Some Novel Methods for Design Intent Verification

Thesis Submitted In Partial Fulfillment Of The Requirements
For The Degree Of

Master of Technology
in
Computer Science and Engineering

by

Arnab Sinha

Roll No: 02CS3022

under the guidance of

Dr. Pallab Dasgupta

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur
May 2007

Certificate

This is to certify that the thesis titled **Some Novel Methods for Design Intent Verification** submitted by **Arnab Sinha** to the Department of Computer Science and Engineering in partial fulfillment for the award of the degree of **Master of Technology** is a bonafide record of work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this Institute and, in our opinion, has reached the standard needed for submission.

Dr. Pallab Dasgupta
Dept. of Computer Science and Engineering
Indian Institute of Technology
Kharagpur 721302, INDIA
May 2007

Acknowledgments

This thesis is the result of research performed under the guidance of **Dr. Pallab Dasgupta** at the Department of Computer Science and Engineering of the Indian Institute of Technology, Kharagpur.

I am deeply grateful to my supervisor for having given me the opportunity of working as part of his research group and the huge amount of time and effort he spent guiding me through several difficulties on the way. Without the help, encouragement and patient support I received from my guide, this thesis would never have materialized. Besides him, I am grateful to **Prof. Rainer Leupers**, Institute for Integrated Signal Processing Systems (ISS), University of Technology, Aachen, Germany for the part of the work done there.

I also acknowledge my senior **Bhaskar Pal** for his encouraging suggestions, sharing the technical skills and the synergy that he brought in our work. My acknowledgments to all the members of Formal-V group, IIT Kharagpur and **Anupam Chattopadhyay** (Institute for Integrated Signal Processing Systems (ISS), University of Technology, Aachen, Germany) for their technical inputs and constant support in my work. Further, I am grateful to my parents for their perennial inspiration.

Arnab Sinha
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
May 2007

Abstract

With the diminishing size and cost of the fabrication of semiconductor chips, the embedded processors are rapidly gaining popularity in various domains. The recent innovations made in the embedded processors and Application Specific Instruction-set Processors (ASIPs) are indicating that the success of a particular product largely depend on the architectural design novelties. The challenge is to ensure whether these architectural decisions are correctly implemented in the RTL blocks. There are two main approaches for expressing the architectural intent, namely - *Architecture Description Languages* (ADLs) and *formal properties*. Unfortunately, there exists no formal method for verifying whether the architectural design intent is correctly translated into the lower levels of the design. In this project, we have attempted to analytically find out test-patterns from the processor behavior description, as described in the ADL based processor design methodology in presence of a integrated semi-formal assertion based verification platform. Moreover, we also investigated the intent coverage question over hybrid specifications consisting of state machine and properties.

Contents

List of Figures

Chapter 1

Introduction

Recent advances in the semiconductor technology has enabled the electronics industry with the dream of realizing more ambitious design goals. The industry and academia have put up a concerted effort of realizing entire systems comprising of multiple units in a single chip (which we nowadays often call as System-on-Chip or SoC).The current state-of-the-art in the domain of digital system design is continuously and rapidly evolving with the advent of the most sophisticated tools and design methodologies. However, this enormous growth has given birth to significant challenges for the developers of tools for electronic system design.

With all the progress in the fabrication technology and the *Computer-Aided-Design* (CAD) tools, there is an observable growth in the market of embedded processors and the *Application Specific Instruction-set Processors* (ASIPs). There is a huge demand of these processors in different domains. It is interesting to note that nowadays the major innovations in these processors are taking place in the architecture level (refer Fig 1.1). In the architecture level the decisions regarding the pipeline depth, caching strategy, priority arbitration, efficient power management etc are to name a few. These architectural decisions are the **design intent** of the system-designer. These design intents are like the pillars of any successful implementation of an intended system.

Verification of these large systems is still a major bottleneck to the Electronic Design Automation (EDA) industry. Although the demand for the processors is large, due to short time-to-market span, there is a huge competition among the vendors. All these factors make the situation more complex. In the technical front, the capacity limitations of the various verification methodologies largely inhibit the idea of verifying the whole system simultaneously. Hence, industry resorts to the verification of the smaller RTL

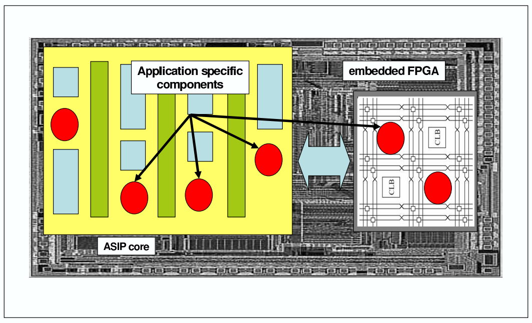


Figure 1.1: Various innovations in the architecture-level

blocks only. Through the verification of the *local* properties, the validation engineer tries to guess whether the *global* intent has been implemented in the RTL. For this purpose, the design intent or the architectural specifications further need to be refined while designing the smaller RTL modules. (see Fig 1.2) These refinements often give birth to the gaps in the specifications of the two levels. The gaps so formed can be detrimental as the intent is getting modified. A gap in the specification represents runs which refutes the RTL specs, although validates the architectural specs. This is natural since refinements appearing in the RTL (where detailed design is available) may reject certain runs, which successfully passed the architectural specs. However, complex design bugs may reside in these specification lacuna.

1.1 Different Design Intent Expression Styles

The design intent expression is usually made from a higher abstraction level. The abstraction level should be higher enough such that the precise details of the design is not visible in this level, however the level should also enable the designer to understand how the major blocks in the hardware is operating. There are two broad classifications of the design intent expression (refer Fig 1.3). These two approaches are namely,

- **Model Based Approach:** The model-based approach attempts to verify the RTL with respect to the architecture description (which is indeed the design intent). The architecture is usually described in the different architecture description languages like Language for Instruction-Set Architecture(LISA) [2], EXPRESSION[3]

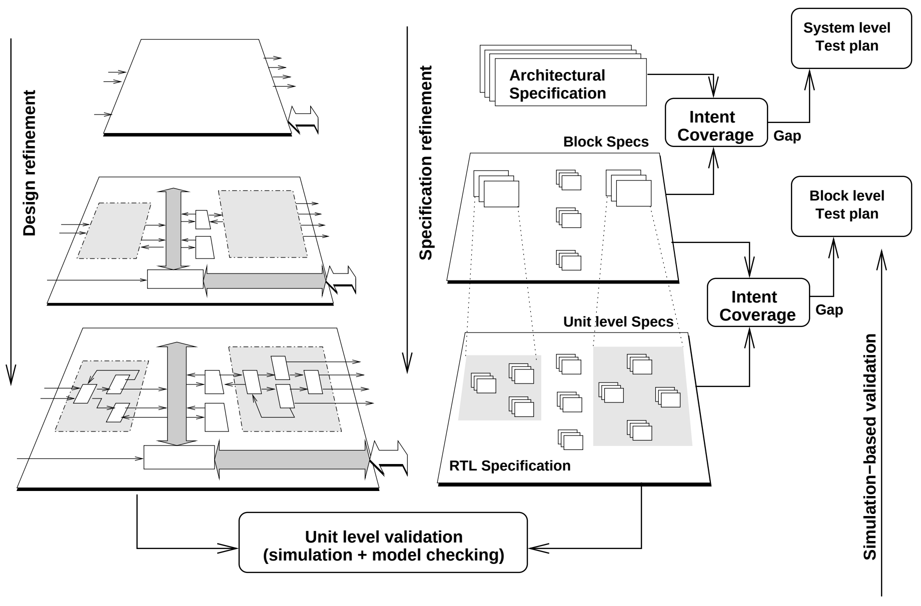
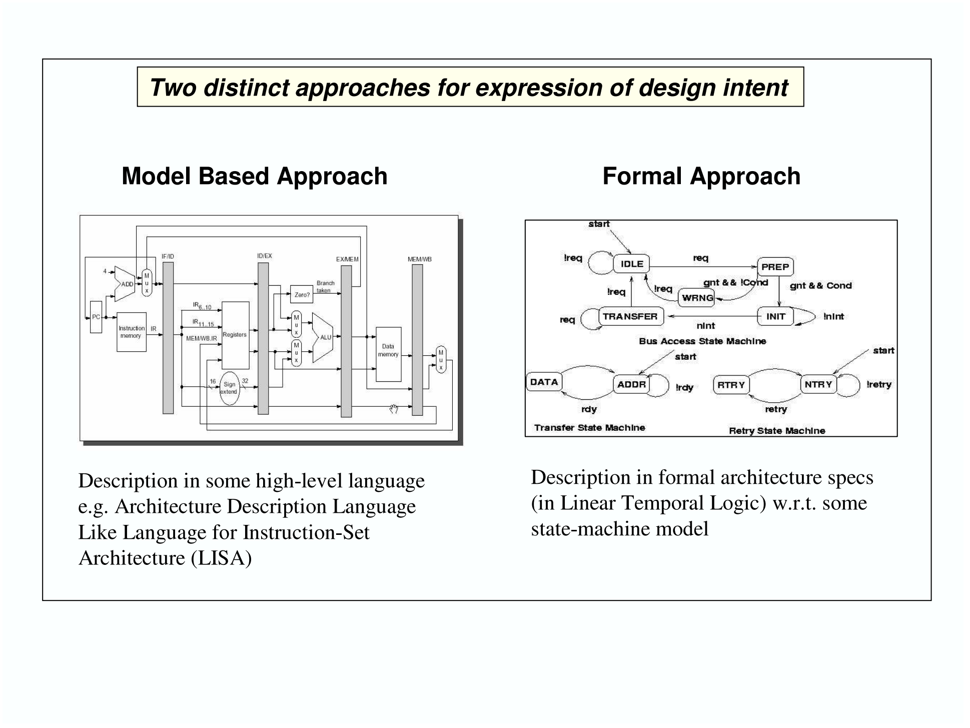


Figure 1.2: Verification of Design Intent. Source: [1]


Two distinct approaches for expression of design intent

Model Based Approach

Formal Approach

start
!req
IDLE
req
PREP
gnt && !Cond
gnt && Cond
!req
!req
WRNG
req
TRANSFER
INIT
!nint
nint
Bus Access State Machine
start
start
DATA
ADDR
!rdy
RTRY
NTRY
!retry
rdy
retry
Transfer State Machine
Retry State Machine

Description in some high-level language e.g. Architecture Description Language Like Language for Instruction-Set Architecture (LISA)

Description in formal architecture specs (in Linear Temporal Logic) w.r.t. some state-machine model

Figure 1.3: Distinct approaches in design intent expression

etc. Here test-cases are churned out of the ADL description targeted for the RTL.

- **Formal Approach:** In the formal approach, the specifications are written in formal languages like Linear Temporal Logic (LTL) etc. Both the properties (architecture and RTL) are written formally. The architectural properties are written assuming a state-machine for the architecture, which is fictitious and transparent to the RTL. In the formal analysis the gap is also represented in the form of a set of LTL properties only.(refer Fig 1.4)

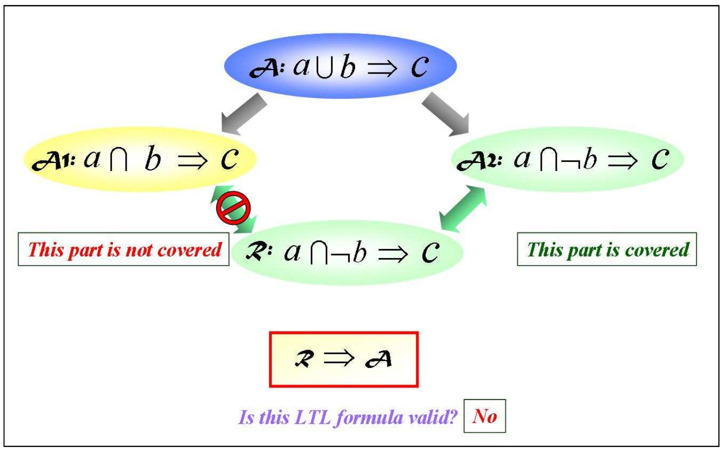

$\mathcal{A}$: $a \cup b \Rightarrow c$
$\mathcal{A}1$: $a \cap b \Rightarrow c$
$\mathcal{A}2$: $a \cap \neg b \Rightarrow c$
This part is not covered
$\mathcal{R}$: $a \cap \neg b \Rightarrow c$
This part is covered
$\mathcal{R} \Rightarrow \mathcal{A}$
Is this LTL formula valid?
No

Figure 1.4: Exemplary gap in formal property

1.2 Problem Formulation

We address the design intent verification problem from both the processor design approaches we discussed. Hence, the problem that we are addressing in this thesis can be broadly classified into two parts which are as follows.

1. **In the ADL Front:** In the ADL based framework the designer starts designing from the high-level design abstraction (i.e. architecture level). From that description, the final RTL is generated in standard Hardware Description Languages like Verilog, VHDL. Given the high-level description of the processor behavior and the generated RTL, the task is to look into the possibility of integrating a semi-formal verification platform utilizing the ADL framework, such that, this platform will enable the processor designer to ensure whether the design intent (i.e. the architectural intent) has been properly translated into the RTL design and to trap the hidden bugs (if any) in the processor description. In case of presence of any bug in the RTL, the designer should be able to refine the architectural specifications easily, to remove the bug during future design-space exploration.

2. **In the Formal Property Front:** Due to rising complexity of the modern designs, the properties are more complex and involved which incorporate several cycle-accurate dependencies (which are hard to express without introducing the concept of auxiliary state-machine). Given such context-sensitive architectural and RTL block-level specifications in the form of properties, our challenge was to compute and represent the gap (if it exists) between the specifications of these two levels.

1.3 Contribution of the Thesis

The contributions of this thesis are the following.

1. We implemented an ADL-driven assertion based verification methodology integrated with Language for Instruction Set Architecture (LISA). The seamless integration with the already existing ADL platform, enabled the automatic tool-flow to achieve high RTL-statement coverage. Moreover, we propose a novel backtracking algorithm for functional test generation of pipelined processors.

2. We studied the relation between satisfiability (in connection with design intent coverage) and model checking and show that they are complexity-wise similar. This relation enabled us to address the coverage question over hybrid specifications consisting of state machine and formal properties. Moreover, we present new algorithms and heuristics for specification refinement that can handle auxiliary state machine specifications.

1.4 Thesis Organization

The thesis is organized as follows.

Chapter 2: The concept, various heuristics and the algorithms of test-case generation from processor description (in LISA) for the complete coverage of the conditional blocks in the RTL are presented in this chapter.

Chapter 3: This chapter formulates the new problem in formal design intent verification approach and answers the primary coverage question along with the heuristic solution to the problem.

Chapter 4: The last chapter presents the future work as well as the outlook of the work done.

In the appendix we have presented the syntax and semantics of the Linear Temporal Logic which is central to the understanding of the work presented in Chapter 3. We have also listed the papers there, either published or communicated containing the novel contributions made in this project so far.

Chapter 2

Model Based Intent Verification

2.1 Introduction

The Architecture Design Languages (ADL) [4, 5, 6] nowadays are offering promising avenues for fast design-space exploration with enough room for optimization for target-specific architectures. The advantages offered by the ADL-based design are as follows:

- *faster design-space exploration*
- *seamless integration of the components* through the automatic generation of the software tool-chain (simulator, HLL compiler, assembler etc) as well as the RTL description of the processor
- *the higher abstraction level* which helps in doing away with the details of the implementation.

These merits coupled with the narrowing application-domains for the processors, encourage modeling of the Application Specific Instruction-Set Processor (ASIP) using the ADLs [7].

However, with the growth in design complexity and short time-to-market, the designer needs to deliver the optimum performance in a short time without compromising the verification issues. Next, we discuss the recent approaches in processor design with the focus in verification. There are broadly two distinct trends in processor design approach.

- **Higher Abstraction-based Design:** At one extreme, optimal processor performance is obtained through introduction of special instructions in the Instruction-Set Architecture (ISA). Here, the designer has the choice of describing the processor behavior starting from scratch. ADLs like LISA [4], EXPRESSION [5], mimola [8] etc. allow such high flexibility, nML [6]. But this approach compromises with the verification cost.

- **Template-based Design:** The processor design is simplified by limiting the choice of design within a range of pre-verified IPs or allowing fine-tuning of an existing template processor core [9, 10]. This *template-based approach* allows quick verification with limited choice of application-specific optimizations. Here, the verification concern is focused on the possible designer-defined configurations or the extensions of the basic template.

It is clear that, in the template-based approach, in order to reach the high flexibility like ADLs, more and more configurations need to be supported, eventually increasing the verification effort. Hence, there is a need for trade-off between optimization and verification effort.

Independent of the two separate approaches, verification has received continuous attention from Electronic System Level (ESL) design community. With the evolving processor design concepts like pipelining and VLIW, exhaustive manual testing no longer remained feasible. Due to the pressing demand from industry as well as academia, several initiatives have been taken to integrate the verification efforts with the high-level design flow. With the traditional processor design approach, the verification is primarily done in the Register Transfer Level (RTL) or below [11] and the tools used mostly simulation-based approaches [12]. Unfortunately, in the context of ADLs, the RTL-based verification is inadequate. It is because the ADL-model has to be synthesized to generate the RTL and then RTL-based verification methodologies need to be applied. In this approach, the ADL-based information about the architecture is often lost in the synthesis process, thereby, making the RTL-driven verification process imprecise and computationally complex.

Nevertheless, with the advent of hardware verification languages like *Vera*, *e*, designed for automatic verification environment, the simulation-based approaches have been blended with the formal verification techniques. The major contemporary processor verification endeavors are high-level simulation [13], sequential logic equivalence checking [14], assertion-based semi-formal verification [15], property-driven verification [16, 17]. For two of the major verification approaches, namely, assertion-based verification and simulation-based verification, test-pattern is the key component to drive the verification. Hence, *au-*

tomatic functional test-generation has attracted considerable focus from the researchers in recent years. In the perspective of processor verification, functional test-generation intelligent efforts can be broadly classified under the following heads: *coverage-driven approaches* and *property-driven approaches.*

Coverage-driven Approach: Coverage-driven approaches strongly rely on the feedback of the coverage to the test-generation mechanism. Exemplarily, Corno et al [18] proposed a genetic algorithm-based framework to build up the test pattern on the basis of coverage feedback. This approach demonstrated 100% RTL statement coverage for a sparc-compatible processor. However, the genetic algorithm required simulation of up to 7.3 million instructions to obtain the compact instruction-set showing 100% RTL statement coverage. Clearly, the runtime of the test generation is a big drawback here. Fast convergence of the algorithm is dependent on the choice of parameters like mutation, cross-over and effective fitness criteria. The analytical approach presented by Luethje [19] is one prominent approach to achieve complete statement coverage from an high-level description. Luethje performed an abstract execution of the conditional blocks of an ADL description and showed full coverage of the ADL statements. However, his algorithm works for instruction-accurate processor models only i.e. the algorithm is not scalable to pipelining effects.

Property-driven Approach: The property-driven approaches are mostly analytical in nature. The processor model is transformed into some form of abstraction (e.g. a graph) and different sophisticated verification techniques (e.g. model-checking) are then applied on it. In these analytical approaches, the instructions which excite the processor resources and/or operators are chosen systematically [20]. The operands are selected to reflect different scenarios like data hazards or control hazards in a pseudo-random manner. Often, manual intervention or good heuristics play an important role in determining the quality of the test-patterns. IBM Genesys Test Generation framework [21] is one such example. In this test generation process, the verification engineer guides the generation process via a Graphical-User-Interface. IBM Genesys framework is shown to be quite useful to reveal corner case bugs, although full coverage of the RTL implementation of the processor is not explicitly targeted. In [22], a HDL-satisfiability checker algorithm is developed to determine stimuli for exciting selected paths in an HDL model. This is close to the solution approach presented in this thesis.

Notable efforts to automate test generation from high-level description are presented by Mishra et al [23, 24, 25]. In [24, 3] Mishra et al proposed a new functional coverage metric on the basis of a graph model of the processor. The functional coverage metric include scenarios like pipeline execution, register read/write, operation execution etc. They transformed a processor description, written in EXPRESSION ADL [26], into a graph consisting of nodes (representing *functional units* and *storages*), and edges (representing *pipeline* operations and *data-transfer* operations). By the help of that graph, they defined a graph-based coverage metric, which in turn, led to a set of test-patterns. A drawback of this approach is that, it considers the operations *atomically* as nodes of the graph and the input/output operands are loaded/stored by specific processor instructions, which are known in advance. If an operation contains deeply nested data-flow within it, this approach may not cover the execution of all conditions within it. In [23] an ADL representation was successfully transformed into an equivalent SMV representation and used SMV-based model checking tools to generate test-patterns. This approach strongly relies on bounded model checking, which is computationally intensive. In another extension of this work, the properties are automatically generated from the ADL description and decomposed [16] to speed up the model checking. However, the properties are derived from atomic operations, which is rather simplistic. It will be interesting to apply these methods for irregular procesor architectures, commonly found in the embedded processor domain.

In summary, there are existing ADL-based verification approaches. However, none of the above-mentioned ADLs support *automatic generation of assertions from the ADL* and *automatic generation of test-patterns for the full-coverage of conditional blocks for a cycle-accurate processor implementation.* The biggest hindrance to such an attempt is the complexity of the processor itself. ADL EXPRESSION used assertions during their top-down validation flow. In their approach, the ADL description automatically generates an RTL description. A symbolic simulator automatically generates property/assertion from the generated RTL description [25]. The drawback with this approach is that the generation of property/assertion is done from the automatically generated RTL and not from the ADL. Therefore, this approach heavily relies on the correct generation of the RTL from the ADL. Understandably, in their approach the automatically generated RTL is considered as golden reference, which is used to verify manually implemented RTL design. In our approach, we take it a step higher and try to verify the correctness of the ADL description itself. Therefore, it is imperative to automatically generate the assertions from the ADL, as proposed in this thesis. This thesis closely follows the work presented in [27, 28]. In summary, the contribution of this thesis is to present:

- An ADL-driven assertion-based verification methodology.
- An ADL-based automatic tool-flow to achieve high RTL-statement coverage.
- A novel backtracking algorithm for functional test generation of pipelined processors

Our work complements the existing ADL-based processor development flow by adding the verification flow. The components of the verification framework are completely derived from the ADL description, thereby minimizing the chances of design inconsistency.

The Organization of the chapter
The chapter is organized as follows. The next section (section 2.2) provides with an overview of the ADL LISA. Section 2.3 presents the issues regarding the test generation in LISA, followed by LISA-based assertion generation in section 2.4. In section 2.5, we discuss the integrated verification flow in LISA. Results of the case-studies are presented in section 2.6 with the conclusion in section 2.7.

2.2 Brief Overview of LISA

In this section, a brief overview of the architecture description language LISA is provided. Only those language elements, which are relevant for this thesis are covered here. For interested readers, a detailed discussion of LISA language and related tools can be found at [4].

2.2.1 LISA Operation Graph

In LISA, an *operation* is the central element to describe the timing and the behavior of a processor instruction. The instruction may be split among several LISA operations. The resources (registers, memories, pins etc.) are declared globally in the *resource section*, which can be accessed from any LISA operation.

The LISA description is based on the principle that a specific common behavior or common instruction encoding is described in a single operation whereas the specialized behavior or encoding is implemented in its *child* operations. With this principle, LISA

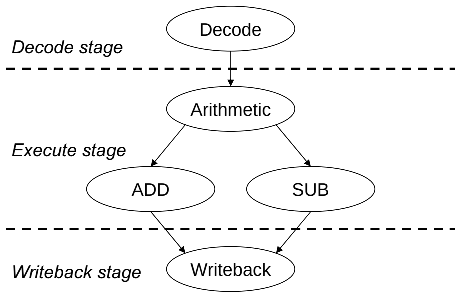


Figure 2.1: LISA Operation DAG Example

operations are basically organized as an n-ary tree. However, specialized operations may be referred to by more than one *parent* operation. The complete structure is a Directed Acyclic Graph (DAG) $\mathcal{D} = \langle V, E \rangle$. V represents the set of LISA operations, E the graph edges as set of child-parent relations. These relations represent either *Behavior Calls* or *Activations*, which refer to the execution of another LISA operation. Figure 2.1 gives an example of a LISA operation DAG. As shown, the operations can be distributed over several pipeline stages. A chain of operations, forming a complete branch of the LISA operation DAG represents an instruction in the modelled processor.

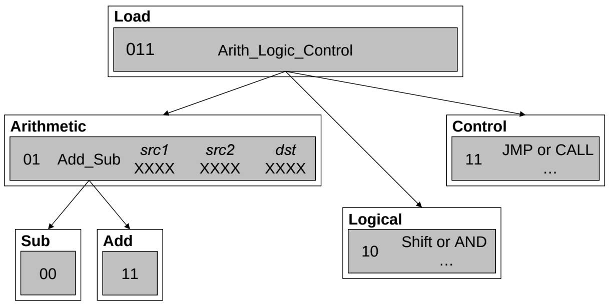


Figure 2.2: LISA Coding Tree Example

A LISA Operation contains different subsections to capture the entire processor behavior. The ones relevant for this thesis are discussed below.

Instruction Coding Description: The instruction encoding of a LISA operation is described as a sequence of *coding fields*. Each coding field is either a terminal bit sequence with “0”, “1”, “don’t care”(X) bits or a nonterminal bit sequence referring to the coding field of another child LISA operation. An example of a coding tree is given in figure 2.2. In this example, the *Add* and *Sub* operations have only terminal codings whereas *Load*, *Arithmetic*, *Logical* and *Control* consist of both terminal and nonterminal coding fields.

Activations: A LISA operation can *activate* other operations in the same or a later pipeline stage. In either case, the child operation may be activated *directly* or via a *group*. A *group* collects several mutually exclusive LISA operations.

Behavior Description: The behavior description of a LISA operation corresponds to the datapath of the processor. Inside the behavior description plain C code can be used. Resources such as registers, memories, signals and pins as well as coding elements can be accessed in the same way as ordinary variables. The behavior section of every *LISA operation* is transformed into a *functional block* in the RTL datapath.

2.2.2 Data Flow Graph (DFG) Representation

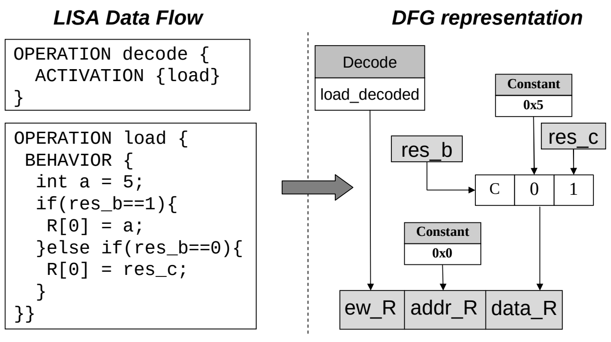
The data flow is captured, on the high abstraction level, by the dependency of LISA operations (also across the pipeline stages). Inside a single LISA operation, the *behavior* section guides the data flow. The behavior section of a LISA operation is converted into a pure, directed Data Flow Graph (DFG) ($G_{DFG} = \langle V_{op}, E_{ic} \rangle$) . The vertices ($V_{op}$) are the basic *operators* for data manipulation e.g. addition, while edges (E_{ic}) represent the flow of unchanged data in form of *interconnections* of inputs and outputs.

Operators: The following list summarizes the basic classes of operators represented by graph vertices.

- Commutative n-ary Operator (`ALU_OP`), $n \geq 2$
- Noncommutative n-ary Operator (`ALU_OP`), $n \geq 1$
- Read/Write Access to Registers (`RESOURCE`, `PIPE_RESOURCE`) and Memories
- Decoding and Control Signal (`DECODE, FLUSH, STALL`)
- Multiplexer

Note that, unary operators are treated as a special case of Noncommutative n-ary operator.

Interconnections: Interconnections represent the data flow on symbol-level representations. The information about the data type transferred is given by an annotation to the interconnection. Bit range subscriptions are included into the interconnection information, too.



```
OPERATION decode {
  ACTIVATION {load}
}

OPERATION load {
 BEHAVIOR {
  int a = 5;
  if(res_b==1){
   R[0] = a;
  }else if(res_b==0){
   R[0] = res_c;
  }
}}
```

Figure 2.3: DFG Representation

The creation of the DFG from the plain C-code of a LISA operation's behavior section is shown in figure 2.3. As depicted there, the DFG is constructed after performing basic compiler-like optimizations. In this case, the constant value of local variable a is propagated. For the read access to non-array registers e.g. *res_c*, we need not pass any address value. For the write access to a one-dimensional resource R, the write enable and the address value is set in the scope of the same vertex. The value for write enable is set to *load_decoded*, which indicates that this operation is being executed.

2.2.3 Instruction-Grammar:

The instruction grammar represents the valid instructions in Backus-Naur Form (BNF) grammar. Table 2.1 shows an exemplary *instruction grammar*. For this example, the instruction word width is 32 bit and there are 16 available registers indexed by `src_reg` and `dst_reg`. Note that, this instruction grammar is essentially an instruction coding tree (like figure 2.2) represented in another form.

`insn`	: `add dst_reg src_reg src_reg` || `sub dst_reg src_reg src_reg` || `nop`
`add`	: `0000 0001`
`sub`	: `0000 0010`
`src_reg`	: `0000 xxxx`
`dst_reg`	: `0000 xxxx`
`nop`	: `0000 0000 0000 0000 0000 0000 0000 0000`

Table 2.1: Exemplary Instruction Grammar

2.2.4 Comparison with other ADLs

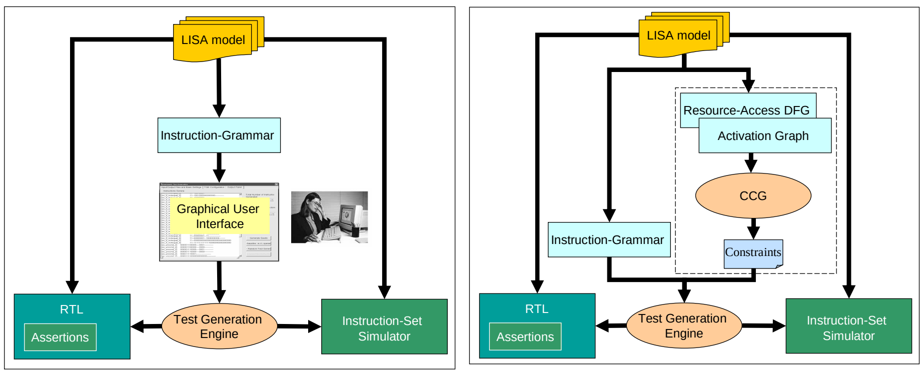
Other prominent ADLs do have access to similar architectural information like LISA. For example, the instruction coding is available in the *image* attribute of the language nML [6]. The sub-instructions are composed using *AND-rules* and *OR-rules*, which correspond to the *groups* in LISA. In nML, the datapath is modelled using the attribute *action*, inside which the global storage elements can be accessed. The ADL EXPRESSION [26] contains attributes like *op_group*, *opcode* and *behavior*, which resemble the *group*, *coding* and *behavior* section in LISA respectively. The *pipeline/data-transfer* edges in EXPRESSION serve similar purpose like *activations* and resource access in LISA. In essence, the existing ADLs do have similar architectural information and our methodology can be generically applied to any of the ADLs.

2.3 LISA-based Test Generation

The LISA-based test generation can be broadly classified under two categories. Those two major test-generation approaches are shown in fig2.3. The test-cases can be generated without (fig 2.4(a)) or with (fig 2.4(b)) some amount of preprocessing of the processor description. We call this preprocessing as *Coverage Constraint Generation (CCG).* This automatic preprocessing is necessary, whenever, test-case generation for the coverage of RTL statements is in question (because that raises the need for analysis of the ADL behavior). The designer with the intent of testing a *specific* processor-behavior need not analyze the whole ADL model. Hence, she should skip the unnecessary preprocessing. This is the reason that we have kept the Test Generation Engine (TGE) and CCG *decoupled* from each other. In summary, there are two modes of test generation in LISA. In one mode, TGE can be executed without the co-operation of CCG, while in the other mode, execution of TGE follows the analysis of CCG. In the following subsections, both the modes are described.

2.3.1 Test Generation without Preprocessing: Test Generation Engine (TGE)

This mode allows the Test Generation Engine (TGE) to operate independently without depending on Coverage Constraint Generation (CCG). Figure 2.4 explains the environment of the TGE. The easy-to-use Graphical User Interface of TGE (fig 2.4(b)) receives



(a) Test Generation without CCG (b) Test Generation with CCG

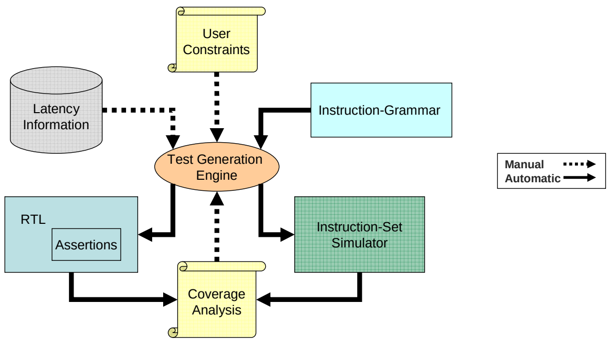


Figure 2.4: Test Generation Engine

the user constraints, the *instruction-latency information* and the instruction-grammar to produce test-cases for the RTL as well as instruction-set simulators. Although, the generated test-cases are not meaningful applications, with careful directives, as we will show later, the TGE mimics the realistic application-like scenario. In this regard, the TGE can effectively be used by a designer (as a stand-alone tool without the support of the CCG) to prepare *synthetic testbenches* for any given processor. The ADL code coverage (obtained from the instruction-set simulation), the RTL statement coverage (obtained from the RTL simulation) and the latency information are manually fed to influence the nature of generated test patterns.

In the following, a particular feature of TGE is demonstrated with an example. For *branch* instructions, it is imperative to control the branch location with respect to the current instruction without losing the randomness. By using the technique of *grammar-rule overriding*, one can modify/add rules to the existing grammar. As shown in table 2.2, the `branch_insn` rule of the grammar is modified by the rule in the following line. There the branch destination is restricted between the program start location and program end location. Furthermore, the branch destination can be set to be in a location offset (positive or negative) to the current program instruction.

```
branch_insn  : 0b01 branch_cond branch_dest
bypass_insn  : 0b01 branch_cond [prog_start, offset, prog_end]
```

Table 2.2: Overriding rules

Automatic Test-Pattern Generation: For automatic generation of the test-pattern, the instruction grammar is loaded into an internal Directed Acyclic Graph (DAG). Test patterns are generated by traversal of this DAG. The nodes and edges of the data-structure is appropriately tagged with the user-defined constraints. For example, to have an instruction with high occurrence frequency, the edge leading to the instruction is traversed with higher probability than the other edges. In case of an overridden grammar rule, an entire sub-branch of the data-structure is replaced with the DAG of the new rule. The supports provided by the TGE necessitates traversal of the instruction grammar which is in essence a DAG. Thus the worst-case time-complexity is same as that of performing DFS in a graph i.e. $\mathcal{O}(n \times e)$, where n and e are the cardinalities of set of vertices and edges in the graph, respectively.

Features of the TGE: Table 2.3 briefly describes the salient features of TGE along with the utility of the respective features [27].

Feature	Description	Purpose
Fully Exhaustive Test-cases	generates all possible instructions for the ASIP (the possibilities are finite)	for complete exploration
Size Constraint	generate given number of instructions	allows precise control over occurrence-frequency, context etc
Storage Access Biasing	generates instructions with given bias values for intended registers/memory	allows one to verify data forwarding in ASIPs
Instruction Group Occurrence Frequency Biasing	generates instructions with specified occurrence frequency biasing over a defined instruction-group, e.g. load-store	allows one to mimic situations where one instruction-group appears more than others
Instruction Occurrence Frequency Biasing	generates random test-cases with biases on specific instructions	allows designer to bias at finer granularity-level
Selection of Instructions	generates instructions for given instructions only	allows observation of processor behavior for a specific instruction
Latency Information Insertion	given number of `nop`'s follow the specified instruction	some instructions block resources for a given no. of m/c cycles
Grammar Rule Overriding	Modify the existing instruction grammar by changing one or more grammar rule	for biasing specific instruction(s).

Table 2.3: Features of TGE

2.3.2 Test Generation with Pre-processing: Coverage Constraint Generation (CCG)

In this mode, as the test-cases targets high RTL statement coverage, the process of generating such test-cases requires understanding of the processor datapath, which in turn is captured in the behavior section of the ADL description in high-level. This factor motivates the analysis of the ADL model in order to generate a set of *constraints*. A *constraint* at a given cycle and pipeline-stage is defined as a collection of resource bias(es) (biases as described in [27]) and/or assertion of some activation signal(s) at that cycle and pipeline-stage. This analysis is done in the Coverage Constraint Generation (CCG) stage.

In order to attain full statement-coverage in the RTL, it is a prudent idea to target the coverage of the conditional blocks in it. These conditional blocks in the RTL is represented as the muxes in the corresponding data-flow graph of the ADL behavior section (refer figure 2.3). So our problem reduces to generating test-patterns which verifies individual muxes, and verification of a mux is possible if it is excited with all the different control-sequences in the muxes. In the following subsubsections we first clearly state the inputs to our developed tool and outputs we target to deliver.

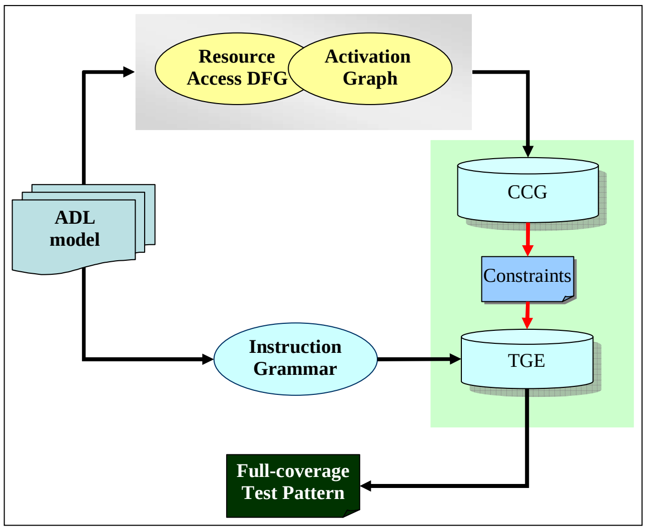


Figure 2.5: CCG Enviroment

Inputs to the CCG:

The inputs to the CCG are the following (refer figure 2.5).
1. The **DFG**(G_{DFG}) described in section 2.2.2.
2. The **LISA Operation DAG**($\mathcal{D}$) described in section 2.2.1.
The output of the CCG is the *constraint* which is a set of LISA operations and/or instruction-register biases with the desired immediate value to cover a particular mux in DFG.

The Algorithm

In this section, we describe the algorithm used in the CCG. The algorithm is backtracking in nature. Apparently, a SAT-solver could have replaced the backtracking algorithm. But the reasons for not using it are as follows,

1. The different heuristics that we have devised add to the efficiency of the methodology. The run-time of our algorithm and the coverage results obtained (in section 2.6) show that the heuristics are appropriate for the automatic test generation from the ADL-description of the embedded processors.

2. The mapping of the behavioral constructs of an ADL in a SAT-solver is also a non-trivial task.

In the backtracking algorithm, since we are targeting the conditional block coverage, the nodes with label `MUX` are targeted. Here the term *pathway* is defined. A *pathway* (Π), is a sequence of vertices (v_0, v_1, ..., v_{n-1}), where v_0 and v_{n-1} are the start-node and last-node respectively ($v_i \in V(G_{DFG})$, $\forall i$).

$$\Pi : v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i \rightarrow \ldots \rightarrow v_{n-1}$$

A pathway with respect to a particular vertex (v_i) is Π_{v_i}, i.e. Π_{v_i}: $v_0 \rightarrow \ldots \rightarrow v_i$. Moreover, the labels of start-node and last-node are as follows.

$$label(v_0) = \texttt{MUX;}\ \ label(v_{n-1}) = \{\texttt{DECODE, CONST}\}$$

The algorithm has three interweaved parts in it. The intuitive idea is explained referring to figure 2.6.

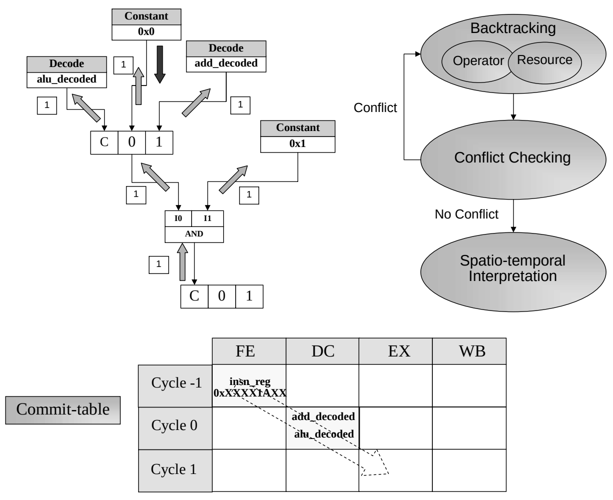


Figure 2.6: The Overall Algorithm

The Intuitive Idea

1. Backtracking: In the first part we backtrack in the G_{DFG}. Referring to figure 2.6, we start the algorithm by initializing the bottom-most `MUX` node with its control value '1'. This '1' in the `MUX` control demands '1' from the `AND`-labeled vertex. Here we define the term *demand-value* (say $demand(v_i)$) and *node-value* (say $nodeval(v_i)$). Demand-value is the set of expected values for a node to satisfy the current pathway upto vertex v_i (Π_{v_i}), and node-value denotes the temporary value which is set in a given node for the path-satisfaction. Formally,

$$demand(v_i) = \{d_{v_i} : d_{v_i} \models nodeval(\Pi_{v_{i-1}})\}$$

where, $d_{v_i} \models nodeval(\Pi_{v_{i-1}})$ represents that d_{v_i} is consistent with the node-values assigned to the sequence of vertices in $\Pi_{v_{i-1}}$. In the fig 2.6, *demand*(`AND`-labeled vertex) $= 1$ (since '1' in the output of `AND`-gate is necessary for the control logic value of the `MUX` being '1'). This node temporarily assumes '1' in its output(hence, $nodeval = 1$) and in its turn propagated '1' to both the operands of this node. In this way, the required value is back-propagated upwards until some node with label `DECODE` or `CONST` is reached e.g. here the back-propagation stopped after reaching the following nodes namely, *alu_decoded*(`DECODE`), *add_decoded*(`DECODE`) and the `CONST`-labeled nodes.

2. Conflict Checking: Back-propagation may not be smooth throughout. For example in figure 2.6, the demand-value to the `CONST` node (with constant value '0') is '1', which is not feasible. The conflicts can be of two types.

1. *Node-value conflict:* This conflict occurs at a particular vertex, when either of the following is true,

   - $nodeval(v_i) \notin demand(v_i)$. This happens when the node-value is already assigned by some other path Π'_{v_i}.
   - There exists no d_{v_i} ($d_{v_i} \in demand(v_i)$), such that d_{v_i} is a feasible node-value in vertex v_i. This is the situation in CONST node of the fig 2.6.

   In case of conflict in the current pathway, other possible options are explored by backtracking.

2. *Mutual exclusion conflict:* Consider the LISA Operation DAG in figure 2.1. There, the two operations ADD and SUB belong to the same LISA *group*, signifying that those are mutually exclusive. Formally,

   - any two operations which have same parent in the activation graph are mutually exclusive. i.e. $(parent(op_i) == parent(op_j)) \Rightarrow mutex(op_i, op_j)$
   - any two operations whose parents are mutually exclusive are also mutually exclusive.i.e. $mutex(parent(op_i), parent(op_j)) \Rightarrow mutex(op_i, op_j)$

During backtracking, it must be ensured that no two such exclusive operations are required to be executed at the same cycle.

3. Spatio-temporal Interpretation: Once there is no conflict in the pathway, i.e. the control value of a particular mux can be achieved through application of some constraints or by setting certain resource biases in proper sequence.This sequence is determined by the pathway denoted as '*conflictless-pathway*'. The '*conflictless-pathway*' is a set of DECODE and CONST vertices in some particular stage and cycle. We maintain a dedicated data-structure - '*commit-table*'(in figure 2.6) for this interpretation. This commit-table supplies the required constraints for the TGE in the proper sequence (figure 2.9). The commit-table consists of rows (*cycles*) and columns (*stages*). Each cell in the commit-table maintains a list of DECODE, CONST nodes and instruction-register biases. The DECODE nodes are converted to constraints for generating appropriate LISA operations and the node-value of CONST node is assumed to be instruction-register bias.

In our example, the commit-table suggests the following constraint *alu_decoded*, *add_decoded* (stage: DC, cycle: 0) and an instruction register bias (stage: FE, cycle: -1). The complete set represents one single instruction, as indicated by the diagonal arrow in the commit-table (in figure 2.6).

Some Special Cases

In order to pace up the back-propagation algorithm, some heuristics are employed. In the following subsubsections we have described some of those heuristics through examples.

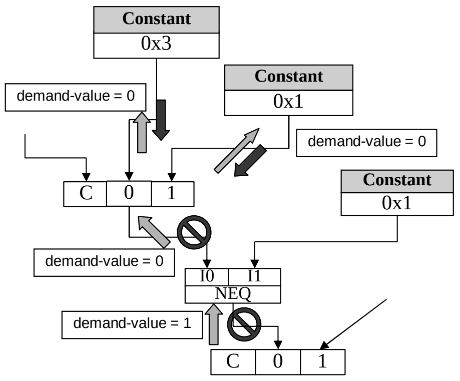


Figure 2.7: The block-path heuristic

Block Path Heuristic: While executing the algorithm described in section 2.3.2, it is important to minimize the back-tracking. Sometimes, it becomes evident from the scenario that no further back-tracking can lead to a feasible $nodeval(v_i)$ which satisfies Π_{v_i}. A typical scenario is shown in the following example. In figure 2.7 it can be observed that the demand-value ('0b1') propagated from the NEQ node to the MUX cannot be achieved since both the operands of the MUX is of type CONST having other values ('0b3' and '0b2'). Unless, we block the current pathway, the NEQ-node will continue sending other infeasible demand-values. So the path is blocked (or 'flagged') so that no more unnecessary iteration takes place in this pathway. Formally, the condition for blocking a path at node v_i is following,

$$(label(v_i) = \texttt{CONST}) \wedge (nodeval(v_i) \text{ does not satisfy } \Pi_{v_i})$$

Constant Lookahead Heuristic: Sometimes, it becomes totally unnecessary to iterate over a pathway with all the possibilities. In the G_{DFG}, the ALU operator nodes often have constant nodes as one of the operands. In this heuristic, the constant is *lookahead*-ed in order to determine a suitable demand-value. For example, a NEQ operator has the demand-value '0b0' and it has one constant operand with value '0b0'. Hence, it is clear that '0b0' needs to be sent in the pathway of the other operand.

2.4 Assertion-based Verification in LISA

In this section, we describe how the LISA-based test generation can be used in assertion-based verification. In the following subsection, the abstract fault models are first described followed by the methodology for automatic generation of the assertions in the RTL.

Assertions: Assertions are logical, sentential forms which are useful in expressing design properties explicitly. For elaboration, an assertion can be the following. At any instant of time, a unique address of the memory, can be written by only one hardware component. The major benefit of using such assertions is the formal rigor embedded in it. Assertions can be broadly classified into two groups, namely, *static* and *temporal* assertions. *Static* assertions do not include time, while, *temporal* assertions are useful in expressing properties which depend on time, i.e. the clock. In our methodology, we have used both static, as well as, temporal assertions, to detect the following abstract fault models.

2.4.1 Abstract Fault Models

In this part, we discuss the various fault models which are covered by our methodology. These abstract fault models are the potential faults likely to be discovered in an ASIP. *Similar fault models have been used to derive test-patterns in* [3]. The contribution of these fault models in this thesis is that we derive the assertions and not the test programs. Therefore, our approach is complementary to the test-pattern generation approach developed in [3].

Structural Fault Models: Structural fault models are those which are embedded in the micro-architecture. These faults are not always easy to be trapped at the ADL level. Usually, these bugs are detected during the RTL simulation phase since, the control flow of the ASIP needs to be monitored. From this perspective, we present the following fault models. Detailed discussion regarding the fault models can be found in [27].

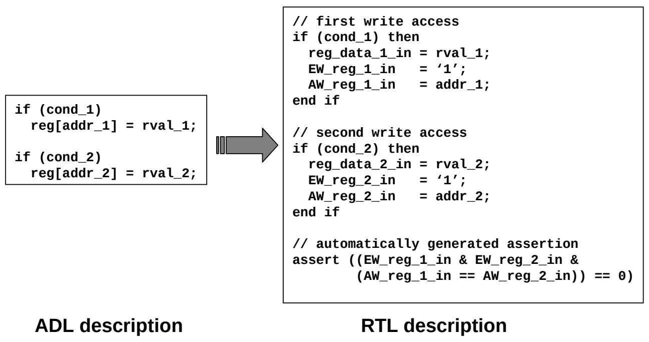
1. Fault Model for Register Transfer Operations
2. Fault Model for Register Write Operations
3. Fault Model for Pipeline Control Operations
4. Fault Model for Single Branch Execution

5. Fault Model for Memory Accessing

Behavioral Fault Models: Behavioral fault models are those, which are usually difficult to track down in the micro-architecture because of the involved data-flow analysis required. For elaboration, there can be a particular signal set to high resulting in some other signal to be set high for next 5 clock-cycles, otherwise there is a fault. Intuitively, trapping of such design-violations requires involved semantic behavior analysis of each instruction in the ADL level, as they are not generic in nature. We provide the designer with the required platform,(i.e. the assertion library) with the help of which one can manually insert assertions in the generated HDL.

2.4.2 Automatic Generation of Assertions

In order to generate the assertions, we resort to static ADL analysis techniques [29]. By static analysis of a LISA model, we derive a **global conflict graph**, where a conflict edge between two LISA operations indicates that those are not mutually exclusive.



```
if (cond_1)
  reg[addr_1] = rval_1;

if (cond_2)
  reg[addr_2] = rval_2;
```

```
// first write access
if (cond_1) then
  reg_data_1_in = rval_1;
  EW_reg_1_in   = ‘1’;
  AW_reg_1_in   = addr_1;
end if

// second write access
if (cond_2) then
  reg_data_2_in = rval_2;
  EW_reg_2_in   = ‘1’;
  AW_reg_2_in   = addr_2;
end if

// automatically generated assertion
assert ((EW_reg_1_in & EW_reg_2_in &
        (AW_reg_1_in == AW_reg_2_in)) == 0)
```

Figure 2.8: Automatic Assertion Generation from ADL

One processor instruction is distributed in the LISA description over a chain of LISA operations. For a single-threaded processor, it is not possible to execute more than one instruction inside one pipeline stage. The afore-mentioned *fault model for single branch execution* is covered by a structural assertion, which checks if two or more exclusive LISA operations are activated at the same instance. The *fault model for memory access* is covered by another assertion, which runs through all possible combinations of memory access from the processor and simply asserts if more than one `enable_write_memory` signal is high at the same instance alongwith a conflicting memory address location. The *Fault Model for Register Transfer Operations* is detected by structural assertions similar in nature. In order to detect whether more than one signal is trying to write simultaneously to a given register, the dynamic exclusiveness of the available enable-write signals is checked.

One example of automatic assertion generation is shown in figure 2.8. Here, the the ADL description of register write access and corresponding RTL description is written in pseudo-code. The prefix *EW* indicates enable write signal, whereas *AW* stands for address of the write access. For a non-faulty behavior, the two register-write operations cannot be activated at the same cycle given, the addresses are equal. Understandably, such errors are difficult to track from static analysis. An automatically generated assertion eases the verification by checking all such combinations across the complete processor.

2.5 Integrated Verification Flow

After the discussion on LISA-based test-generation and assertion generation, now we are in a position to describe the integrated verification flow. With reference to fig 2.9, we now outline the major components of the flow.

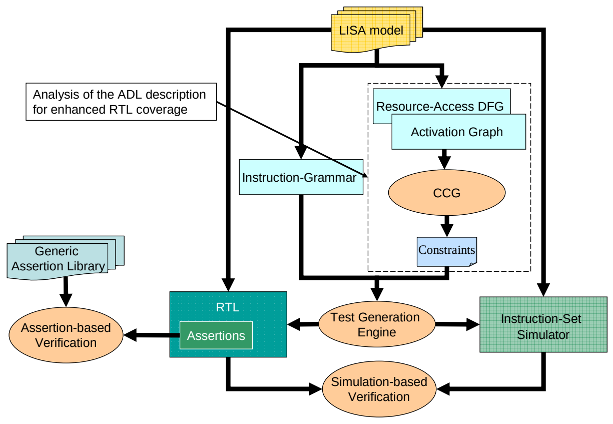

LISA model
Analysis of the ADL description for enhanced RTL coverage
Resource-Access DFG
Activation Graph
Instruction-Grammar
CCG
Constraints
Generic Assertion Library
Assertion-based Verification
RTL
Assertions
Test Generation Engine
Instruction-Set Simulator
Simulation-based Verification

Figure 2.9: The Integrated Verification Flow

- **ADL model:** In our proposed verification scheme the primary input is the ADL model. The entire tool-suite (e.g. assembler, dis-assembler, linker, instruction-set simulator, HLL-compiler etc), abstract processor description (e.g. resource-access data flow graph, instruction-grammar, activation graph etc) and the RTL (with embedded assertions) are automatically generated.

- **CCG:** The abstract processor description is next fed to the CCG to generate the constraints (refer section 2.3.2). This step is optional as CCG can be avoided and directly TGE can be used by the verification engineer.

- **TGE:** The instruction-grammar and constraints (if CCG preceds TGE in execution order) are the inputs to the TGE for generation of binary test-patterns (refer section 2.3.1). These test-patterns are utilized in the *dual verification modes* described next.

- **Assertion-Based Verification:** The automatically generated RTL consists of embedded assertions, which are generated assuming certain fault-models in the processor (described in detail in [27]). The simulation of RTL, with the test-cases generated from the TGE, is monitored by the assertions. Any discrepancy or failure of the asserted property during the simulation is reported to the user. This gives rise to the assertion-based verification.

- **Simulation-Based Verification:** This is the last phase of our complete verification platform. In this phase, we separately run the automatically generated instruction-set simulator (cycle-accurate) and the RTL simulator for the same test-case. The processor states are compared for each cycle between two simulations. We consider the cycle-accurate instruction-set simulator as the golden reference. Hence, a mismatch of the states indicates a design implementation error. The complete simulation-based verification is guided by a shell script.

2.6 Case Study

The complete integrated verification approach, proposed in this paper, is tested on several processor architectures. To analyze the strengths and drawbacks of different parts of the verification flow, the case study is divided into three phases. For various phases, different processor architectures are used, to show the *applicability* of the methodology over a diverse range of processors. In the following sub-sections, the case studies of proposed integrated verification approach are elaborated. We show case studies for following cases:

1. *GUI-based Test Pattern Generation* - without prior generation of any constraint.

2. *Coverage Constraint Generation* - with prior analysis of the model for constraint generation.

Toward the end of this section, we also show the number of assertions generated automatically for the different processors and the different kinds of bugs those were trapped by our approach.

2.6.1 GUI-based Test Pattern Generation

In order to test the strength of user-driven GUI-based test pattern generation, we have chosen the following considerably complex processors.

LEON3: LEON3 [30] is a 7-stage pipelined processor, compliant with the SPARC V8 architecture. Due to its SPARC compatibility (which is a well-studied architecture) and availability of its full source code under the GNU GPL license, LEON3 finds high relevance in research and academic pursuits. For our case study, we implemented the integer pipeline of LEON3 processor with integrated multiplication and division unit using ADL LISA.

LT_DSP: Another in-house processor architecture, named LT_DSP, is used for this experiment, too. LT_DSP is a 3-stage pipelined processor with several special instructions for digital signal processing as well as support for features like zero-overhead loop, multiple addressing modes. This processor allows upto 3 parallel memory accesses.

The LISA description of the processor is highly modular, where the same LISA operation is called from multiple operations. This modular nature makes the test generation challenging, since the different biasing modes of the TGE have to be carefully selected for attaining a high coverage. A brief summary of the processor architectures are presented in the following table 2.4.

	LEON3	LT_DSP
Basis Architecture	32-bit RISC	32-bit RISC
Pipeline Stages	7	3
Lines of LISA Code	9800	5695
Lines of VHDL Code	65136	64602

Table 2.4: Benchmark Processors

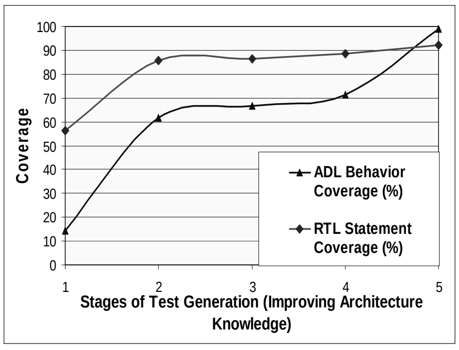
Coverage Results

The coverage results for the target architectures are presented in the following table 2.5. The RTL coverage metrics are obtained using Synopsys VCS tool flow [12]. We utilized the power of TGE for the generation of required test-benches, moving from pure random to constricted test-cases gradually. In case of LT_DSP, further stages of refinement were necessary since, the program memory allowed only 10K instructions to be loaded. However, an exhaustive run generates more than 170K instructions. Therefore, several levels of biasing were necessary to achieve the same coverage with much less number of instructions. For LEON3, 100% ADL coverage was not achievable since, we did not target to cover the LEON3 floating point unit. For LT_DSP, the reason for not achieving full coverage is the highly complex decoding structure. For example, a particular indirect addressing mode for memory alone was accessed from 36 different contexts. To cover all those possibilities with its underlying branching possibilities lead to 106K instructions, while generating via TGE. This would overflow the program memory space. In future, this can be attempted by repeated runs with reloading the program memory, while maintaining an incremental database of coverage.

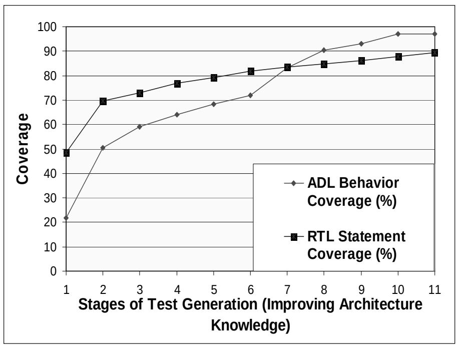
	ADL Behavior Coverage Coverage (%)	RTL Statement Coverage Coverage (%)	Number of Instructions	Number of Man-Days
LEON3	98.88	92%	532	1.5
LT_DSP	96.97%	89.3%	4689	2

Table 2.5: Coverage Results : GUI-based Test Generation

Coverage Analysis



(a) LEON3



(b) LT_DSP

Figure 2.10: ADL vs RTL Coverage

In the above figures, an analysis of the coverage results obtained by the GUI-based test generation scheme is presented. With increasing architecture knowledge, we found steeper rise in the ADL coverage, compared to that of RTL coverage (refer figure 2.10).

The ADL coverage represents the line coverage in the ADL *behavior* description and the RTL coverage represents the line coverage in the RTL description. The increasing design expertise is simply an abstract notation, where the higher value indicates greater usage of the architecture knowledge in test-generation.

The comparative result that we obtained regarding the ADL coverage vs RTL coverage, has demonstrated some features in it. It is interesting to note that the ADL coverage is initially much lower than the RTL coverage, whereas with high design knowledge the ADL coverage is higher than the RTL coverage. The reasoning behind this observation is the following. In an ADL, the hardware details are implicitly mentioned. For elaboration, an operation, has got the resource-usage declaration, coding, and the behavior, in the ADL level. The corresponding RTL forms are registers, decoders and data-path respectively. Hence, a given behavior gets expanded in RTL code. So, RTL coverage is initially bulkier, compared to ADL behavior coverage. With the increase in number of instructions, ADL coverage rises, while RTL coverage curve assumes a nearly flat, slow-rising gradient, as the major bulk is already covered. The two coverage curves converge at a point (e.g. in case of LEON3, when both attain 90% coverage). Beyond that point, ADL coverage curve steeply rises, as more and more corner cases get revealed.

2.6.2 Coverage Constraint Generation

The coverage constraint generation approach, presented in this paper, are tested with two different pipelined RISC processors.

LTRISC: The first one, LTRISC, is a 32-bit 4-stage pipelined RISC processor with basic support for arithmetic, load-store and branch operations. LTRISC contains mechanism for automated detection of pipeline data-hazards. All the instructions of LTRISC are conditional instructions, preventing straight-forward high-coverage test-case generation.

ICORE: The ICORE [31] architecture is dedicated for Terrestrial Digital Video Broadcast (DVB-T) decoding. It is based on a pipelined Harvard architecture implementing a set of general purpose arithmetic instructions as well as specialized trigonometric operations. Other notable features in ICORE, presenting hindrance to a smooth test-pattern generation, includes zero-overhead-loop, extensive branch instructions, complex instructions involving nested data-flow.

A brief summary of the processors is shown in table 2.6.

	ICORE	LTRISC
Basis Architecture	21-bit RISC	32-bit RISC
Pipeline Stages	4	4
Lines of LISA Code	2200	1838
Lines of Verilog Code	25200	9501

Table 2.6: Benchmark Processors

The test program for both the architectures are automatically generated using the flow described in this paper. The run-time of the backtracking algorithm is observed to be extremely low, due to the heuristics applied in the process. For both the architectures, the complete test program is generated in less than a second on a AMD Athlon XP 2600+ Processor (1916 MHz, 512 MB RAM) running SuSE Linux 9.2 operating system. The total number of instructions generated for ICORE is 727, whereas for LTRISC, 132 instructions are generated. In order to avoid unnecessary large loops or jumping outside program location, the regular features from TGE e.g. branch address biasing are extensively used during coverage constraint generation. Actually, it gave the complete test-automation tool an immense advantage to have a constrained random test pattern generator at the back-end.

Coverage Results

The test programs are then fed to the HDL descriptions and the coverage metrics are measured using Synopsys VCS tool flow [12]. The overall results of the block coverage and statement coverage are presented in the following table 2.7.

	ICORE	LTRISC
RTL Block Coverage	96.87%	97.71%
RTL Statement Coverage	98.42%	98.52%

Table 2.7: Coverage Results : CCG-based Test Generation

By checking the individual sections of the two processors, it turned out that the instruction behaviors are completely covered delivering 100% ADL statement coverage. Actually, those are the conditional blocks explicitly targeted. However, the register file contained several general purpose registers, which remain uncovered. For example, the following piece of assembly code is generated to cover pipeline data-hazard in LTRISC.

```
r0 = 1600
```

$$r0 = (r2 \leq 1685\ )$$
$$r0 = (r0 \leq 1529\ )$$

Here, register `r0` is used to model the data-hazard in comparison instructions of LTRISC. Similarly, the coverage of conditional blocks presented the test generator with varied choices of general purpose registers and one is randomly picked up. This assured full coverage of the instruction behaviors but, the coverage of the register files is still at the best constrained random. This resulted in less than 100% block coverage for LTRISC. However, the register file is regular in their structure and their coverage can be achieved by writing generic test-pattern generation subroutines. For ICORE, the overall block coverage and statement coverage is comparable to LTRISC. However, ICORE contained few uncovered conditional blocks in the instruction behavior. Closer inspection revealed that those blocks are conditioned by I/O pins. Obviously those could not be controlled by the instruction grammar-based TGE.

It is interesting to compare the advantage of the automated test pattern generator with the manual constrained random test pattern generation approaches. By allowing an experienced processor designer to use the ADL-driven constrained random TGE (the back-end of the presented tool-flow), it took 2 days to achieve similar coverage results. A completely manual creation of the test pattern for achieving full instruction coverage, without the deep knowledge of the target processor, will certainly take much longer.

Current Limitation

Currently, the backtracking algorithm is not capable of predicting suitable operands for arithmetic operations. The arithmetic operations like multiplication, addition generates a huge number of input possibilities for a given output. These possibilities need to be restricted for obtaining the test patterns within a reasonable time. This limitation prevented us from testing the CCG for more complex processors.

2.6.3 Assertion-based Verification

For all the above processors, OVA assertions were generated automatically from the ADL description. The assertions acted as checks to prevent several classes of faults. In case of those faults existing, a simulation run with automatically generated test-cases revealed it. The number of generated assertions for these processors are given in the following table 2.8.

	OVA Assertions
LEON3	7754
LT_DSP	16213
ICORE	5332
LTRISC	142

Table 2.8: Number of OVA Assertions

2.6.4 Detection of Bugs

The automatic test generation coupled with assertions helped us to uncover several bugs in the designs. Generally, we have detected design errors of two kinds. Firstly, there are *resident errors*, which were unknowingly introduced during the design-phase by the designer. Secondly, there are *injected errors*, which were intentionally introduced during the verification-phase.

Design Resident Errors

Same write-port of the memory was being accessed, simultaneously, by multiple units in LEON3. It got detected, as the assertion responsible for checking exclusiveness among the write-enable signals, got triggered.

The designer wanted to assign the following in the writeback stage of LEON3 processor, `Reg[reg_num] = a; Reg[reg_num | 1] = b;` Unfortunately, in a corner case, `reg_num` was *odd*, and hence the same register got accessed twice in the same cycle. It got trapped by the same class of assertion as mentioned in the previous example. This kind of faults can only be found by *dynamic execution* of instructions.

For LEON3 processor, the execution of the `div` instruction was initially carried out in the EX stage of the pipeline, alongwith other arithmetic, logic instructions, where all of those were activating *write_regarith* operation of WB stage. The designer wanted to decrease the critical path of the processor, which was running along the `div` operation. The execution of `div` instruction was distributed over two pipeline stages - *EX* and *ME*. Due to unmindful copying, still then, both the parts were erroneously activating the *write_regarith* operation, and got itself trapped into the single branch execution fault model.

In LT_DSP processor, two instructions were designed with the same syntax. During test pattern generation, the patterns are generated in binary form, which got dis-assembled correctly. However, while running the ADL simulation, the assembling phase produced the same instruction twice. The second instruction remained uncovered. A closer inspection quickly revealed the bug.

Trapping of Injected Bugs

A temporary register was inserted inside the LEON3 model which was written by two instructions, from different pipeline stages, simultaneously. It was caught by the fault model for register transfer operations.

Two new instructions were inserted, which can be represented by the following table 2.9. Here `x[29]` stands for 29 consecutive *don't care* bits in the instruction-word. It is easy to appreciate that, `testA` and `testB` have *partial similar coding contribution* which made the decoder activate two execution branches simultaneously. This design violation was covered by our fault model for single branch execution.

test_insn	: testA ∥ testB
testA	: 110 x[29]
testB	: 1 testC x[28]
testC	: 100

Table 2.9: Instructions with Coding-Overlap

2.7 Conclusion and Future Work

With the growth in demand of complex, optimized, fault-critical systems, ADLs integrated with verification support will become inevitable for the designers. Traditional verification approaches, performed using test automation tools or assertions begin at RTL level and therefore key design points might be missed due to the complexity of the system. In this work, we presented a verification methodology, in order to assist the ADL users with a completely self-sufficient, consistent design-and-test environment. The novelty of this approach lies in having stronger control over the complete verification flow, without doing away with the comfort of high abstraction level. Moreover, experiments showed that the integration of TGE and assertion-based verification approach would not burden the tool-chain and micro-architecture generation process, because of its manageable light kernel.

In future, we will like to focus on extending the backtracking algorithm for complex arithmetic operations. Furthermore, the automated interaction between coverage results and test generation framework to guide the test generation process will be an interesting future work.

Chapter 3

Formal Design Intent Verification

3.1 Introduction

Capacity limitations continue to impede the widespread adoption of *formal property verification* (FPV) in industrial design verification flows. In view of the theoretical lower bounds of the problem, it is unlikely that engineering advances in model checking tools will be able to scale the technology to a large extent in near future. Practitioners of FPV therefore advocate (a) the development of a well structured *formal verification plan* that attempts to verify different blocks of the design under well formed constraints (assumptions) about the rest of the design, and (b) using the results of these local verification efforts to reason about the global behavior of the whole design. The overall approach follows the *divide-and-conquer* paradigm, which is possibly the most traditional weapon in the arsenal of the human engineer for tackling complex problems.

The task of breaking down the problem of verifying a formal property over a design into the tasks of verifying local properties over the blocks of the design is very hard to automate since it requires considerable domain knowledge about the design. On the other hand, when design architects develop the functional specs of the blocks, they do expect the blocks to work together in a way that satisfies the architectural properties of the whole design. Today, formal tools do not provide a way to check whether the functional specs of the blocks cover the design's architectural intent, and to debug the coverage gap. The *design intent coverage* paradigm attempts to solve this problem through a new style of reasoning over formal specifications.

Our previous work on the design intent coverage paradigm can be found in [32]. In this approach, the design architect's knowledge about the partitioning of the design intent into the functional specifications of the blocks is exploited to develop a formal property suite $\mathcal{F}_i$ for each block, $\mathcal{M}_i$. This gives us two levels of specifications:

1. Specification of architectural properties over the whole design. We shall refer to this as the level-1 specs, $\mathcal{A}$, which represents the design's architectural intent.

2. The collection $\mathcal{R}$, of the formal property suites $\mathcal{F}_i$ of the blocks $\mathcal{M}_i$. We shall refer to this as the level-2 specs, $\mathcal{R}$ which represents the set of functional specifications of the component blocks in the architecture.

The design intent coverage analysis has two tasks, namely:

[**Primary coverage analysis:**] For each property, $\mathcal{P} \in \mathcal{A}$, we check whether $\mathcal{R} \Rightarrow \mathcal{P}$, that is, whether property $\mathcal{P}$ is *covered* by the level-2 specs.

[**Coverage gap presentation:**] If some property $\mathcal{P}$ is found to be not covered by the primary coverage analysis, then we refine $\mathcal{P}$ to a weaker property $\mathcal{P}'$ which captures those behaviors satisfying $\mathcal{P}$ that were not covered by $\mathcal{R}$. In other words, we attempt to find the weakest refinement $\mathcal{P}'$ of $\mathcal{P}$ such that $\mathcal{P}' \bigcup(\mathcal{R}) \Rightarrow \mathcal{P}$.

The refinement algorithm used in the second task produces a *structure preserving refinement*, that is the properties $\mathcal{P}$ and $\mathcal{P}'$ are structurally (and hence, visually) very similar. Visual similarity between the original and refined properties is needed to provide a legible feedback to the verification engineer about the gap between the level-1 and level-2 specifications. Our tool, SpecMatcher [33], uses several heuristics for working on the syntax tree of $\mathcal{P}$ to derive a visually close $\mathcal{P}'$ based on the coverage gap analysis. Some of these heuristics can be found in [32].

Our original version of SpecMatcher accepts only formal property specifications (specifically, Linear Temporal Logic (LTL) specifications), that is, both the level-1 and level-2 specifications are sets of formal properties. The coverage analysis is mainly done through LTL satisfiability checks.

In this chapter we extend the design intent coverage paradigm by supporting state machines as part of the level-1 and/or level-2 specifications. Specifically we support two types of hybrid specifications consisting of properties and state machines:

1. *Auxiliary State Machines.* Auxiliary state machines are abstract state machines explicitly defined by the verification engineer over the macro states of the protocol between the design under test (DUT) and its environment. For example, in a Bus protocol, the macro states could be Bus acquisition, Transfer, and Bus release. Since many properties are applicable in only specific macro states (such as properties describing valid ways to release the Bus), the use of auxiliary state machines in the specification helps the verification engineer to code the formal properties is a succinct and readable way. More details on the the use of auxiliary state machines in developing formal specifications can be found in [34, 1].

2. *Level-2 State Machines.* Often formal properties are not available for some of the components of the level-2 specification. Keeping in mind that the computational advantage of design intent coverage lies in restricting the analysis to the property domain (as opposed to model checking, where the level-2 consists of RTL blocks only), we show that small RTL components can be used in the level-2 specification by exporting them into the property domain. This also helps us in including the *glue logic* that is typically used to interconnect the blocks in the design.

A preliminary version of the second extension was presented in the DATE-2006 conference [35]. There are two fundamental contributions in this chapter, that enables the new design intent coverage analysis with state machines. These are:

1. We study the relation between satisfiability (which is primarily used in design intent coverage) and model checking, and show that in the context of our approach they are fundamentally similar. This helps us to develop a proof mechanism for answering the coverage question over hybrid specifications consisting of state machines and formal properties.

2. We present new algorithms for specification refinement that can handle auxiliary state machine specifications.

The chapter is organized as follows. The following section explores known results on the relation between LTL satisfiability and model checking. Section 3.3 describes the motivation, formulation and heuristics for design intent verification in presence of auxiliary state-machine. In section 3.5 we present results showing the runtimes of the proposed algorithms.

3.2 LTL: Satisfiability versus Model Checking

In this chapter we will develop the theory using *Linear Temporal Logic* (LTL) as the language for specifying formal properties. Since existing language standards such as PSL and SVA are based on LTL, no fundamental shift in our formal methods is expected when we generalize these to PSL/SVA. For sake of completeness, we present a very brief account on the syntax and semantics Linear Temporal Logic in the Appendix.

Example 1 *Consider a two bit Gray counter. A gray encoder has the following property: The next value of the counter differs from the present value of the counter in exactly one bit. Let us represent the two bits as x_1 and x_2. We consider a high active input* `reset`*, which resets the counter in the following cycle, i.e.*

$$G(\ \texttt{reset} \Rightarrow\ X(\neg x_1 \neg x_2))$$

Suppose, we wish to verify the following property: if the counter is not reset, then the next value of the counter differs from the present value by exactly one bit. Formally, in LTL this property may be expressed as follows:

$$\varphi: \qquad G(\neg\texttt{reset} \Rightarrow\ (x_1 \oplus Xx_1)\ \oplus\ (x_2 \oplus Xx_2))$$

```
module gray_counter (reset, x1, x2, clk);
input reset, clk;
output x1, x2;

always@(posedge clk)
begin
  if(!reset) begin
    x1 <= x2;
    x2 <= ~x1;
  end
  else begin
    x1 <= 1'b0;
    x2 <= 1'b0;
  end
end

endmodule;
```

Figure 3.1: Verilog implementation of a 2-bit gray counter

Fig 3.1 shows an implementation of the counter (in verilog). We wish to verify whether this implementation statisfies φ. The traditional approach for LTL model checking checks whether any accepting run for $\neg\varphi$ belongs to the implementation. This check is done by extracting a state machine $\mathcal{T}$ representing the implementation and verifying whether the product of $\mathcal{T}$ with any acceptor of $\neg\varphi$ is empty.

$\mathcal{T}$ corresponding to our Gray counter implementation (fig 3.1) can be expressed in LTL as follows,

$$P_1 \ : \ G\,((\neg\texttt{reset} \ \wedge \ x_2) \ \Leftrightarrow \ X(x_1)) \tag{3.1}$$

$$P_2 \ : \ G\,((\neg\texttt{reset} \ \wedge \ \neg x_1) \ \Leftrightarrow \ X(x_2)) \tag{3.2}$$

There can be two approaches of verifying the property φ.

1. Model-check the property φ on the model extracted from the 2-bit gray counter module.

2. The other approach is to check the LTL-satisfiability of the following LTL property.

$$[S_0 \ \wedge \ (P_1 \wedge P_2)] \Rightarrow \varphi$$

The above approach is fundamentally equivalent to checking whether the logical representation of $\mathcal{T}$ implies φ, that is, we check whether $\mathcal{T} \Rightarrow \varphi$ is valid. This follows from the fact that $\mathcal{T} \Rightarrow \varphi$ is valid iff $\mathcal{T} \wedge \neg\varphi$ is unsatisfiable. The above example indicates that for LTL, the model checking problem and the satisfiability problem are fundamentally similar. Not surprisingly, the computational complexity of both problems are the same (both are PSPACE-complete). Therefore, theoretically we can always transform the LTL model checking problem into a LTL satisfiability problem. We do not take explicitly take this approach in practice, because verification tools perform a lot of reductions (pruning) on the state machine before the actual model checking step, and these reductions are easier to apply on state machines rather than on formal properties.

Let us now compare the model checking problem with the design intent coverage problem. In both problems we essentially check whether a level-2 specs logically implies the level-1 specs. Design intent coverage represents one extreme of the spectrum, where the level-2 specs consists only of formal properties over the component modules. Model checking represents the other end of the spectrum, where the level-2 specs consists only of state machines (RTL blocks). In both problems, the level-1 specs consists of a set of formal properties. State machines typically contain a lot more details than needed to prove a level-1 property – replacing the state machines with a set of properties that capture only the relevant behaviors gives the computational advantage that the design intent coverage approach enjoys over model checking.

In this chapter, we use the transformation of state machines into LTL properties to unify the problems of design intent coverage and model checking, and solve the unified problem using LTL satisfiability. Moreover we define new algorithms for refining the specification under such transformations to reflect the coverage gap between the level-1 and level-2 specifications in a readable way.

3.3 Design Intent Verification with State-machine Based Properties

In this section we elaborate our contribution in this new design intent verification problem for auxiliary state-machine based properties. Before we embark upon the formal discussion, consider the following motivating example. In section 3.4 we discuss about the possible generic extensions of the problem.

3.3.1 An Example of Auxiliary State-machine based Properties

Consider a Bus protocol (MyBus protocol) that supports multiple master devices, multiple memory-mapped slave devices, and a single arbiter. The Bus has 64-bit multiplexed address and data lines. During a transfer, address and data are time multiplexed on these 64 lines, with address and data appearing in alternately. We focus on the behavior of a specific master device, $\mathcal{M}$, which is the *highest priority* device on the Bus. The interface of $\mathcal{M}$ is shown in Fig 3.2.

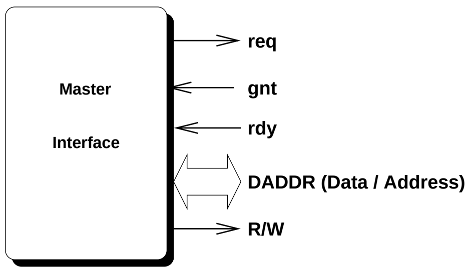


Figure 3.2: Master Interface of MyBus Protocol Source: [1]

The master interface is IDLE when $\mathcal{M}$ does not intend to perform a transfer. When the master intends to start a transfer, it raises its request line, `req`. The arbiter provides the grant (by asserting the `gnt` signal) in the immediate next cycle as $\mathcal{M}$ has the *highest priority.*

On receiving the `gnt`, $\mathcal{M}$ In case the region is available, $\mathcal{M}$ floats the address in the

Bus and waits (from the next cycle onwards) for the `rdy` signal from the slave device. We refer this phase of the transfer as the ADDRESS cycle.

The `rdy` signal from the slave indicates that the slave is ready for the transfer. On receiving this signal the master enters the DATA cycle and does the following:

1. In the case of a write, it floats the data on the Bus.

2. In the case of a read, it expects the slave to produce the data on the Bus.

The intent to read/write is indicated by a R/W signal – high indicates write, low indicates read. After each data cycle, the master may start another address cycle by floating the next address on the Bus. At any point of time the master can return to the IDLE state by lowering the `req` line, which signals the end of the transfer to the arbiter. A sample transfer is shown in Fig 3.3.

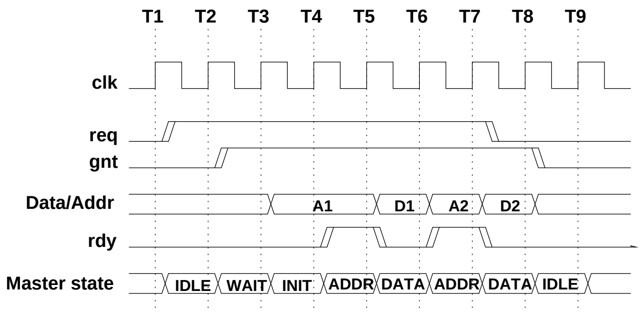


Figure 3.3: A sample transfer Source: [1]

Fig 3.4 show the abstract state machine for the MyBus master device The state-machine contains only sufficient information that carries it through the major phases of the protocol. For example, the status of the `gnt` line are not indicated in the DATA and ADDR cycles, though this is relevant to the first property. As there are five states, three bits are used to encode the states.

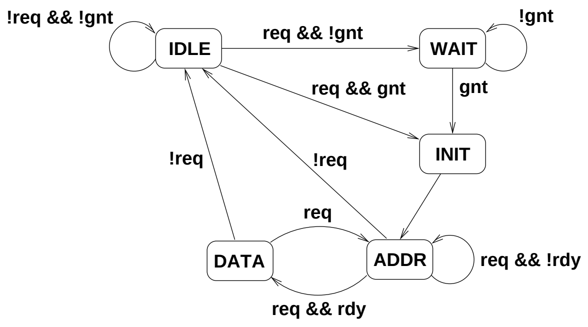


Figure 3.4: State Machine for the MyBus Master Device Source: [1]

The presented MyBus protocol will be the running example in our discussion. We will modify the state machine to show different aspects of the problem, as and when required. In the next section, we formulate the problem and describe the algorithms and heuristics to address the design intent verification problem.

3.3.2 Formal Characterization

The problem that we address in this section is the following. Given the architectural specs ($\mathcal{A}$) assuming an auxiliary architectural state-machine ($\mathcal{T}$) and the RTL specs of the modules ($\mathcal{R}$), verify whether there exists a gap between the specs $\mathcal{A}$ and $\mathcal{R}$. The formal characterization of the problem is presented first, followed by the proposed approach to solve it.

Auxiliary state-machine based design intent verification essentially checks whether RTL specs ($\mathcal{R}$) covers the architectural specs ($\mathcal{A}$), where $\mathcal{A}$ is based on a assumed state-machine $\mathcal{T}$. This state-machine $\mathcal{T}$ contains *states* and *transitions*. The *states* are encoded with a set ($\mathcal{AP}_\mathcal{T}$) of state-bit variables which is fictitious in the RTL level. In the motivating example presented in the previous subsection, `TRANSFER`, `ADDR`, `RETRY` and `NTRY` are the states. The transitions are made over the different input and output conditions.

The inputs to this problem are listed as follows.

1. The *architectural specification* $\mathcal{A}$ is a set of LTL properties over a set $\mathcal{AP}_\mathcal{A} \cup \mathcal{AP}_\mathcal{T}$ of boolean signals, where $\mathcal{AP}_\mathcal{A}$ and $\mathcal{AP}_\mathcal{T}$ are the sets of architectural signals and the auxiliary state-bits respectively.

2. The *RTL specification* $\mathcal{R}$ is a set of LTL properties over a set $\mathcal{AP}_\mathcal{R}$ of boolean signals, where $\mathcal{AP}_\mathcal{R}$ is the set of the RTL signals.

3. The state-machine model $\mathcal{T}$, of the architecture where the states are different valuations of $\mathcal{AP}_\mathcal{T}$ and the transitions are over $\mathcal{AP}_\mathcal{A}$.

Our assumptions are following.

Assumption 1 *Throughout this thesis we assume $\mathcal{AP}_\mathcal{A} \subseteq \mathcal{AP}_\mathcal{R}$ i.e. the set of architectural signals is a subset of the set of RTL signals.*

Typically this is not a restrictive assumption within the design hierarchy, since it is generally considered a good practice for designers at a lower level of the design hierarchy to inherit the interface signal names from the previous level of hierarchy.

Assumption 2 $\mathcal{AP}_{\mathcal{T}} \cap \mathcal{AP}_{\mathcal{R}} = \phi$ *i.e. the state-bits are fictitious and non-transparent to the RTL.*

Definition 1 **[Coverage Definition:]**
The RTL specification covers the architectural intent iff there exists no run that refutes one or more properties of the architectural intent (based on a fictitious state-machine model) but does not refute any property of the RTL specification. □

The current verification problem is almost similar as described in the first problem. It is as follows:

- To determine whether the RTL specification covers the architectural specification, given the state-machine model of the architecture
- If the answer to the previous question is no, then to determine a set of additional properties that represent the coverage gap (that is, these properties together with the RTL specification succeed in covering the architectural specification)

One of the possible approach might be following. A *state* in $\mathcal{T}$ represents the different valuations of the signals at a given time. So we can replace a given state, by the conjunction of different paths leading to that particular state from the start-state(s). However, the number of paths is usually large (and even infinite) rendering the idea infeasible. The following theorem shows us an alternate intuitive way to answer the first question.

Theorem 1 *The RTL specification, $\mathcal{R}$, covers the architectural specification $\mathcal{A}$, iff the temporal property $\mathcal{R} \wedge \neg(\mathcal{T} \Rightarrow \mathcal{A})$ is false.*
Proof: *The property $\mathcal{R} \wedge \neg(\mathcal{T} \Rightarrow \mathcal{A})$ represents the set of runs which refutes architectural intent but are passed by the RTL properties. If this property is false then these runs are not present in the complete RTL specification. Hence all runs passed by $\mathcal{R}$ are present in $\mathcal{T} \Rightarrow \mathcal{A}$ and thus RTL specification covers the architectural intent. On the other hand if $\mathcal{R} \wedge \neg(\mathcal{T} \Rightarrow \mathcal{A})$ is true, then there exists a run which is passed by the RTL spec but will be refuted by the architectural specs and hence the RTL does not cover the architectural intent.*□

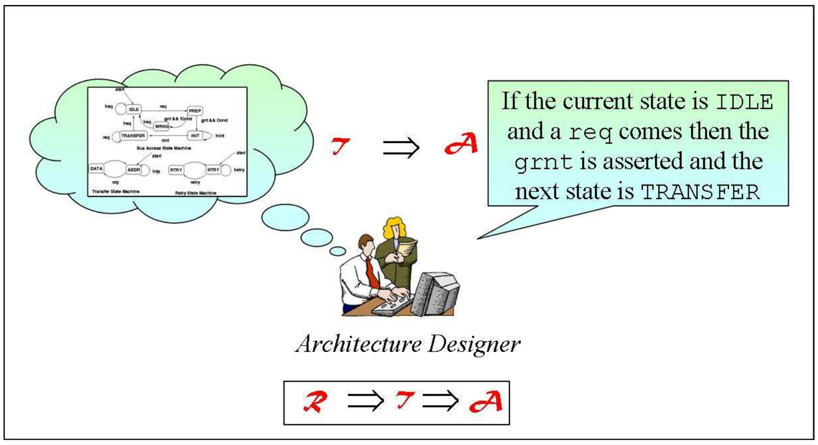

$\mathcal{T} \Rightarrow \mathcal{A}$
If the current state is IDLE and a req comes then the grnt is asserted and the next state is TRANSFER
Architecture Designer
$\mathcal{R} \Rightarrow \mathcal{T} \Rightarrow \mathcal{A}$

Figure 3.5: The architectural state-machine is an *assumption* made by the designer, regarding the desired behavior of the architecture, for the ease of expressing the *context-sensitive* properties.

Now this state-machine is completely fictitious and it is the intent of the architecture designer. The architect writes the architectural specifications assuming the underlying state-machine of the architecture. Fig 3.5 illustrates the idea.

The theorem 1 answers the primary coverage question by checking the validity of $\mathcal{R} \Rightarrow (\mathcal{T} \Rightarrow \mathcal{A})$. The next challenges are following (a) "*how do we answer the primary coverage question efficiently?*" and "*if gap found, how do we represent it?*. Observe that the major challenge in this case is the fact that the state-bits are non-transparent to the RTL specs, i.e. $\mathcal{AP}_{\mathcal{T}} \cap \mathcal{AP}_{\mathcal{R}} = \phi$, which inhibits the usage of our tool named Specmatcher. In the following derivation we circumvent this obstacle.

$$\mathcal{R} \Rightarrow (\mathcal{T} \Rightarrow \mathcal{A}) \tag{3.3}$$
$$\Rightarrow \mathcal{R} \Rightarrow (\neg\mathcal{T} \vee \mathcal{A}) \tag{3.4}$$
$$\Rightarrow \neg\mathcal{R} \vee (\neg\mathcal{T} \vee \mathcal{A}) \tag{3.5}$$
$$\Rightarrow (\neg\mathcal{R} \vee \neg\mathcal{T}) \vee \mathcal{A} \tag{3.6}$$
$$\Rightarrow \neg(\mathcal{R} \wedge \mathcal{T}) \vee \mathcal{A} \tag{3.7}$$
$$\Rightarrow (\mathcal{R} \wedge \mathcal{T}) \Rightarrow \mathcal{A} \tag{3.8}$$

Now the state-bits of the auxiliary state-machine is abstracted as the RTL signals. Intuitively, the physical significance of the equation 3.8 is easy to conceive. The state-machine is just an assumed model to represent the actual implementation. This model is invisible to the RTL specification. In the following expression,

$$(\mathcal{R} \wedge \mathcal{T}) \Rightarrow \mathcal{A}$$

we compose the RTL specification with the architectural state-machine and determine the validity. The idea is depicted in Fig 3.6.

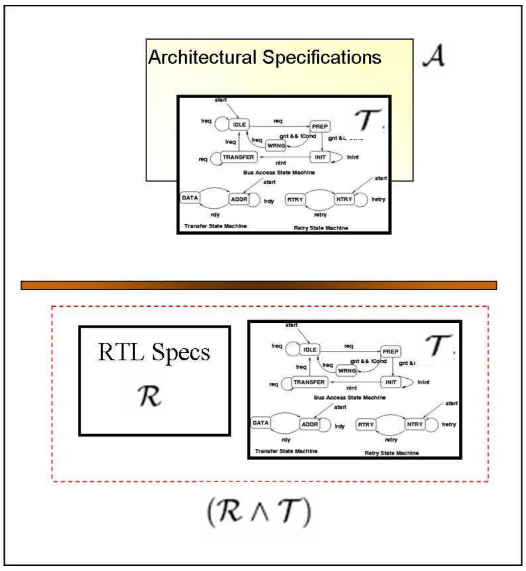


Figure 3.6: The architectural state-machine is originally *invisible* to the RTL. The physical significance of the derived expression $(\mathcal{R} \wedge \mathcal{T}) \Rightarrow \mathcal{A}$ is that the fictitious architectural state-machine becomes *visible* to the RTL, i.e. the architectural state-machine now becomes part of the RTL specs too.

The other remaining problem is more challenging. If equation 3.8 is not valid, how should we represent the gap elegantly? We will be using the variant of the same **Push_Term** algorithm [36]. Before outlining the algorithm, we define the following.

Definition 2 [Strong and weak properties:]
A property $\mathcal{F}_1$ is stronger than a property $\mathcal{F}_2$ iff $\mathcal{F}_1 \Rightarrow \mathcal{F}_2$ and $\mathcal{F}_2 \not\Rightarrow \mathcal{F}_1$. We also say that $\mathcal{F}_2$ is weaker than $\mathcal{F}_1$. $\square$

Definition 3 [Coverage Hole in RTL Spec:]
A coverage hole in the RTL specification is a property $\mathcal{R}_H$ over $\mathcal{AP}_{\mathcal{R}}$, such that $(\mathcal{R} \wedge \mathcal{R}_H) \wedge \neg\mathcal{A}$ is false *in $\mathcal{M}$, and there exists no property, $\mathcal{R}'_H$, over $\mathcal{AP}_{\mathcal{R}}$ such that $\mathcal{R}'_H$ is weaker than $\mathcal{R}_H$ and $(\mathcal{R} \wedge \mathcal{R}'_H) \wedge \mathcal{A}$ is also* false *in $\mathcal{M}$. In other words, we find the weakest*

property that suffices to close the coverage hole. Adding the weakest property strengthens the RTL specification in a minimal way. □

3.3.3 Coverage Algorithm

The core idea behind our algorithm is to present a structure preserving form of the coverage gap. Our algorithm takes each formula $\mathcal{F}_{\mathcal{A}}$ from the architectural intent $\mathcal{A}$ and finds the coverage gap, $\mathcal{G}$, for $\mathcal{F}_{\mathcal{A}}$, with respect to the RTL properties $\mathcal{R}$ in presence of the auxiliary architectural state-machine $\mathcal{T}$. Since $\mathcal{R}$ and $\mathcal{M}$ are required to cover every property in $\mathcal{A}$, we use this natural decomposition of the problem. The algorithm below implements this idea. Here we use $\mathcal{U}$ to represent the RTL coverage hole.

Algo. 3.3.1 Coverage Algorithm

```
Find_Coverage_Gap($\mathcal{F}_{\mathcal{A}}$,$\mathcal{R}$,$\mathcal{T}$)

  1. compute $\mathcal{U} = \mathcal{F}_{\mathcal{A}} \vee \neg(\mathcal{R} \wedge T)$
  2. If $\neg(\mathcal{U})$ is not false in $\mathcal{M}$ then
      (a) Unfold $\mathcal{U}$ to create a set of uncovered terms, $\mathcal{U}_M$, that approximates the coverage gap;
      (b) Use universal quantification to eliminate signals belonging to $\mathcal{AP}_{\mathcal{R}} - \mathcal{AP}_{\mathcal{A}}$,
      (c) Push the terms of $\mathcal{U}_M$ into $\mathcal{F}_{\mathcal{A}}$ to obtain $\mathcal{F}_{\mathcal{U}}$.
      (d) Weaken $\mathcal{F}_{\mathcal{U}}$ to obtain the final uncovered formula $\mathcal{G}$.
  3. Return $\mathcal{G}$;
```

The details of Algorithm 3.3.1 were presented in [35, 36]. It is interesting to note the step 2(c) (known as **Push_Term** algorithm presented in [36]) of the above algorithm. We briefly outline the essence of the algorithm used there. In this function, the terms of $\mathcal{U}_M$ are pushed into the syntactic structure of the property $\mathcal{F}_{\mathcal{A}}$. To intuitively explain the working of the algorithm, consider a case where $\mathcal{F}_{\mathcal{A}}$ is of the form $f \Rightarrow g$. Let $Var(f)$ and $Var(g)$ denote the denote the set of variables of f and g respectively. We compute the universal abstraction of $\mathcal{U}_M$ with respect to $Var(g)$ and recursively push the terms containing variables only from $Var(g)$ to g. We then compute the universal abstraction of $\mathcal{U}_M$ with respect to $Var(f) \cup \mathcal{EV}$ (where $\mathcal{EV}$ denote those variables which are present in $\mathcal{AP}_\mathcal{A}$ but absent in $\mathcal{F}_\mathcal{A}$) and recursively push the terms to f. The decision to push the terms containing entering variables to the left of the implication is a heuristic (but correct, since we could push them either way). Pushing the other way may expose another form of the uncovered architectural intent (and we may like to present both the forms).

In this particular extension of the design intent verification problem, if we directly use the **Push_Term** algorithm presented in [36], the result may not be comprehensible. We analyze those outcomes and apply a simple trick (which we call as the *state-bit renaming heuristic*) to do away with the problem, described in the next subsection.

3.3.4 The State-bit Renaming Heuristic

We obtained the following expression, $(\mathcal{R} \wedge \mathcal{T}) \Rightarrow \mathcal{A}$. We will show that a variant of the **Push_Term** algorithm in [32] will be able to give the coverage gap. We explore this further with the following toy example.

Example 2 *Let the architectural and RTL properties are given as follows,*

$$\mathcal{A}: \quad (r_1 \wedge \neg s) \Rightarrow X(g_1 \cup g_2) \wedge X(s) \tag{3.9}$$
$$\mathcal{R}: \quad (r_1 \wedge r_2) \Rightarrow X(g_1 \cup g_2) \tag{3.10}$$

where, r_1, r_2, g_1, $g_2 \in \mathcal{AP}_\mathcal{A}$ and s is the state-bit as shown in fig 3.7. The transitions in the architectural state-machine are shown in the same figure.

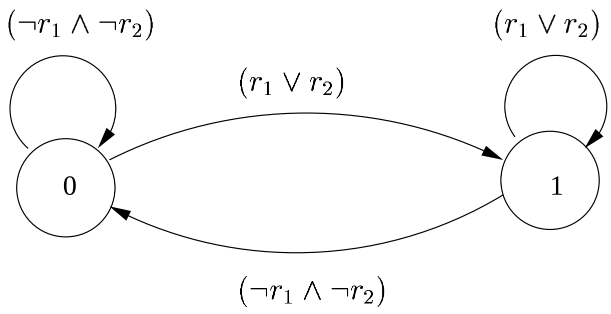


Figure 3.7: The architectural state-machine

As the number of states is only two, only one state-bit (s) is sufficient to represent them. Moreover, this state-machine is visible at the architecture level only. The state transitions are shown in following LTL properties.

$$T1: \quad G(\neg s \wedge (\neg r_1 \wedge \neg r_2) \Rightarrow \neg X(s)) \tag{3.11}$$

$$T2: \quad G(\neg s \wedge (r_1 \vee r_2) \Rightarrow X(s)) \tag{3.12}$$

$$T3: \quad G(s \wedge (r_1 \vee r_2) \Rightarrow X(s)) \tag{3.13}$$

$$T4: \quad G(s \wedge (\neg r_1 \wedge \neg r_2) \Rightarrow \neg X(s)) \tag{3.14}$$

Now, from the equation 3.8, we test the validity of the following equation.

$$\mathcal{R} \wedge (T1 \wedge T2 \wedge T3 \wedge T4) \Rightarrow \mathcal{A}$$

The architectural property is silent regarding the atomic proposition r_2, and hence it is easy to appreciate that there exists a gap between the specifications (since the RTL specifies nothing when $(r_1 \wedge \neg r_2)$ holds true). Applying left-insertion heuristic (as described in [32]) SpecMatcher [33] gives the following coverage gap,

$$\mathcal{F}_{\mathcal{U}}: \quad G((r_1 \wedge \neg r_2 \wedge \neg s \wedge X(s)) \Rightarrow X((\neg g_1 \wedge \neg s) \vee (g_1 \cup g_2))) \tag{3.15}$$

Clearly, this clumsy representation of the coverage gap is incomprehensible. Naturally, we raise the next question: *How to represent the gap more succinctly and meaningfully?* Before addressing this question, we analyze the steps in **Push_Term** algorithm to get deeper insight.

Analysis of Push_Term Algorithm: Without loss of generality, we assume left-insertion strategy throughout the discussion. The algorithm defines $Var(f)$ (the variables in formula f) and the *entering variables* $(\mathcal{EV})$, (variables belonging to $(\mathcal{AP}_{\mathcal{A}} - \mathcal{F}_{\mathcal{A}})$, where $\mathcal{F}_{\mathcal{A}}$ is the set of variables in architectural formula). In **case**$(\mathcal{F} \equiv (f \Rightarrow g))$, the variables either in $Var(f)$ or in $\mathcal{EV}$ enter the left-subtree of the formula (shown in fig 3.8).

This insight helps us in understanding the genesis of such complicated gap. The lhs of equation 3.15 contains both the terms s and $X(s)$, making the equation more complex, because the presence of the current and the future values of the state-bit on the same side of the implication bears no physical significance. In order to avoid these situations, we propose the following: *In all the specifications and the state-transitions, replace the X-guarded terms of the state-bits with new variable names.* For example, $\mathcal{A}$ in example 2 becomes,

$$\mathcal{A}': \quad (r_1 \wedge \neg s) \Rightarrow X(g_1 \vee g_2) \wedge next\_s \tag{3.16}$$

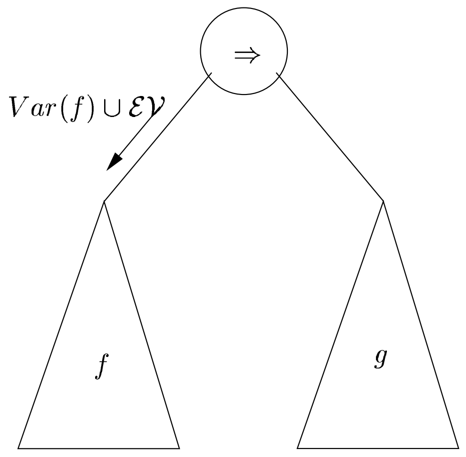


Figure 3.8: The entering variables enter the left subtree in **case:** $(\mathcal{F} \equiv (f \Rightarrow g))$

We replace $X(s)$ (in $\mathcal{A}$) with $next\_s$ (in $\mathcal{A}'$). Now, since $next\_s \notin (Var(f) \cup \mathcal{EV})$, it will not enter the same subtree as s does. Applying this change in SpecMatcher, we obtain the following gap which is more meaningful to the designer.

$$\mathcal{F}_{\mathcal{U}}' : \quad G(\neg s \wedge (r_1 \wedge \neg r_2) \Rightarrow X(g_1 \cup g_2)) \tag{3.17}$$

The problem we addressed in this section is a simple model since we assumed auxiliary state-machine only in the architectural level. But the state-machine may be present in the both the specifications. In the next section, we discuss some more generic cases of state-machine based expression of the properties, with illustrative examples. In the end we outline the unified solution toward solving state-machine based design intent verification problem in the presence of the concrete modules.

3.4 The Generic Extentions

The problem described in the previous section is a very simplistic model of the actual problem. But in order to appreciate the challenge of design intent verification problem, the bigger picture needs to be perceived. To give the flavor of the complexity, next we list the various models based on the the absence/presence of auxiliary state-machines at different levels. We start with the simplest LTL property based intent coverage problem [37] and gradually broaden the scope of the problem.[1]

[1]The cases listed are those cases where the *concrete modules* [35] are absent.

- **Case 1:** Both the architectural specs ($\mathcal{A}$) and the RTL specs ($\mathcal{R}$) are LTL properties, with $\mathcal{AP}_{\mathcal{A}} \subseteq \mathcal{AP}_{\mathcal{R}}$. We check the validity of

$$\mathcal{R} \Rightarrow \mathcal{A} \tag{3.18}$$

  and the gap (if present) is represented following the **Push_Term** algorithm presented in [37].

- **Case 2:** The architectural state-machine ($\mathcal{T}_{\mathcal{A}}$) alongwith the LTL architectural specs based on it ($\mathcal{A}$) is given. The RTL specs ($\mathcal{R}$) are also given in LTL. The architectural specs ($\mathcal{A}$) are on ($\mathcal{AP}_{\mathcal{A}} \cup \mathcal{AP}_{\mathcal{T}_{\mathcal{A}}}$). Moreover, ($\mathcal{AP}_{\mathcal{A}} \subseteq \mathcal{AP}_{\mathcal{R}}$) and ($\mathcal{AP}_{\mathcal{T}_{\mathcal{A}}} \cap \mathcal{AP}_{\mathcal{R}} = \phi$). In this case, we check the validity of,

$$\mathcal{R} \Rightarrow (\mathcal{T}_{\mathcal{A}} \Rightarrow \mathcal{A}) \tag{3.19}$$

  and the gap (if present) is represented following the *State-bit Renaming Heuristic* described in subsection 3.3.4 .

- **Case 3:** Next suppose, there is an underlying state-machine for the RTL also. Let it be $\mathcal{T}_{\mathcal{R}}$. The RTL specs are on ($\mathcal{AP}_{\mathcal{R}} \cup \mathcal{AP}_{\mathcal{T}_{\mathcal{R}}}$). Besides, ($\mathcal{AP}_{\mathcal{T}_{\mathcal{R}}} \cap \mathcal{AP}_{\mathcal{A}} = \phi$). In this case, we check the validity of,

$$(\mathcal{T}_{\mathcal{R}} \Rightarrow \mathcal{R}) \Rightarrow (\mathcal{T}_{\mathcal{A}} \Rightarrow \mathcal{A}) \tag{3.20}$$

- **Case 4:** We further assume that suppose there are n RTL modules. For each module, suppose there exists a state-machine ($\mathcal{T}_{R_i} \forall i = 1$ to n). Let the RTL specs be $\mathcal{R}_1, \ldots, \mathcal{R}_n$. In order to verify the design intent, we check the validity of,

$$\bigwedge_i (\mathcal{T}_{R_i} \Rightarrow \mathcal{R}_i) \Rightarrow (\mathcal{T}_{\mathcal{A}} \Rightarrow \mathcal{A}) \tag{3.21}$$

- **Case 5:** Lastly, suppose the architectural state-machine is *factored* in m state-machines (e.g. $\mathcal{T}_{A_j}, j = 1$ to m). In that case, we check the validity of the following,

$$\bigwedge_i (\mathcal{T}_{R_i} \Rightarrow \mathcal{R}_i) \Rightarrow ((\bigwedge_j \mathcal{T}_{A_j}) \Rightarrow \mathcal{A}) \tag{3.22}$$

  This is the most generalized validity check equation for any architecture which consists of no *concrete* modules.

3.4.1 Design Intent Verification in MyBus Protocol

In this subsection, based on the architectural state machine presented in MyBus protocol, we choose the following simple architectural and RTL properties.

Arch. spec ($\mathcal{A}$): *Each DATA cycle is of unit cycle duration.* The LTL representation based on the state machine of fig 3.4 is the following.

$$G((state = DATA) \Rightarrow \neg X(state = DATA))$$

RTL spec ($\mathcal{R}$): *The three signals (*`req`*,* `gnt`*,* `rdy`*) will not remain high for two consecutive cycles.* The LTL representation is the following.

$$G((req \wedge gnt \wedge rdy) \Rightarrow (X(\neg req) \vee X(\neg rdy) \vee X(\neg gnt)))$$

In order to answer the initial coverage question, we apply equation 3.19 and the SpecMatcher returns *complete coverage.* $\mathcal{T}_{\mathcal{A}}$ is the conjunction of the transitions in the finite state-machine of the architecture intent. Next we explore the case where multiple state-machines are present in the architectural intent - *factored architectural state-machine.*

Factored Architectural State Machine: We observe, that the main challenge in adopting the state-machine based property writing strategy is in creating the state machine model for the protocol - specifically in deciding the level of details that should go into the state machine model. The state machine should not be too detailed, then the coding of the state machine will grow unreadable and error prone. On the other hand, too abstract state machine leaves no room for simplification of the coding of the properties. We take a middle path.

Fig 3.9 shows a set of factored abstract state machines that explicitly model the contexts. The choice of these factored state machines depends on the user. The first state machine explicitly captures the different contexts of the Bus access in terms of the `req` and `gnt` signals. The second state machine is triggered by the outedge from the INIT state of the first, describes a more detailed state of the master during the transfer.

We encode the first state machine in terms of a 2-bit variable, *fsm1_state*, and the second one in terms of a 1-bit variable, *fsm2_state.* The context of each of the properties is encoded in terms of the states of the factored state machines and other interface signals. Next, we define the properties that we want to consider while design intent verification.

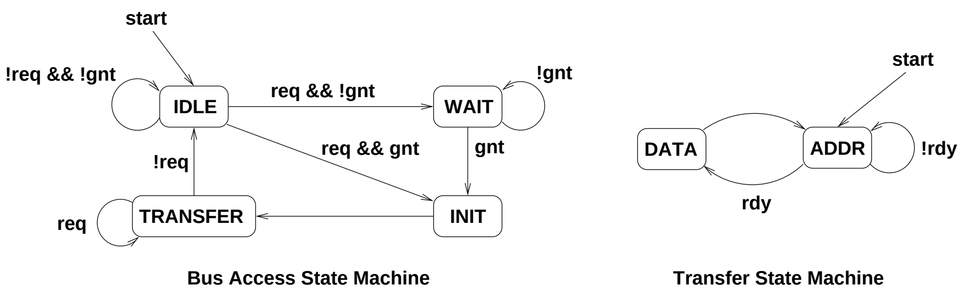


Figure 3.9: Factored State Machines of MyBus Protocol Master

Arch. spec ($\mathcal{A}$): *With the assertion of* `req` *and* `rdy` *signals in the ADDR state, the Bus access state remains in the TRANSFER state while the master floats data in the Bus.* The LTL representation based on the state machine of fig 3.4 is the following.

$$G((fsm1\_state = TRANSFER) \ \wedge \ (fsm2\_state = ADDR) \ \wedge \ (req \wedge rdy)$$

$$\Rightarrow X(fsm1\_state = TRANSFER) \wedge X(fsm2\_state = DATA))$$

RTL spec ($\mathcal{R}$): The RTL specs remain the same as earlier case.

We apply the following formula to verify whether the RTL specs cover the architectural specs.

$$\mathcal{R} \ \Rightarrow \ ((\bigwedge_j \mathcal{T}_{A_j}) \Rightarrow \mathcal{A})$$

The SpecMatcher returns *complete coverage* as expected.

RTL State Machine: Most often, the RTL specs are also quite complex requiring context in the specification. In order to incorporate the context of a spec, we can use the state machine for RTL too. As an example, we take the running example of MyBus protocol. The state machine for the RTL is presented in fig 3.10.

Let, the RTL has two extra signals: d and a, which are invisible to the architecture. Whenever, d (or a) is asserted, the data (or address) is floated in the Bus. Either, both of them are low, or any one of them is high. But both cannot be asserted simultaneously ever. So there are three states (S_0, S_1 and S_2) as shown in fig 3.10. Next, the specs are presented. This time the RTL specs also comprise of the states from its state machine. The state is encoded using a two-bit variable rtl_state.

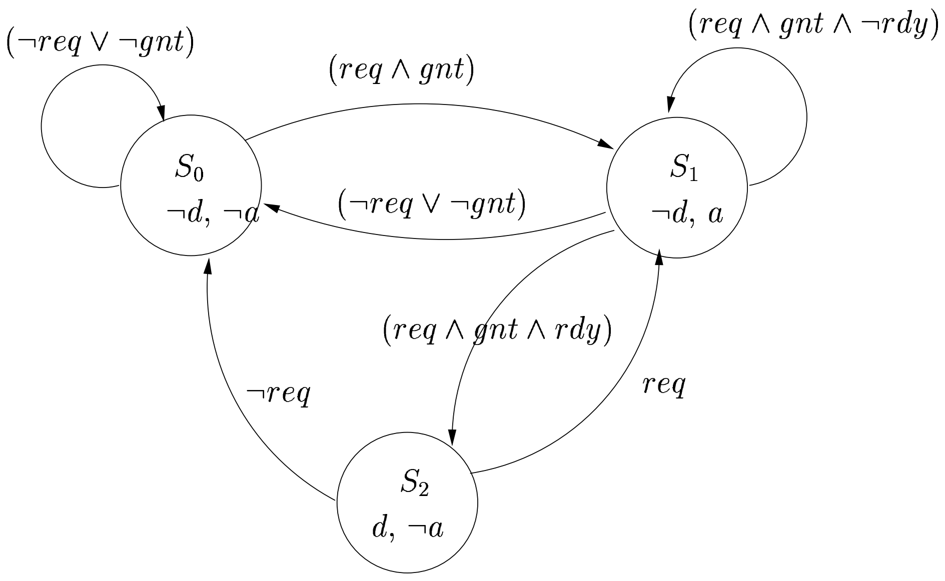

$(\neg req \vee \neg gnt)$
$(req \wedge gnt)$
$(req \wedge gnt \wedge \neg rdy)$
S_0
$\neg d, \neg a$
$(\neg req \vee \neg gnt)$
S_1
$\neg d, a$
$(req \wedge gnt \wedge rdy)$
$\neg req$
req
S_2
$d, \neg a$

Figure 3.10: RTL State Machine of MyBus Protocol Master

Arch. spec ($\mathcal{A}$): The architectural specs remain same as the previous case.

$$G((fsm1\_state = TRANSFER) \;\wedge\; (fsm2\_state = ADDR) \;\wedge\; (req \;\wedge\; rdy)$$

$$\Rightarrow X(fsm1\_state = TRANSFER) \wedge X(fsm2\_state = DATA))$$

RTL spec ($\mathcal{R}$): We modify the earlier RTL spec to include state-bits.

$$G((rtl\_state = S_2) \;\Rightarrow\; X(\neg req \;\vee\; \neg gnt \;\vee\; \neg rdy))$$

By applying equation 3.22 given as follows:

$$(\mathcal{T}_R \Rightarrow \mathcal{R}) \Rightarrow (\bigwedge_j \mathcal{T}_{A_j}) \Rightarrow \mathcal{A}))$$

we find that the RTL specs cover the architectural specs.

Handling the Gaps: Till now, we have not considered any gaps in the architectural specs. Next, consider the following architectural spec in the perspective of the factored state machine (refer fig 3.9). Here, we consider two cases. First, where the RTL state machine is absent and second, where the state machine is present.

Arch. spec ($\mathcal{A}$): *The transfer state machine changes state from ADDR to DATA on* `req`. Formally, we state it as follows.

$$G((fsm2\_state = ADDR) \ \wedge \ (rdy) \Rightarrow X(fsm2\_state = DATA))$$

RTL spec ($\mathcal{R}$):

- *Case 1:* The RTL state machine is absent.
  The three signals (`req`*,* `gnt`*,* `rdy`*) will not remain high for two consecutive cycles.* The LTL representation is the following.

  $$G((req \wedge gnt \wedge rdy) \Rightarrow (X(\neg req) \vee X(\neg rdy) \vee X(\neg gnt)))$$

  In this case, the governing equation is,

  $$\mathcal{R} \ \Rightarrow \ ((\bigwedge_j \mathcal{T}_{A_j}) \Rightarrow \mathcal{A})$$

- *Case 2:* The RTL state machine is present.
  The earlier RTL property (in Case 1) is modified a bit in this case.

  $$G((rtl\_state = S_2) \ \Rightarrow \ X(\neg req \ \vee \ \neg gnt \ \vee \ \neg rdy))$$

  Here, the governing equation will be,

  $$(\mathcal{T}_R \Rightarrow \mathcal{R}) \Rightarrow ((\bigwedge_j \mathcal{T}_{A_j}) \Rightarrow \mathcal{A})$$

Next we discuss the adopted approach in order to meaningfully present the gap. The steps for presenting the gaps are elaborated below.

1. Rename the X-guarded architectural state-bits in $\mathcal{A}$ according to *State-bit Renaming Heuristic.*

2. Execute the regular **Push_Term** algorithm to get the following gap.

$$G(rdy \ \wedge \ \neg req \ \wedge \ (fsm2\_state = ADDR) \Rightarrow \ X(fsm1\_state = IDLE)$$

$$\vee \ (X(fsm1\_state = INIT) \ \wedge \ X(fsm2\_state = DATA))$$

$$\vee \ (X(fsm1\_state[0]) \ \wedge \ X(fsm2\_state = DATA)))$$

3. Perform *universal abstraction* of the *fsm1_state* bits in the obtained complex gap. Those bits disappeared, and hence, a weaker gap is obtained. The final gap, in terms of the variable *fsm2_state* and interface signals, is presented as follows,

$$G((fsm2\_state = ADDR) \wedge rdy \wedge \neg req \Rightarrow X(fsm2\_state = DATA))$$

3.5 Results on SpecMatcher

Circuit	No. of RTL properties	Time (sec)	
		Primary Coverage Question	Gap Finding Time
ARM AMBA AHB	29	14.17	25.7
MyBus Example	2	0.18	1.2

Table 3.1: Runtimes of SpecMatcher

SpecMatcher is our tool for verifying design intent coverage. The original tool used to accept only LTL specifications. With the new methods, the tool now also accepts properties with the auxiliary state machine. Table 3.1 shows the runtimes of our tool on several designs. For each design, we selected an architectural property which requires contributions from multiple submodules. For example, ARM AMBA AHB is bus protocol involving master, slave and arbiter devices. The exact arbitration policy is not defined in the protocol, we therefore targeted a system level property with the RTL of the arbiter and set of properties over the master and slave. The tool accepted all 29 RTL properties and the RTL of the arbiter and produced the coverage gap in less than a minute. The last design is the toy example demonstrated in this thesis. The time break-ups show the time spent (on a 2GHz P4) by the tool in each of the major steps of the coverage algorithm.

If we bring in larger RTL blocks into the picture, we will have state explosion. Therefore the proposed method should not be viewed as a new way to do model checking. The value of the original methodology lies in providing the validation engineer with a formal proof that the decomposition of the specification is correct, or with a clear representation of the coverage gap. The new methodology aids the process by allowing simple modules (such as glue logic) to be accepted as is, but is still totally inclined towards the original goal.

Chapter 4

Conclusion

With the growth in demand of complex, optimized, fault-critical systems, both the model-based and formal approaches are currently being deeply explored for their inherent power of design-hierarchy abstraction (in order to circumvent the capacity limitation of formal tools) and ease of formal verification (necessary for decrementing the risk for the safety-critical applications like health or space explorations).

In this regard, ADLs integrated with verification support will become inevitable for the designers. Traditional verification approaches, performed using test automation tools or assertions begin at RTL level and therefore key design points might be missed due to the complexity of the system. In this work, we presented a verification methodology, in order to assist the ADL users with a completely self-sufficient, consistent design-and-test environment. The novelty of this approach lies in having stronger control over the complete verification flow, without doing away with the comfort of high abstraction level. Moreover, experiments showed that the integration of TGE and assertion-based verification approach would not burden the tool-chain and micro-architecture generation process, because of its manageable light kernel.

On the other hand, for the formal practitioners, writing *context* sensitive properties will be easier if they adopt auxiliary state-machine based properties to get a more comprehensible view of the whole architecture. With this vision, we envisage that it will be equally important for the designer to find out the gap in the architectural level. So the major contribution of this thesis are the following.

Design Intent Verification in ADL: In our work we have achieved seamless integration of semi-formal assertion based verification along with the novel test-generation framework for the Architecture Description Language based processor design to get the bugs detected with minimal number of the instructions.

Formal Design Intent Verification: Given the architecture and RTL properties in presence of auxiliary state-machines, we have extended the design intent coverage paradigm. We have also studied the variant of this problem when there are *concrete* RTL modules in the system.

Appendix

A: Syntax and Semantics of LTL

The formal syntax and semantics of LTL [38] is defined over a Kripke structure. Formally, we define a Kripke structure as a tuple, $K = \langle \mathcal{AP}, S, \tau, s_0, \mathcal{F} \rangle$, where:

- S is a finite set of states,
- $\mathcal{AP}$ is a set of atomic propositions labeling S,
- $\tau \subseteq S \times S$ is the transition relation, which must be total (for all states $s_i \in S$, there exists a state $s_j \in S$ such that $(s_i, s_j) \in \tau$),
- $s_0 \subseteq S$ is the set of start states,
- $\mathcal{F} : S \rightarrow 2^{\mathcal{AP}}$ is a labeling of states with atomic propositions true in that state.

A *path*, π, in the Kripke structure is an infinite sequence of states, $s_0, s_1, \ldots$, such that for all i, $s_i \in S$, and $(s_i, s_{i+1}) \in \tau$. s_0 is called the starting state of π. We use π_j to denote the suffix of π starting from s_j.

The formal syntax of LTL is as follows:

- Each atomic proposition in $\mathcal{AP}$ is a LTL formula.
- If f and g are LTL formulas, then so are $\neg f$, $f \wedge g$, $X\ f$, $f\ U\ g$.

The formal semantics of LTL is as follows (f and g are LTL formulas; p is an atomic proposition; $\pi = s_0, s_1, \ldots$ is a path in K):

- $\pi \models p$ iff $p \in \mathcal{F}(s_0)$

- $\pi \models \neg f$ iff $\pi \not\models f$
- $\pi \models f \wedge g$ iff $\pi \models f$ and $\pi \models g$
- $\pi \models X\ f$ iff $\pi_1 \models f$
- $\pi \models f\ U\ g$ iff $\exists j$, such that $\pi_j \models g$ and $\forall i, i < j$ we have $\pi_i \models f$

B: Publications and Communications made

1. Bhaskar Pal, **Arnab Sinha**, Pallab Dasgupta, P.P.Chakrabarti, Kaushik De, Hardware Accelerated Constrained Random Test Generation, Accepted in IET Computers and Digital Techniques.
2. Anupam Chattopadhyay, **Arnab Sinha**, Diandian Zhang, Rainer Leupers, Gerd Ascheid, H. Meyr, Integrated Verification Approach during ADL-Driven Processor Design. Submitted in the special issue of Rapid System Prototyping in Elsevier Microelectronic Journal, 2007.
3. Anupam Chattopadhyay, **Arnab Sinha**, Diandian Zhang, Rainer Leupers, Gerd Ascheid, H. Meyr, ADL-driven Test Pattern Generation for Functional Verification of Embedded Processors. To appear in 12th *IEEE European Test Symposium*, May 21-24, 2007.
4. Anupam Chattopadhyay, **Arnab Sinha**, Diandian Zhang, Rainer Leupers, Gerd Ascheid, H. Meyr, Integrated Verification Approach during ADL-Driven Processor Design. In 17th *IEEE International Workshop on Rapid System Prototyping*, 2006

Bibliography

[1] Pallab Dasgupta. *A Roadmap for Formal Property Verification.* Springer, 2006.

[2] CoWare/LISATek. *http://www.coware.com.*

[3] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Design Automation and Test in Europe (DATE)*, pages 678–683, 2005.

[4] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA.* Kluwer Academic Publishers, 2002.

[5] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[6] A. Fauth et al. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, Mar. 1995.

[7] A. Hoffmann and O. Schliebusch and A. Nohl and G. Braun and O. Wahlen and H. Meyr. A Methodology for the Design of Application Specific Instruction-Set Processors Using the Machine Description Language LISA. In *Proc. of the Int. Conf. on Computer Aided Design (ICCAD)*, Nov. 2001.

[8] S. Bashford and U. Bieker and B. Harking and R. Leupers and P. Marwedel and A. Neumann and D. Voggenauer. The MIMOLA Language, Version 4.1. Reference Manual, Department of Computer Science 12, Embedded System Design and Didactics of Computer Science, 1994.

[9] MIPS Technologies. *http://www.mips.com.*

[10] Tensilica. *http://www.tensilica.com.*

[11] W. Mao and R. K. Gulati. Improving gate level fault coverage by RTL fault grading. In *Proceedings of International Test Conference, 1996.*

[12] Synopsys. *VCS*
http://www.synopsys.com/products/simulation/simulation.html.

[13] T. Kempf, M. Dörper et al. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2005.

[14] Calypto Design Systems. *http://www.calypto.com/.*

[15] OpenVera. *http://www.open-vera.com/.*

[16] H. Koo and P. Mishra. Functional Test Generation using Property Decompositions for Validation of Pipelined Processors. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 2006.

[17] A. Banerjee, B.Pal, S. Das, A. Kumar, P. Dasgupta. Test generation games from formal specifications. In *Proceedings of DAC*, 2006.

[18] F. Corno et al. Automatic test program generation: A case study. In *IEEE Design and Test of Computers*, 2004.

[19] O. Luethje. A methodology for automated test generation for lisa processor models. In *The Twelfth Workshop on Synthesis And System Integration of Mixed Information technologies, Kanazawa, Japan.* Synthesis And System Integration of Mixed Information technologies (SASIMI 2004), October 18-19, 2004.

[20] Target Compiler Technologies. *http://www.retarget.com.*

[21] A. Adir, E. Almog et al. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design and Test*, 2004.

[22] F. Fallah, S. Devadas and K. Keutzer. Functional Vector Generation for HDL models using Linear Programming and 3-satisfiability. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, 1998.

[23] P. Mishra and N. Dutt. Automatic functional test program generation for pipelined processors using model checking. In *Seventh IEEE International Workshop on High Level Design Validation and Test (HLDVT)*, 2002.

[24] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Design Automation and Test in Europe (DATE), Paris, France*, pages 182–187, February 16-20, 2004.

[25] P. Mishra et al. A top-down methodology for validation of microprocessors. In *IEEE Design and Test of Computers (Design and Test)*, pages 122–131, 2004.

[26] P. Grun and A. Halambi and A. Khare and V. Ganesh and N. Dutt and A. Nicolau. EXPRESSION: An ADL for System Level Design Exploration. Technical Report 98-29, Department of Information and Computer Science, University of California, Irvine, Sep. 1998.

[27] A. Chattopadhyay, A. Sinha, D. Zhang, R. Leupers, G. Ascheid, H. Meyr. Integrated Verification Approach during ADL-Driven Processor Design. In *Seventeenth IEEE International Workshop on Rapid System Prototyping*, 2006.

[28] A. Chattopadhyay, A. Sinha, D. Zhang, R. Leupers, G. Ascheid, H. Meyr. ADL-driven Test Pattern Generation for Functional Verification of Embedded Processors. In *12th IEEE European Test Symposium (To Appear)*, May 21-24, 2007.

[29] O. Schliebusch, A. Chattopadhyay, E. M. Witte, D. Kammler, G. Ascheid, R. Leupers and H. Meyr. Optimization Techniques for ADL-driven RTL Processor Synthesis. In *IEEE Workshop on Rapid System Prototyping (RSP)*, Montreal, Canada, June 2005.

[30] Gaisler Research. *http://www.gaisler.com/.*

[31] T. Gloekler and S. Bitterlich and H. Meyr. ICORE: A Low-Power Application Specific Instruction Set Processor for DVB-T Acquisition and Tracking. In *Proc. of the ASIC/SOC conference*, Sep. 2000.

[32] P. Basu, S. Das, A. Banerjee, P. Dasgupta, P.P. Chakrabarti, C.R. Mohan, L. Fix, R. Armoni. Design Intent Coverage - A new paradigm for Formal Property Verification. In *Computer-Aided Design of Integrated Circuits and Systems*, October 2006.

[33] SpecMatcher. *http://www.facweb.iitkgp.ernet.in/~pallab/forverif.html.*

[34] A. Banerjee, B. Pal, P. Dasgupta, P.P. Chakrabarti, M. Jha., E. Cerny. Design Issues for Assertion-Based Verification IPs: The OVA Experience. In *SNUG*, 2004.

[35] Sayantan Das, Prasenjit Basu, Pallab Dasgupta, and P. P. Chakrabarti. What lies between design intent coverage and model checking? In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1217–1222, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[36] A. Das, P. Basu, A. Banerjee, P. Dasgupta, P. P. Chakrabarti, C. Rama Mohan, L. Fix, and R. Armoni. Formal verification coverage: computing the coverage gap between temporal specifications. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 198–203, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Prasenjit Basu. *Design Intent Verification by Formal Property Coverage.* PhD thesis, IIT Kharagpur, 2006.

[38] E.M. Clarke and O. Grumberg and D. A. Peleg. *Model Checking.* MIT Press, 1999.