These state machine reduction techniques have enabled validation to be performed for several large-scale designs.

3.4 Gate-level fault models

A gate-level fault model is one which was originally developed for and applied to gate-level circuits. Manufacturing testing research has defined several gate-level fault models which are now applied at the behavioural level [36, 37]. For example, the stuck-at fault model assumes that each signal may be held to a constant value of 0 or 1 due to an error. The stuck-at fault model has also been applied at the behavioural level for manufacturing test [38] and for hardware/software covalidation [39, 40]. Behavioural designs often use variables which are represented with many bits and gate-level fault models are typically applied to each bit, individually. For example, if we assume that an integer as declared in Fig. 3*a* is 32 bits long, then applying the single stuck-at fault model to a variable would produce 32 stuck-at-1 faults and 32 stuck-at-0 faults. The toggle coverage fault model, which requires that each bit signal transition up and down, has been applied for design validation and has been expanded to consider observability [25].

Gate-level fault models have the potential weakness that they are structural in nature rather than behavioural. The relationship between behaviour and structure has been studied in previous work [41], but the effectiveness of applying structural fault models to a behavioural description is unproven. The time complexity of coverage computation with gate-level models has been well studied in the area of manufacturing test. In general a coverage computation with a gate-level fault model should require more time than that required when a more abstract model is used, but many efficient fault simulation approaches have been developed for gate-level analysis [36, 37].

3.5 Application-specific fault models

A fault model which is designed to be generally applicable to arbitrary design types may not be as effective as a fault model which targets the behavioural features of a specific application. To justify the cost of developing and evaluating an application-specific fault model, the market for the application must be very large and the fault modes of the application must be well understood. For this reason, application-specific fault models are seen in microprocessor test and validation [42–48]. Early microprocessor fault models target relatively generic microprocessor features. For example, researchers define a fault model for instruction-sequencing functions [44] by describing the fault effects (i.e. activation of erroneous microorders), and describing the fault detection requirements. More recent fault models target the modern processor features such as pipelining [46–48].

Another alternative to the use of a traditional fault model is to allow the designer to define the fault model. This option relies on the designer's expertise at expressing the characteristics of the fault model in order to be effective. Several tools have been developed which automatically evaluate user-specified properties during simulation to identify the existence of faults. These approaches differ in the method by which the designer specifies the fault model. The simplest techniques used in common hardware/software debuggers allow the user to specify breakpoints based on the values of a subset of state variables. More sophisticated tools allow the designer to use temporal logic primitives to express faulty conditions [49, 50].

3.6 Interface faults

To manage the high complexity of hardware/software design and covalidation, efforts have been made to separate the behaviour of each component from the communication architecture [51]. Interface covalidation becomes more significant with the onset of core-based design methodologies which utilise predesigned, preverified cores. Since each core component is preverified, the system covalidation problem focuses on the interface between the components.

A case study of the interface-based covalidation of an image compression system has been presented [52]. Researchers classify the interface fault which occurred during the design process into three groups: (i) COMP2-COMP faults involving communication between pairs of components, (ii) COMP2COMM faults involving the interaction between each component and the communication architecture, and (iii) COMM faults involving the coordinated interactions between the communication architecture and all components. In [52], test benches are developed manually to target each of these interface fault classes.

Additional interface complexity is introduced by the use of multiple clock domains in large systems. The interfaces between different clock domains must be essentially asynchronous. Unless a high-overhead timing-independent circuit implementation is used (such as differential cascode voltage switch logic), asynchronous interfaces are particularly vulnerable to timing-induced faults. Timing-induced faults are described in [53] as faults which cause the definition of a signal value to occur earlier or later than expected. An example of the occurrence of this type of fault would be an increased delay on the *empty* status signal of a FIFO. If the *empty* signal is issued too late, the FIFO may be read from while it is empty. In [21] a timing fault model is presented and a technique for fault coverage evaluation is introduced.

4 Automatic test generation techniques

Several automatic test generation (ATG) approaches have been developed which vary in the class of search algorithm used, the fault model assumed, the search space technique used and the design abstraction level used. In order to perform test generation for the entire system, both hardware and software component behaviours must be described in a uniform manner. Although many behavioural formats are possible [1], previous ATG approaches have focused on CDFG and FSM behavioural models.

Table 2 presents a taxonomy of covalidation test generation techniques classified according to the coverage goal of the search algorithm. Each class of test generation techniques is described in the following Sections.

Two classes of search algorithms have been explored, fault-directed and coverage-directed. Figure 4 shows an outline of both of these classes of algorithms. Fault-directed techniques successively target a specific fault and construct a test sequence to detect that fault. Each new test sequence is merged with the current test sequence (typically through concatenation) and the resulting fault coverage is evaluated to determine if test generation is complete. This class of algorithms suffers in terms of time complexity because it directly solves the test generation problem for individual faults, requiring a complex search of the space of input sequences. However, fault-directed algorithms have the advantage that they are complete in the sense that a test sequence will be found for a fault if a test sequence exists. Another class of search algorithms are the coverage-directed algorithms which seek to improve coverage without targeting any specific fault. These algorithms heuristically modify an existing test set to improve total coverage, and then evaluate the fault coverage produced by the modified test set. If the modified test set corresponds to an improvement in fault coverage then the modification is accepted. Otherwise the modification is either rejected or another heuristic is used to determine the acceptability of the modification. Coverage directed techniques have the potential to be much less time consuming than fault-directed techniques because may use fast heuristics to modify the test set. The drawback of coverage-directed techniques is that they are not guaranteed to detect any particular fault although the fault may be detectable.

Table 2: Taxonomy of covalidation test generation techniques

Test generation class	Solving technique
Fault-directed	Linear programming + SAT
	Integer linear programming + SAT
	Constraint logic programming
	Model checking counterexample
	Switching theory
	Implicit state enumeration
Coverage-directed	Genetic algorithms
	Random mutation hill climbing
	Directed random tests

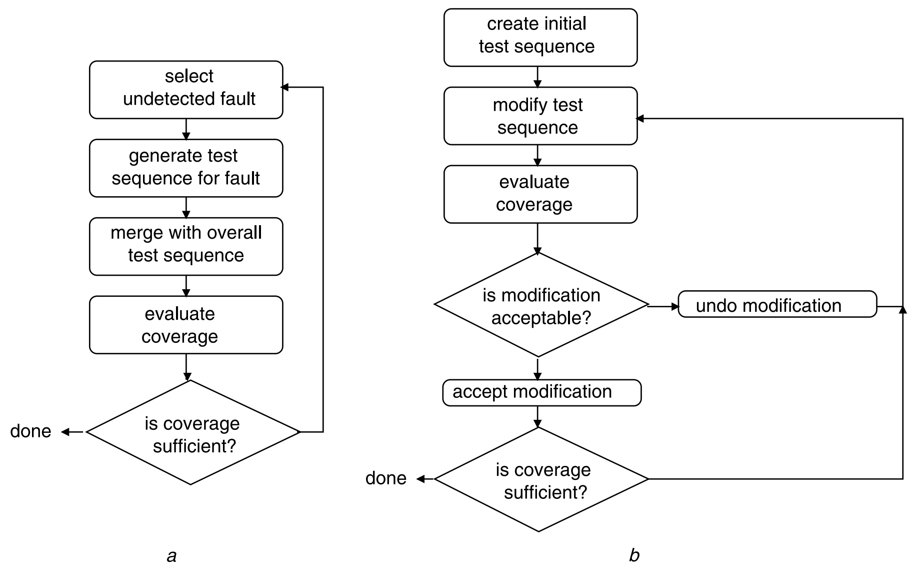


Fig. 4 *Classes of test generation algorithms*
a Fault directed
b Coverage directed

4.1 Fault-directed techniques

Several researchers have chosen to address the test generation problem directly at the CDFG level by identifying a set of mathematical constraints on the system inputs which cause a chosen CDFG path to be traversed. Once the constraints have been identified, the test generation problem is equivalent to the problem of solving the constraints simultaneously to produce a test sequence at the system inputs. Each CDFG path can be associated with a set of constraints which must be satisfied to traverse the path. For example, in Fig. 3*b* the path containing nodes 1, 2, 4 and 6 is associated with the requirement that $c \geq a$ and $c < in2$. Because the operations found in a hardware/software description can be either boolean or arithmetic, the solution method chosen must be able to handle both types of operations. The boolean version of the problem is traditionally referred to as the SATISFIABILITY (SAT) problem [54] and has been well studied as the fundamental NP-complete problem. Handling both boolean and arithmetic operations poses an efficiency problem because classical solutions to the two problems have been presented separately. For instance, BDD-based techniques perform well for boolean operations but the complexity of modelling word-level operations with BDDs is high.

In [55, 56] researchers define the HSAT problem as a hybrid version of the SAT problem which considers linear arithmetic constraints together with boolean SAT constraints. Researchers in [55] present an algorithm to solve the HSAT problem which combines a SAT-solving technique [57] with a traditional linear program solver. The algorithm progressively selects variables and explores value assignments while maintaining consistency between the boolean and the arithmetic domains. The approach presented in [55] is shown to operate up to three orders of magnitude faster than a strictly boolean SAT tool, and solved a 104 constraint problem which could not be solved using another tool. Other researchers have solved the problem by expressing all constraints in a single domain and using a solver for that domain. In [58] researchers formulate boolean SAT constraints as integer linear arithmetic constraints. This allows the entire set of constraints to be solved using an integer linear program (ILP) solver. Using the CPLEX ILP solver, the approach described in [58] was used to solve a 3673-constraint problem in under 9 s on a 500 MHz Pentium III.

Constraint logic programming (CLP) techniques [59] have been employed which can handle a broad range of constraints including nonlinear constraints on both boolean and arithmetic variables. CLP techniques are novel in their use of rapid incremental consistency checking to avoid exploring invalid parts of the solution space. Different CLP solvers use a variety of constraint description formats which allow complex constraints to be captured. Researchers in [60] use the GNUProlog engine [61] to generate tests by converting boolean and arithmetic constraints into Prolog predicates. CLP has also been used to generate tests for path coverage in a control-dataflow graph in [11] where the arithmetic constraints expressed at each branch point of a path are solved together to generate a test which traverses the path. Using the CLP(R) solving engine [62], researchers generated path tests for several large examples, including a microprocessor description of 400 lines of VHDL code. In [12] researchers employ an approach similar to the one used in [11] to explore a subset of paths which are linearly independent. In [63] the CLP approach is used to generate tests related to the synchronisation between concurrent hardware/software processes. Constraints are generated which describe the behaviour of the hardware/software system and which describe the conditions which would activate a potential synchronisation fault. In general, the performance of using CLP has been shown to be several orders of magnitude better than the use of strictly boolean SAT solvers. For example, results in [60] show up to three orders of magnitude improvement in solution time compared to some of the best boolean SAT solvers.

Although binary decision diagrams (BDDs) represent boolean functions by their nature, BDDs have been used at the behavioural level to describe the CDFG of a behavioural VHDL description [40, 64, 65]. These approaches describe arithmetic functions in the boolean domain by describing each output bit function as a BDD. Stuck-at faults are inserted at each variable bit to generate faulty BDDs. Test patterns are identified by solving the SAT for the machine which is the exclusive OR of the good and faulty machines. In most cases, BDDs reduce storage requirements by enabling compact representation of logic functions. This trend is borne out in the results presented in [65], but it is violated in a few cases. For example, the b11 benchmark is described with 119 lines of VHDL code and requires 15.2 Mbyte of storage, while the shewa example is only 102 lines of VHDL code but requires 554 Mbyte of storage.

Test generation at the state machine level involves the identification of paths through the state machine which satisfy the selected fault coverage goal. Once a path has been identified, the test sequence is the sequence of input events which trigger each edge on the path. Generating a test sequence from a path becomes complicated if the state machine inputs are not primary inputs of the system. This situation exists in microprocessor test where the inputs to the controller are instruction and data streams taken from memory. The commonly used state machine fault models are state coverage and branch coverage, so the goal of test generation is to identify a path which includes all states or transitions.

State machine testing has been accomplished by defining a transition tour which is a path which traverses each state machine transition at least once [30]. Transition tours have been generated by iteratively improving an existing partial tour by concatenating on to it the shortest path to an uncovered transition [30]. In [31], a test sequence is generated for each transition by asserting that a given transition does not exist in a state machine model, and then using the model checking tool SMV [66] to disprove the assertion. A byproduct of disproving the assertion is a counterexample, which is a test sequence that includes the transition. Since this technique relies on model checking technology, it shares its performance and memory requirement characteristics with model checking approaches.

If a fault effect can be observed directly at the machine outputs, then covering each state and transition during test is sufficient to observe the fault. In general, a fault effect may cause the machine to be in an incorrect state, which cannot be immediately observed at the outputs. In this case, a *distinguishing sequence* must be applied to differentiate each state from all other states based on output values. The testing problems associated with state machines, including the identification of distinguishing, synchronising and homing sequences, are well understood [32, 33].

A significant limitation to state machine test generation techniques is the time complexity of the state enumeration process performed during test generation. The abstraction method used to represent the state machine has been shown to greatly impact the complexity of the state enumeration process. BDDs have been used to represent the state transition relation and efficiently perform implicit state enumeration by defining an *image* computation which computes the states which are reachable from a given set of states [67]. The efficiency of this method of state enumeration has led to its use during the state machine test generation process [27, 35]. In [27], the state coverage metric is used during test generation, but the direct metric is inaccurate if there is a large set of unreachable states in a machine. To increase the accuracy of the state coverage metric, the number of states covered is divided by the number of reachable states in the machine; implicit state enumeration is used to identify the set of reachable states. In [35], coverage analysis and test generation are made more efficient by evaluating only a portion of the state space. Implicit state enumeration is performed on machine R to generate a new machine R_I which only contains paths in R which lead to states which are 'interesting' for testing. The set of interesting states is used during implicit state enumeration to generate the reduced machine R_I.

4.2 Coverage-directed techniques

Several techniques have been developed which generate test sequences without targeting any specific fault. Coverage is improved by modifying an existing test sequence, and then evaluating the coverage of the new sequence. These techniques differ in the method used to modify the test sequence the cost function used to evaluate a sequence and the criteria used to accept a new sequence. The modification method is typically either random or directed random.

An example of such a technique is presented in [10, 68] which uses a genetic algorithm to successively improve the population of test sequences. In the terminology of genetic algorithms, a 'chromosome' describes a test sequence. Many test sequences are initially generated randomly. Random matings can occur between the chromosomes which describe the test sequences, but the mating process defines and restricts the way in which two test sequences are merged. The cost function (or *fitness* function) used to evaluate a test sequence is the total number of elementary operations (variable read/write) which are executed. In this technique, the total number of elementary operations is being used as an approximation of the likelihood of error detection. This algorithm has been applied to an industrial VHDL description containing over 10 000 lines of code. Using 150 CPU hours, the genetic algorithm generated a test sequence with 5 219 patterns producing 77.13% branch coverage, while a manually created test set containing 400 vectors achieved only 63.17%.

Work presented in [69] uses a random mutation hill climber (RMHC) algorithm which randomly modifies a test sequence to improve a testability cost function. The test sequence modification is completely random and the criteria for accepting a new sequence is that the cost function is improved. The fault model targeted using this approach is the single stuck-at fault model applied to the individual bits of each variable in the behavioural description. The cost function used contains two parts: (i) the number of statements executed by the sequence, and (ii) the number of outputs which contain a fault effect. Results show that the CPU time required using this approach is nearly an order of magnitude less than the time required using commercial gate-level test generation tools.

In [70] researchers generate directed-random pattern sequences to be used for test. No particular fault model is assumed in this approach, so it is up to the user to provide the directives for pattern generation. Two types of directives are used: (i) constraints which define the boundaries of the space of feasible test patterns, and (ii) biases which direct assignments of values to signals in a nonrandom way. For example, a constraint might indicate that the following relationship between variables must hold, $in1 < in2$. Because this is a constraint, no test can be generated which violates this condition. A bias expresses the desired probability distribution for the values of a signal throughout the set of all patterns. For example, a bias of the form $(in2, 0.9)$ would indicate that the probability that input $in2$ is equal to 1 should be 0.9. It is the task of the test engineer to develop a set of constraints and biases which will reveal a particular class of faults. As an example, this technique was used to generate tests for a bus interface unit. The stated test goal was to execute as many read/write requests as possible. The automatically generated test patterns generated 130 times as many read/write requests compared to a fully random sequence of the same length.

5 Test response analysis

Detection of errors requires that the test responses gathered during cosimulation be compared to the correct test responses Since any signal value can be observed at any point during simulation, the potential amount of test data is enormous. Manual evaluation of test responses requires that correct signal values be predetermined. Manually computing correct test results is time consuming and efforts have been made to automate the process. The use of assertions and self-checkers to evaluate test responses have been investigated in both the hardware and software domains.

5.1 Assertions

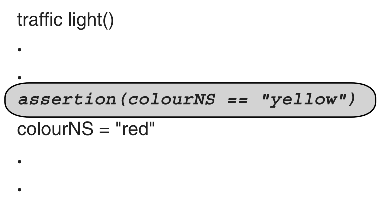
An assertion is a logical statement which expresses a fact about the system state at some point during the system's execution. Assertions have been proposed and used in both software [71, 72] and hardware domains [73]. The use of assertions is well accepted and has been integrated into the design proccess of many large-scale systems [74–76]. Assertions are primarily useful in evaluating the correctness of the system state during simulation, but the use of assertions has also extended to supporting functional test generation [70] and performance analysis [77].

An assertion is typically written as a logical relationship between the values of different storage elements which would be variables in a software program, but would also include registers, flip-flops, latches and time-varying signals in a hardware description. Many languages for the description of assertions have been proposed, but we will use a simple first-order predicate calculus in most of our examples. In this discussion we are interested in the concepts behind the use of assertions, rather than any specific implementation details. For this reason our examples will use a generic syntax, which may not match any specific assertion language, but is sufficient to describe the concepts related to assertions. In a system which describes the operation of traffic lights at an intersection, we might want to express the fact that both lights cannot be green at the same time as follows:

$$\overline{(colourNS == ``green") \cap (colourEW == ``green")} \tag{1}$$

In (1) variables *colourNS* and *colourEW* represent the colours of the north–south and east–west signal lights, respectively. The assertion in (1) can be referred to as a positive assertion because it expresses a relationship which must be satisfied. Notice that when (1) is negated, then it is a negative, assertion which expresses a relationship which must not be satisfied. Negative assertions are essentially the same as error handling code, commonly used in software, which throw an exception when the system enters an incorrect state. Notice that positive and negative assertions are equivalent in their information content so we will assume the use of positive assertions through the remainder of this Section, without loss of generality.

Assertions can be evaluated during simulation to determine whether or not an error occurred which forces the system into a state which is known to be incorrect. Assertions may be defined globally which must be satisfied at all times, or assertions may be defined locally which are only satisfied at the point in the description where the assertion is placed. The traffic light controller assertion in (1) is an example of a global assertion. A local assertion has the potential to specify more fine-grained properties because the assertion can be defined to use state information derived from its position in the description. Figure 5 shows an example of a local assertion in a traffic light controller code segment. The assertion in Fig. 5 states that the colour of the north–south light must be yellow. This statement is clearly not always true in general, but the assertion is added just before the statement which changes the light colour to red. If we assume that the light must be yellow before it is red, then the assertion must be true at the point in the description where it is asserted.



```
traffic light()
.
.
assertion(colourNS == "yellow")
colourNS = "red"
.
.
```

Fig. 5 *Local assertion in a traffic light controller*

An assertion which describes the state of a system at a point in its execution can be referred to as an 'instantaneous assertion'. The assertions in (1) and Fig. 5 are both instantaneous because they both describe properties at one time step. Although the assertion in (1) is globally true at all time steps, it is considered to be instantaneous because it expresses a statement about each time step individually, independent of all other time steps. An assertion is referred to as being *temporal* if it expresses constraints on sequences of time steps. In order to express temporal constraints, a logic must be used which expresses temporal relationships. To give an example of a temporal assertion we will introduce the *next* operator used in PSL [78]. The statement $p \rightarrow next\ q$ states that if statement p is true at time step t then statement q must be true at time step $t+1$. Using this temporal operator we can state the fact that the north–south traffic light must turn red one time step after it becomes yellow with the following expression:

$$(colourNS == ``yellow") \rightarrow next\ (colourNS == ``red") \tag{2}$$

An assertion defines boundaries on the correct execution of the system. An instantaneous assertion defines a subset of the state space and a temporal assertion defines a subset of the set of all execution sequences. The discussion here will be limited to instantaneous assertions, but the same argument could be extended to temporal assertions as well. The state space subset defined by an instantaneous assertion must contain the actual system state at the point in execution where the assertion is evaluated. Figure 6 is used to show the state space hierarchy during system execution. The largest space in Fig. 6, called the cross-product state space is the space defined by the cross-product of the states of all individual state elements in the system. Only a subset of these states, referred to as 'all feasible states', may be entered during the operation of the system if it is free of design errors. At any given point during the operation of the system there is a subset of the all feasible states set, called 'current states', which must contain the current system state if the system is

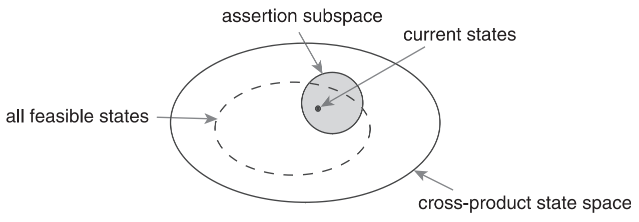


Fig. 6 *State space hierarchy*

error free. If the system is completely deterministic then the set of current states must have cardinality 1. The current states set must be a subset of the all feasible states set because the current states set is dependent on the input sequence and the point in simulation being evaluated, while the all feasible states set is the union of feasible states over all possible test sequences and points in simulation.

An instantaneous assertion defines a subspace referred to as the 'assertion subspace', which must completely contain the current states as shown in Fig. 6. Test response evaluation is performed by checking the satisfaction of each assertion; if an assertion is not satisfied then a design error exists.

5.1.1 Assertion completeness:

The main difficulty with the use of assertions is that the satisfaction of all assertions does not guarantee that errors did not occur during simulation. This is because the assertion subspace is a superset of the set of current states. In the traffic light controller, for example, both north–south and east–west traffic lights may become yellow at the same time due to a design error without violating the assertion in (1). To increase the chances that errors are detected, the set of assertions must be as complete as possible. In terms of the state space, this means that the assertion subspace must be as small as possible while still containing the correct states set. This requires that the assertions be written as strictly as possible to reduce the number of incorrect states which can satisfy the assertion. For example, the assertion for the traffic light controller in (1) can be replaced by the stronger assertion in (3).

$$(colourNS == ``red'') \cup (colourEW == ``red'') \qquad (3)$$

The assertion in (3) is stronger than (1) because the subspace that (3) defines is a proper subset of the subspace defined by (1). Notice that the assertion in (3) catches the erroneous condition when both light directions are yellow at the same time.

Defining a set of assertions which is complete is difficult because the task of assertion definition is largely manual, so the completeness of a set of assertions depends on the abilities of individual designers. Definition of a complete set of assertions is typically very expensive due to the rigorous and manual nature of the process. The cost investment is worthwhile for some highly standardised and reused applications, such as floating-point division [79] and the PCI bus protocol [80]. In [81] researchers have evaluated the completeness of a set of properties/assertions by injecting design errors and determining whether or not the assertion set can detect each error. If an error is inserted which cannot be detected, then the assertion set is assumed to be incomplete and it is the responsibility of the designer to add assertions to detect the error. The nature of the errors inserted are 'bit-flip' errors where a single bit in the description may be flipped to an incorrect value.

The approach presented in [81] can identify weaknesses in the set of assertions, but it is still the designer's task to write the assertions. It has been observed in previous work [82] that a formal specification can itself be used to generate assertions at the inputs and outputs of a system. The primary use of a specification is to express the system outputs as a function of the system inputs. If the relationship between inputs and outputs is expressed in a formal logic language then the specification is a set of assertions on the system outputs. In [82] researchers present a set of style rules, which should be followed in the definition of the formal specification to make it more effective as a set of assertions.

5.2 Self-checking

A self-checking component is one which automatically evaluates its correctness by comparing its results to the results of one or more other redundant components which implement the same function. Using only two redundant components enables the detection of errors, but not correction, since it is impossible to know which redundant version is the one with the correct result. At least three or more components allow correction as well. If it is assumed that the likelihood of a majority of components producing incorrect answers is very small, then correction can be achieved by selecting the result produced by the majority of redundant components.

Self-checkers are distinguished from assertions in a number of ways including their description style. While assertions are described declaratively, as logical statements, self-checkers are described procedurally as a sequence of operations. Also, a self-checker does not simply restrict the space of correct results as an assertion would. A self-checker actually computes the correct result(s). In terms of the state space hierarchy shown in Fig. 6, a self-checker computes the set of correct states, just as an ordinary component would. Defining features of a self-checking technique includes the implementation of the redundant components and the number of redundant components used.

An important distinction between self-checking techniques is the point in the system's lifecycle when they are applied. Self-checking can be applied prior to deployment of the system in the field for the purpose of validation. Self-checking can also be applied post-deployment to enhance reliability. Self-checking incurs some overheads in terms of cost, performance and power, which can be difficult to justify in tightly constrained systems. High overhead is one reason that self-checking is not well used in standard hardware and software projects today.

A key requirement of any self-checking technique is that the redundant components used for comparison must not operate in exactly the same way as the original component so that they do not all manifest the same errors. One way to accomplish this is by assigning completely different design teams to implement the same system. This approach is referred to as *N*-version programming in the software domain [83–85]. A significant limitation of this approach is the exorbitant cost of multiple design teams. The reliabilty provided using this approach relies on the independence of the design teams. Such independence is difficult to establish in practice because programmers are likely to be trained in the same industrial environment and using the design tools. Designer independence also contradicts the current trend toward design reuse to reduce design times.

A theoretical framework for self-checking has been developed by Blum and Kanna [86] and has been applied to several practical programming examples [87]. In [87] a general technique is presented to create a self-checking program from a non-self-checking program for numerical programs including matrix multiplication and integer division. The self-checking technique exploits a property of many numerical functions referred to as 'random self-reducibility'. A function f is random self-reducible if $f(x)$ can always be computed as $F(f(a_1), \ldots, f(a_c))$, where F is an easily computable function and the numbers $a_1, \ldots, a_c$ are randomly deistributed and are also easily computable given x. The key idea is that $f(x)$ can be computed as a function of $f(a_1), \ldots, f(a_c)$. If the numbers $a_1, \ldots, a_c$ are randomly distributed, then it is very unlikely that the implementation of f would produce an incorrect result for the majority of values $a_1, \ldots, a_c$.

The advantage of the technique presented in [87] is that the self-checking function is created in a straightforward way from the original program without the need for alternate design teams. This greatly reduces design cost compared to *N*-version programming, and it simplifies reliability analysis since the level of independence between different design teams is not an issue.

5.2.1 Self-checking for physical defects:

Current trends in soft error rates in hardware are motivating the use of self-checking to detect and correct physical defects as opposed to designer errors. As device sizes shrink to nanometer scales, the occurrence and impact of physical defects are increasing to the point that architectural strategies are required to mitigate the cost of defects. Even small parametric variations are having a significant impact on functional operation of extremely small devices in current integrated circuits ICs [88]. The number of defects produced both by electronic wear mechanisms and by terrestrial radiation sources are on the rise as device sizes scale down. Currently the increasing defect levels are primarily impacting only the most dense components, such as DRAMs, but the impact on general logic is growing and is projected to be substantial by 2011 [89].

This paper investigates the detection of design errors rather than physical defects. For this reason the problem of self-checking for physical defects in hardware is considered to be out of the scope of this paper.

6 Conclusions and future directions

It is clear that the field is maturing as researchers have begun to identify and agree on the essential problems to be solved. Our understanding of covalidation has developed to the point that industrial tools are available which provide practical solutions to test generation, particularly at the state machine level. Although automation tools are available, they are not fully trusted by designers and as a result, a significant amount of manual test generation is required for the vast majority of design projects. By examining the state of previous work, we can identify areas which should be studied in future work in order to increase the industrial acceptance of covalidation techniques.

A significant obstacle to the widespread acceptance of available techniques is the lack of faith in the correlation between covalidation fault models and real design errors. Automatic test generation techniques have been presented which are applicable to large scale designs, but, until the underlying fault models are accepted, the techniques will not be applied in practice. Fault models must be evaluated by identifying a correlation between fault coverage and detection of real design errors. Essential to this evaluation is the compilation of design errors produced by real designers. Research has begun in this direction [90, 91] and should be used to evaluate existing covalidation fault models. Once covalidation fault models are empirically evaluated we can expect to see large increases in covalidation productivity through the automation of test generation.

Analysis of test responses is a bottleneck in the covalidation process because the definition of assertions and self-checkers requires design understanding that only a designer can have. Assertions, for example, express properties of the design which must be satisfied, but developing these properties requires an understanding of the specification. It is possible to generate assertions which are generically applicable to a class of designs such as microprocessors (i.e. 'all RAW hazards are illegal in any pipeline') but properties unique to a design must be expressed manually.

A great deal of research in hardware/software covalidation is extended from previous research in the hardware and software domains, but communication between hardware and software components is a problem unique to hardware/software covalidation. The interfaces between hardware and software introduce many new design issues which can result in errors. For example, software may be executed on an embedded processor which is in a different clock domain to other hardware blocks which it communicates with. Such communication requires the use of some asynchronous communication protocol, which must be implemented in hardware and in software. Asynchronous communication is a difficult concept for both hardware and software designers, so it can be expected to result in numerous design errors. Hardware/software communication complexity is also increased because interprocessor communication is handled very differently in hardware compared to software. Hardware description languages typically provide only the most basic synchronisation mechanisms, such as the *wait* expression in VHDL. More complicated protocols (i.e. two-way handshake) must be implemented manually and are therefore vulnerable to design errors. Interprocess communication in software tends to use high-level communication primitives, such as monitors (i.e. the *synchronised* statement in Java). Although the implementation of each primitive may be known to be correct, the primitive itself may be used incorrectly by the designer, resulting in design errors. Relatively little research has investigated testing the interfaces between hardware and software components, but this research area is essential.

7 References

1 Gajski, D.D., and Vahid, F.: 'Specification and design of embedded hardware-software systems', *IEEE Des. Test Comput.*, 1995, **12**, (1), pp. 53–67
2 Rowson, J.A.: 'Hardware/software co-simulation'. Design Automation Conf., June 1994, pp. 439–440
3 Tasiran, S., and Keutzer, K.: 'Coverage metrics for functional validation of hardware designs', *IEEE Des. Test Comput.*, 2001, **18**, (4), pp. 36–45
4 Beizer, B.: 'Software testing techniques' (Van Nostrand Reinhold, 1990, 2nd edn.)
5 King, K.N., and Offutt, A.J.: 'A fortran language system for mutation-based software testing', *Softw. Pract. Exp.*, 1991, **21**, (7), pp. 685–718
6 Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., and Zapf, C.: 'An experimental determination of sufficient mutant operators', *ACM Trans. Softw. Eng. Methodol.*, 1996, **5**, (2), pp. 99–118
7 Al Hayek, G., and Robach, C.: 'From specification validation to hardware testing: A unified method'. International Test Conf., October 1996, pp. 885–893
8 von Mayrhauser, A., Chen, T., Kok, J., Anderson, C., Read, A., and Hajjar, A.: 'On choosing test criteria for behavioural level harware design verification'. High Level Design Validation and Test Workshop, 2000, pp. 124–130
9 Hajjar, A., Chen, T., and von Mayrhauser, A.: 'On statistical behaviour of branch coverage in testing behavioural vhdl models'. High Level Design Validation and Test Workshop, 2000, pp. 89–94
10 Corno, F., Sonze Reorda, M., Squillero, G., Manzone, A., and Pincetti, A.: 'Automatic test bench generation for validation of RT-level descriptions: an industrial experience'. Design Automation and Test in Europe, 2000, pp. 385–389
11 Vemuri, R., and Kalyanaraman, R.: 'Generation of design verification tests from behavioural vhdl programs using path enumeration and constraint programming', *IEEE Trans. Very Large Scale Intergr. Syst. (VLSI)*, 1995, **3**, (2), pp. 201–214
12 Paoli, C., Nivet, M.-L., and Santucci, J.-F.: 'Use of constraint solving in order to generate test vectors for behavioural validation'. High Level Design Validation and Test Workshop, 2000, pp. 15–20
13 McCabe, T.J.: 'A complexity measure', *IEEE Trans. Softw. Eng.*, 1976, **SE-2**, (4), pp. 308–320
14 Rapps, S., and Weyuker, E.J.: 'Selecting software test data using data flow information', *IEEE Trans. Softw. Eng.*, 1985, **SE-11**, (4), pp. 367–375
15 Frankl, P.G., and Weyuker, J.E.: 'An applicable family of data flow testing criteria', *IEEE Trans. Softw. Eng.*, 1988, **SE-14**, (10), pp. 1483–1498

16 Clarke, L.A., Podgurski, A., Richardson, D.J., and Zeil, S.J.: 'A formal evaluation of data flow path selection criteria', *IEEE Trans. Softw. Eng.*, 1989, **SE-15**, (11), pp. 1318–1332
17 Ntafos, S.C.: 'A comparison of some structural testing strategies', *IEEE Trans. Softw. Eng.*, 1988, **SE-14**, pp. 868–874
18 Laski, J., and Korel, B.: 'A data flow oriented program testing strategy', *IEEE Trans. Softw. Eng.*, 1983, **SE-9**, pp. 33–43
19 Zhang, Q., and Harris, I.G.: 'A data flow fault coverage metric for validation of behavioural hdl descriptions'. Int. Conf. on Computer-Aided Design, November 2000
20 White, L., and Cohen, E.: 'A domain strategy for computer program testing', *IEEE Trans. Softw. Eng.*, 1980, **SE-6**, (3), pp. 247–247
21 Zhang, Q., and Harris, I.G.: 'A domain coverage metric for the validation of behavioural VHDL descriptions'. Int. Test Conf., October 2000
22 Devadas, S., Ghosh, A., and Keutzer, K.: 'An observability-based code coverage metric for functional simulation'. Int. Conf. on Computer-Aided Design, November 1996, pp. 418–425
23 Fallah, F., Devadas, S., and Keutzer, K.: 'Occom: Efficient computation of observability-based code coverage metrics for functional verification'. Design Automation Conf., June 1998, pp. 152–157
24 Costa, J.C., Devadas, S., and Montiero, J.C.: 'Observability analysis of embedded software for coverage-directed validation'. Int. Conf. Computer-Aided Design, November 2000, pp. 27–32
25 Thaker, P.A., Agrawal, V.D., and Zaghloul, M.E.: 'Validation vector grade (VVG): A new coverage metric for validation and test'. VLSI Test Symp., 1999, pp. 182–188
26 Cheng, K.-T., and Jou, J.-Y.: 'A functional fault model for sequential machines', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 1992, **11**, (9), pp. 1065–1073
27 Moundanos, D., Abraham, J.A., and Hoskote, Y.V.: 'Abstraction techniques for validation coverage analysis and test generation', *IEEE Trans. Comput.*, 1998, **47**, (1), pp. 2–14
28 Malik, N., Roberts, S., Pita, A., and Dobson, R.: 'Automaton: an autonomous coverage-based multiprocessor system verification environment'. IEEE Int. Workshop on Rapid System Prototyping, June 1997, pp. 168–172
29 Gupta, A., Malik, S., and Ashar, P.: 'Toward formalizing a validation methodology using simulation coverage'. Design Automation Conf., June 1997, pp. 740–745
30 Ho, R.C., Yang, C.H., Horowitz, M.A., and Dill, D.L.: 'Architecture validation for processors'. *Int. Symp. on Computer Architecture*, 1995, pp. 404–413
31 Geist, D., Farkas, M., Landver, A., Ur, S., and Wolfsthal, Y.: 'Coverage-directed test generation using symbolic techniques'. Proc. Formal Methods in Computer-Aided Design FMCAD, November 1996, pp. 143–158
32 Kohavi, Z.: 'Switching and finite automata theory' (McGraw Hill, 1978)
33 Lee, D., and Yannakakis, M.: 'Principles and methods of testing finite state machines - a survey', *IEEE Trans. Comput.*, 1996, **84**, (8), pp. 1090–1123
34 Cheng, K.-T., and Krishnakumar, A.S.: 'Automatic functional test bench generation using the extended finite state machine model'. Design Automation Conf., 1993, pp. 1–6
35 Bergmann, J.P., and Horowitz, M.A.: 'Improving coverage analysis and test generation for large designs'. Int. Conf. on Computer-Aided Design, 1999, pp. 580–583
36 Abramovici, M., Breuer, M.A., and Friedman, A.D.: 'Digital systems testing and testable design' (Computer Science Press, 1990)
37 Bushnell, M.L., and Agrawal, V.D.: 'Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits' (Kluwer Academic Publishers, 2000)
38 Thaker, P.A., Agrawal, V.D., and Zaghloul, M.E.: 'Register-transfer level fault modelling and test evaluation techniques for VLSI circuits'. Int. Test Conf., 2000, pp. 940–949
39 Fin, A., Fummi, F., and Signoretto, M.: 'SystemC: A homogenous environment to test embedded systems'. Int. Workshop on Hardware/Software Codesign (CODES), 2001
40 Ferrandi, F., Fummi, F., Gerli, L., and Sciuto, D.: 'Symbolic functional vector generation for VHDL specifications'. Design Automation and Test in Europe, 1999, pp. 442–446
41 Dey, S., Raghunathan, A., and Wagner, K.D.: 'Design for testability techniques at the behavioural and register-transfer level', *J. Electron. Test. Theory Appl. (JETTA)*, 1998, **13**, (2), pp. 79–91
42 Puig-Medina, M., Ezer, G., and Konas, P.: 'Verification of configurable processor cores'. Design Automation Conf., June 2000, pp. 426–431
43 Shen, J., and Abraham, J.A.: 'Native mode functional test generation for processors with applications to self test and design validation'. International Test Conf., October 1998, pp. 990–999
44 Brahme, D., and Abraham, J.A.: 'Functional testing of microprocessors', *IEEE Trans. Comput.*, 1984, pp. 475–485
45 van de Goor, A.J., and Verhallen, Th.J.W.: 'Functional testing of current microprocessors'. Int. Test Conf., September 1992, pp. 684–695
46 Levitt, J., and Olukotun, K.: 'Verifying correct pipeline implementation for microprocessors'. Int. Conf. on Computer-Aided Design, 1997, pp. 162–169
47 Mishra, P., Dutt, N., and Nicolau, A.: 'Automatic verification of pipeline specifications'. High-Level Design Validation and Test Workshop, 2001, pp. 9–13
48 Utamaphetai, N., Blanton, R.D., and Shen, J.P.: 'Relating buffer-oriented microarchitecture validation to high-level pipeline functionality'. High-Level Design Validation and Test Workshop, 2001, pp. 3–8
49 Grinwald, R., Harel, E., Orgad, M., Ur, S., and Ziv, A.: 'User defined coverage - a tool supported methodology for design verification'. Design Automation Conf., June 1998, pp. 158–163
50 Ruf, T., Hoffmann, D.W., Kropf, T., and Rosentiel, W.: 'Checking temporal properties under simulation of executable system descriptions'. High Level Design Validation and Test Workshop, 2000, pp. 161–166
51 Rowson, J.A., and Sangiovanni-Vincentelli, A.: 'Interface-based design'. Design Automation Conf., June 1997, pp. 178–183
52 Panigrahi, D., Taylor, C.N., and Dey, S.: 'Interface based hardware/software validation of a system-on-chip'. High Level Design Validation and Test Workshop, 2000, pp. 53–58
53 Zhang, Q., and Harris, I.G.: 'A validation fault model for timing-induced functional errors'. Int. Test Conf., October 2001
54 Garey, M.R., and Johnson, D.S.: 'Computers and intractability: a guide to the theory of NP-completeness' (W. H. Freeman and Company, 1979)
55 Fallah, F., Devadas, S., and Keutzer, K.: 'Functional vector generation for hdl models using linear programming and 3-satisfiability'. Design Automation Conf., June 1998, pp. 528–533
56 Fallah, F., Pranav, A., and Devadas, S.: 'Simulation vector generation from hdl descriptions for observability-enhanced statement coverage'. Design Automation Conf., June 1999, pp. 666–671
57 Larrabee, T.: 'Test pattern generation using boolean satisfiability', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 1992, **11**, pp. 4–15
58 Zeng, Z., Kalla, P., and Ciesielski, M.: 'Lpsat: A unified approach to rtl satisfiability'. Design, Automation and Test in Europe Conf., 2000
59 Van Hentenryck, P.: 'Constraint satisfaction in logic programming' (MIT Press, 1989)
60 Zeng, Z., Ciesielski, M., and Rouzeyere, B.: 'Functional test generation using constraint logic programming'. VLSISOC Conf., 2001
61 Diaz, D.: GNU Prolog: A native prolog compiler with constraint solving over finite domains. The GNU Project, www.gnu.org, 1999
62 Jaffar, J., Michaylov, S., Stuckey, P.J., and Yap, R.H.C.: 'The CLP(R) language and system', *ACM Trans. Program. Lang. Syst.*, 1992, **14**, (3), pp. 339–395
63 Xin, F., and Harris, I.G.: 'Test generation for hardware-software covalidation using non-linear programming'. High-Level Design Validation and Test Workshop, 2002
64 Ferrandi, F., Fummi, F., and Sciuto, D.: 'Implicit test generation for behavioural VHDL models'. Int. Test Conf., October 1998, pp. 587–596
65 Ferrandi, F., Ferrara, G., Sciuto, D., Fin, A., and Fummi, F.: 'Functional test generation for behaviorally sequential models'. Design Automation and Test in Europe, March 2001, pp. 403–410
66 McMillan, K.L.: 'Symbolic model checking' (Kluwer Academic Publishers, 1993)
67 Touati, H.J., Savoj, H., Lin, B., Brayton, R.K., and Sangiovanni-Vincentelli, A.: 'Implicit state enumeration of finite state machines using BDD's'. Int. Conf. on Computer-Aided Design, November 1990, pp. 130–133
68 Corno, F., Prinetto, P., and Sonza Reorda, M.: 'Testability analysis and ATPG on behavioural RT-level VHDL'. In Int. Test Conf., 1997, pp. 753–759
69 Lajolo, M., Lavagno, L., Rebaudengo, M., Sonza Reorda, M., and Violante, M.: 'Behavioural-level test vector generation for system-on-chip designs'. High Level Design Validation and Test Workshop, 2000, pp. 21–26
70 Yuan, J., Shultz, K., Pixley, C., Miller, H., and Aziz, A.: 'Modelling design constraints and biasing in simulation using BDDs'. Int. Conf. on Computer-Aided Design, 1999, pp. 584–589
71 Hoare, C.A.R.: 'Assertion: A personal perspective', *IEEE Ann. Hist. Comput.*, 2003, **25**, (2), pp. 14–25
72 Rosenblum, D.S.: 'A practical approach to programming with assertions', *IEEE Trans. Softw. Eng.*, 1995, **21**, (1), pp. 19–31
73 Foster, H.D., Krolnik, A.C., and Lacey, D.J.: 'Assertion-based design' (Kluwer Academic Publishers, 2003)
74 Kantrowitz, M., and Noack, L.M.: 'I'm done simulating: Now what? verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor'. Design Automation Conf., June 1996, pp. 325–330
75 Taylor, S., Quinn, M., Brown, D., Dohm, N., Hildebrandt, S., Huggins, J., and Ramey, C.: 'Functional verification of a multipleissue, out-of-order, superscalar alpha processor-the dec alpha 21264 microprocessor'. Design Automation Conf., June 1998, pp. 338–343
76 Foster, H., and Coelho, C.: 'Assertions targeting a diverse set of verification tools'. Int. HDL Conf., March 2001
77 Chen, X., Luo, Y., Hsieh, H., Bhuyan, L., and Balarin, F.: 'Utilizing formal assertions for system design of network processors'. Design Automation and Test in Europe, 2004
78 Accelera proposed standard property specification language (psl) 1.0, January 2003
79 Clarke, E.M., Khaira, M., and Xiao, X.: 'Word level model checking - avoiding the pentium fdiv error'. Design Automation Conf., June 1996, pp. 245–248
80 Chauhan, P., Clarke, E.M., Lu, Y., and Wang, D.: 'Verifying ip-core based system-on-chip designs'. ASIC/SOC Conf., 1999, pp. 27–31
81 Castelnuovo, A., Fedeli, A., Fin, A., Fummi, F., Pravadelli, G., Rossi, U., Sforza, F., and Toto, F.: 'A 1000$\times$ speed up for properties completeness evaluation'. High Level Design Validation and Test Workshop, 2002, pp. 18–22
82 Shimizu, K., and Dill, D.L.: 'Using formal specifications for functional validation of hardware designs', *IEEE Des. Test Comput.*, 2002, **19**, (4), pp. 96–106

83 Chen, L., and Avizienis, A.: '*N*-version programming: a fault-tolerance approach to reliability of software operation'. Fault Tolerant Computing Symp., 1978, pp. 3–9
84 Avizienis, A., and Kelly, J.P.J.: 'Fault tolerance by design diversity: Concepts and experiments', *Computer*, 1984, **17**, (8), pp. 67–80
85 Leveson, N.G., Cha, S.S., Knight, J.C., and Shimeall, T.J.: 'The use of self checks and voting in software error detection: An empirical study', *IEEE Trans. Softw. Eng.*, 1990, **16**, (4), pp. 432–443
86 Blum, M., and Kanna, S.: 'Designing programs that check their work'. Annual ACM Symp. on Theory of Computing, 1989, pp. 86–97
87 Blum, M., Luby, M., and Rubinfeld, R.: 'Self-testing/correcting with applications to numerical problems'. Annual ACM Symp. on Theory of Computing, 1990, pp. 73–83
88 Segura, J., Keshavarzi, A., Soden, J.M., and Hawkins, C.F.: 'Parametric failures in CMOS ICS - a defect-based analysis'. Int. Test Conf., 2002, pp. 90–99
89 Shivakumar, P., Kistler, M., Keckler, S.W., Berger, D., and Alvisi, L.: 'Modelling the effect of technology trends on the soft error rate of combinational logic'. Int. Conf. Dependable Systems and Networks, 2002
90 Tasiran, S., and Keutzer, K.: 'Coverage metrics for functional validation of hardware designs', *IEEE Des. Test Comput.*, 2000, **17**, (4), pp. 51–60
91 Gaudette, E., Moussa, M., and Harris, I.G.: 'A method for the evaluation of behavioural fault models'. High Level Design Validation and Test Workshop, November 2003, pp. 169–172