UNIVERSITY OF CALIFORNIA
RIVERSIDE

Verification and Analysis of System Designs With Functional and Performance Constraints

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xi Chen

August 2005

The Dissertation of Xi Chen is approved:

Committee Chairperson

University of California, Riverside

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to Prof. Harry Hsieh, who has led me to the research field and guided me throughout the entire graduate career with his constant enthusiasm and patience. Harry has taught me almost everything about being a qualified researcher. Without his invaluable encouragement and support, I would not have been able to finish this work.

I am also grateful to Dr. Felice Balarin and Dr. Yosinori Watanabe from Cadence Berkeley Laboratories for their important advice and contributions to this work. I have benefited significantly from the collaboration and discussions with them. Their suggestions and comments have always substantially influenced and improved the final results.

I must also thank team members of the Metropolis project led by Prof. Sangiovanni-Vincentelli from University of California at Berkeley. The work presented here has mostly been done within the framework of the Metropolis project, and it has benefited greatly from many discussions with team members.

Thanks also go to the computer architecture group led by Prof. Laxmi Bhuyan for providing me with their network processor simulator NePSim, a comprehensive and realistic experimental platform for several case studies presented in this thesis.

To my wife and parents.

ABSTRACT OF THE DISSERTATION

Verification and Analysis of System Designs With Functional and Performance Constraints

by

Xi Chen

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2005
Dr. Harry Hsieh, Chairperson

With the increasing complexity and heterogeneity of today's embedded systems, design methodologies at higher levels of abstraction become a necessity. It is expected that the next major productivity gain will come in the form of system level design since designing at the register transfer level or sequential C-code level is no longer efficient. It follows that new verification and analysis technologies have to be developed in each and every step of the design flow in order to catch design errors as early as possible and to reduce the overall design cost.

Simulation remains a major means of verification for complex system level designs, especially when designs are refined with more design details realized. In this work, a simulation verification methodology is proposed based on trace analysis and automatic trace checker generation. From formal specification of design constraints with mathematical logics such as Linear Temporal Logic (LTL) and Logic of Constraints (LOC), i.e. formal assertions, monitors or checkers are automatically generated and used to verify simulation traces during or after simulation. As a major contribution, LOC formalism is extensively studied, and an efficient checking algorithm is proposed. LOC is also used in automatic generation of distribution analysis tools, which have been exercised on low power techniques in network processor architectures. By utilizing formal assertions, a designer can easily verify both functional and performance constraints of a design in simulation. In addition, a deadlock analysis mechanism is proposed with built-in simulation monitors. This approach is demonstrated in the Metropolis design framework.

For small but important designs or library modules that will be instantiated many times, exhaustive verification is possible and useful. A formal verification methodology for system level design is therefore proposed, where an existing software formal verification tool (e.g. Spin) is utilized as the back-end verification engine, and an automatic translation mechanism from system specifications to verification models is developed. Furthermore, automatic abstraction propagation algorithms can be used to simplify the verification models. In this study, Metropolis is used as a major experiment platform, where a designer is allowed to formally verify design constraints specified with LTL and LOC within the integrated design framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing complexity of embedded systems today demands more sophisticated design and verification methodologies. Systems are becoming more integrated as more and more functionalities and features are required for products to succeed in the market. Embedded system architectures likewise have become more heterogeneous as it is becoming more economically feasible to have various computational resources (e.g. microprocessor, digital signal processor, reconfigurable logics) all utilized on a single chip [62]. Designing at the register transfer level [38] or sequential C-code level is no longer efficient. More than ever, design and verification methodologies at higher levels of abstraction are required to fill the gap between the increasing semiconductor manufacturing capabilities and the lag-behind design productivity.

1.1 System Level Design

The system level design methodology, based on orthogonalization of design concerns, as well as pre-defined platforms, has been proposed for the next major productivity gain [47]. To combat complexity and to explore design space effectively, it is necessary to represent systems at multiple levels of abstraction. Initial specification of the function and the architecture of a system is done at a high abstraction level without particular lower level implementation details. The function is then mapped onto the architecture after iterations of refinement procedures (see Figure 1.1). Significant advantages in design flexibility, as compared to the traditional fixed architecture and *a priori* partitioning approach, can result in significant advantages in the performance and design cost of the product.

Synthesis (i.e. steps taken toward implementation) is applied systematically to transform high level specifications to manufactured products. Synthesis steps may include structural transformations, a design is partitioned, composed, or otherwise altered; formal refinements, where possible behaviors of a design are formally refined through the use of constraints or implementation annotations; and mapping, where the functional specification at a particular abstraction level is mapped to an architectural specification at a particular abstraction level. There therefore exist multiple levels of abstraction in a design flow, which also indicates necessity for suitable verification techniques to be applied at each level. A formal grounding for all system representations and operations is essential for the ability to perform analysis and optimization with high degree of automation. Furthermore, abstraction is an effective oper-

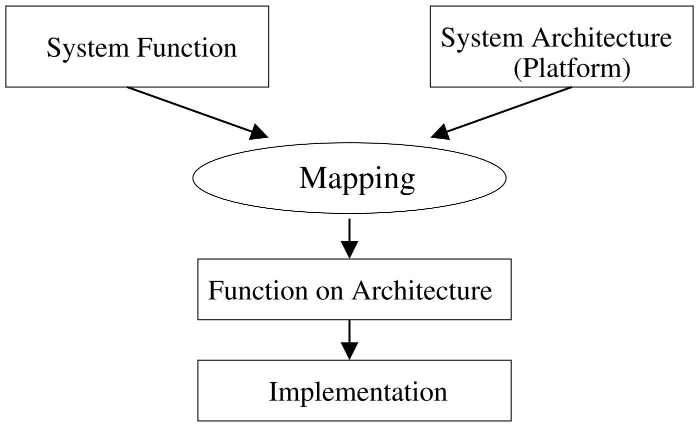


Figure 1.1: *System level design methodology*

ation to manage complexity during verification procedures. The tendency is to abstract (or simplify) the design for verification purposes and to refine the design as more implementation details are determined.

1.2 Verification Methods

In general, verification is a process to make sure if a design is what a designer intends to design. Due to the increasing complexity of today's embedded system designs, errors are likely to happen at all stages in the design flow. It has been reported that more than 70% of the development time is spent on design verification, and verification is becoming the bottleneck in the semiconductor industry according to International Technology Roadmap for Semiconductors (ITRS) [11]. This number is even expected to grow in the future and imposes yet unsolved challenges on tomorrow's design automation industry. Therefore, verification of system designs (embedded hardware/software systems) is one of the most important tasks in the design process. To cope with the increasing complexity, various attempts have been made to increase design productivity.

Traditionally, most verification techniques have been based on simulation and testing methods [15, 40, 31]. At high levels of abstraction, executable simulation models are built from design specifications. Since it is usually prohibitive to exhaustively simulate all the possible execution paths of a simulation model, test cases are carefully designed or selected to achieve as much coverage as possible [1]. Testing is done at a lower abstraction level once a product prototype is available, and random test cases are automatically generated and used to check if the execution of the prototype is correct.

More recently, formal methods such as temporal property checking or symbolic model checking have become increasingly popular [51, 27]. Formal verification techniques attempt to overcome the weakness of non-exhaustive simulation by proving the correspondence between some abstract design specification and the original design. The abstract model of a design is represented symbolically or with efficient data structures such as hash tables, and then the entire state space is searched for any design errors or property violation. The complexity of searching the entire state space is at least exponential to the number of states, so formal verification techniques are expensive, and their applicability is currently restricted to small or medium sized designs or to a specific phase in the design cycle.

To make the practice of designing from high level system specification a reality, verification methods must accompany every step in the design flow. Specification at the system level

[1]Verification coverage is the percentage of a design that is checked in the verification

makes formal verification possible. Designers can prove a property of a design by writing down the property they want to check in some logic (e.g. CTL [36] and LTL [59]), generate verification models from the design specification automatically or semi-automatically, and use a formal verification tool (e.g. the model checker SMV [55] and Spin [42]) to run exhaustive verification. Then the entire state space of the design can be searched to verify the specified property without any uncertainty.

As designs are refined with more implementation details realized, however, the complexity can quickly overwhelm the automatic tools, and simulation becomes the primary means for verification. The confidence of a simulation verification mainly depends on the design or selection of test cases. In order to uncover bugs of designs during simulation, designers can insert embedded assertions, i.e. formally specified design properties or constraints, into their design specifications in hardware description languages or high level modeling languages. Today's assertion languages capture those simple logics as language or platform specific library blocks. A set of extended temporal connectives or regular expression operators are then used to operate on those blocks for expressing more complex assertions. Examples of assertion languages include PSL [6] and OpenVera [4].

1.3 Functional and Performance Constraints

In this work, Linear Temporal Logic (LTL) [59] and Logic of Constraints (LOC) [19] are two main logics used for specification of design constraints. It will be shown that LTL and LOC have different domains of expressiveness and indeed complement each other quite well. At the verification stage, both static and runtime verification techniques can be used to check the design constraints and to report design errors if there is any constraint violation.

LTL is suitable for specifying functional constraints, such as mutual exclusion, liveness, and safety, and can effectively describe the temporal patterns of system state transitions. LTL is defined over *executions* of a system, i.e. linear sequences of *state transitions*. LTL formulas are constructed using terms, i.e. Boolean expressions on variables or system states, classical Boolean operators such as $\neg$ (not), $\vee$ (or), $\wedge$ (and), and $\rightarrow$ (imply), and the linear temporal operators G (always), F (eventually), X (next), and U (strong until). For example, G(A) is true if A is true for all states, F(A) is true if A eventually becomes true in a future state, X(A) is true if A is true in the following state, and A U B is true if B eventually becomes true in a future state and A is true from the current state to that future state.

It has been proved that LTL formulas can be translated to equivalent Büchi automata [63]. Based on this theory, formal techniques like model checking are developed and utilized for verification of both digital designs (e.g. SMV [55]) and software protocols(e.g. Spin [33]). LTL is also used as the basis for the formal constraint specification for simulation verification [4, 29], which is important to assure the integration and correctness of reusable IP (Intellectual Property) blocks. LTL has been a very popular and well studied logic for more than a decade, so its details will not be covered in this thesis.

We believe that the existing logics or hardware assertion languages are not natural to express more abstract constraints such as transaction level constraints, where only the events observable from a system and their annotations are considered. Nor are they convenient to directly express performance constraints that are quantitative in nature (e.g. latency or throughput). To this end, we propose a constraint formalism: Logic of Constraints (LOC). LOC is designed for specification of performance constraints such as rate, throughput, and latency, as well as quantitative functional constraints such as I/O data consistency at the transaction level, where system events and their annotations are considered. It is very well-suited for analyzing traces from execution of higher, transaction level system models. LOC consists of all the terms and operators allowed in sentential logic [30], with additions that make it possible to specify quantitative constraints without compromising the ease of analysis. The basic components of an LOC formula include event names (e.g. $pipeline$ and $sram\_enq$), instances of events (e.g. $pipeline[4]$), indices of event instances (e.g. 0, 1, ..., etc), the index variable i, and annotations (e.g. $cycle$, pc, and $addr$). LOC can be used to specify many important system level performance constraints that are inconvenient, and sometimes impossible, to specify with other logics. For example, the rate constraints:

$$cycle(pipeline[i+1]) - cycle(pipeline[i]) = 10 \tag{1.1}$$

requires that the difference between the values of annotation $cycle$ for any two consecutive instances of $pipeline$ event should equal to 10. A complete study of LOC will be presented in Chapter 2.

Constraint definition is central to many methodologies. A general approach is taken by the Rosetta [16] language: different domains of computation are described declaratively and constraints can be expressed as predicates on some defined quantities. Constraints are then applied by combining the different domains. In this work, we restrict the scope of constraint definition in favor of a representation that is more natural to the designer and that is more computationally tractable.

Object Constraint Language (OCL) [1], part of the Unified Modeling Language (UML), takes a more restricted approach. OCL supports invariants, pre- and post-conditions, and guards, applied to classes, operations of classes, and states, respectively. Another related proposal is the Design Constraints Description Language (DCDL) [2] sponsored by Accellera, which is intended mostly for low-level (i.e. chip-level) constraints like clock slew, operating voltages, and port capacitances. In both of these approaches, constraints are specified for a collection of entities that represent a system (classes and their operations and states in case of OCL, and physical objects in case of DCDL). This facilitates specifying constraints associated with the system as a whole, e.g. area, yield, testability, and time to market. In contrast, we focus on specifying constraints for particular executions of a system, like response time, energy consumption, and memory usage. OCL also supports this, to some extent, through pre-conditions, post-conditions, and guards. However, while these constructs naturally express constraints on a single transition, LOC makes it easy to express constraints that span several transitions. In fact, in our approach, it is easy to specify constraints for which it is impossible to bound in advance the number of transitions needed to check them.

Many constraint formalisms have been proposed that are at most as expressive as ω-regular languages (and in some case strictly less expressive). An incomplete list includes

S1S [22], LTL [59], PSL [6], HAAD [24], and many variants of finite-state automata on infinite words, e.g. [17, 37]. MONA [39], on the other hand, is based on regular languages and finite-state automata on finite words. It is believed that LOC is a good complement to all these approaches, as there are certain natural constraints (e.g. *data consistency* that are not ω-regular, but can be expressed and verified (both formally and by simulation) using LOC.

Real-Time Logic (RTL) [45] is a formalism for expressing timing constraints in real-time systems. With RTL, the constraints are specified by means of timing relations on occurrences of events. RTL was primarily intended for formal reasoning, while LOC is more biased toward simulation monitoring. For example, RTL allows any number of index and time variables which can be arbitrarily quantified. This makes it very unsuitable for verification by simulation. In contrast, LOC allows only one index variable and no time variables or quantification. This choice is made precisely for the purpose of efficient simulation monitoring. Also, arithmetic in RTL is limited to Presburger arithmetic (i.e. linear inequalities), to ease formal reasoning, while LOC allows more complex expressions, because they can be handled quite easily in simulation. This separation of purposes is not total. When we consider a subset of LOC suitable for formal verification, we restrict LOC to Presburger arithmetic. Similarly, Mok and Liu have proposed a subset of RTL suitable for simulation monitoring [56, 57], and that subset indeed resembles LOC. However, they have not proposed any automatic formal verification technique for that subset. A subset of LOC suitable for formal verification can be seen as a generalization of the subset of RTL suitable for simulation monitoring, as it allows specification of constraints related to annotations other than time. In fact, data consistency, one of the constraints that distinguish LOC from formalisms based on ω-regular languages, also distinguishes it from RTL, as it does not involve time at all.

1.4 Metropolis Design Framework

Metropolis is developed as an integrated and unified design framework for next generation system level design [20]. Metropolis allows designers to represent and manipulate their designs at multiple levels of abstraction and with multiple models of computation (MoC). Central to the design framework is a high level modeling language, Metropolis Meta-Model (MMM), and a set of back-end tools are integrated into the framework, with which one can simulate, synthesize, and verify a design at hand. Metropolis is used as the main experiment platform throughout this thesis.

1.4.1 Framework and Design Methodology

The integrated design environment consists of a design specification language, Metropolis Meta-Model (MMM), a front-end that constructs intermediate representations and analyzes static network structures, and a set of back-end tools that are responsible for simulation, synthesis, verification, and other tasks. Different high-level languages, models of computation, design constraints, as well as specifications of system functions, architecture platforms, and function-architecture mappings can be represented in MMM while retaining their precise semantics. Constructs in MMM are designed to facilitate transformations and refinements

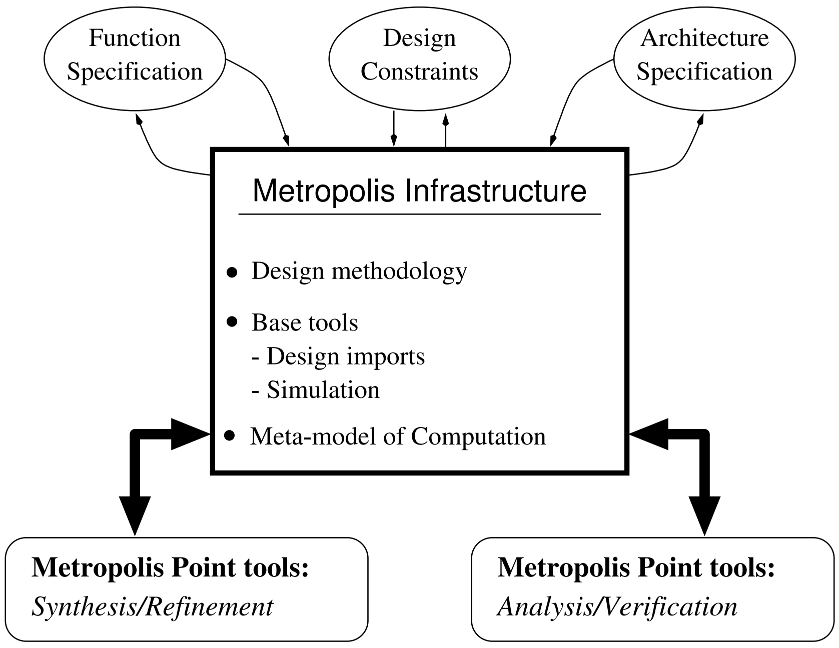


Figure 1.2: *Metropolis design framework*

between different abstraction levels. Different design aspects are orthogonalized, such as computation versus communication, function versus architecture, and specification versus implementation. The design complexity can therefore be effectively reduced, and the design space can be efficiently explored. Figure 1.2 shows a flow diagram for the Metropolis framework.

1.4.2 Metropolis Meta-Model Language

Metropolis Meta-Model is a system representation formalism capable of representing designs at different levels of abstraction. A description of a system (function and/or architecture) can be made in terms of computation, communication, and coordination.

Processes, Media, and Netlists

In Metropolis Meta-Model, systems are represented as networks of *processes* that communicate through *media* [18]. Processes and media are used to describe computation and communication respectively. The syntax of MMM is similar to Java but includes many system level modeling extensions. A process defines an active object and always includes a method called *thread* as the top-level method, where its behavior is specified. A communication medium implements a set of methods that are declared in *interfaces*. Processes connect to media through *ports*. Each port has a type that must be an interface implemented by the medium to which the port is connected. Processes communicate to each other by invoking interface methods implemented in the shared media through these ports.

In MMM, objects such as processes, media, and their connectivities can be grouped in a *netlist*, which is used to model a complete network. Figure 1.3 shows an example of a functional netlist *FuncNet*. The netlist defines two processes, *p1* and *p2*, communicating through a medium *m1*. A netlist can also contains other netlists to form a hierarchical network. In addition, refinement constructs are available to specify that one netlist is the formal refinement of another within a network.

Coordination

Processes run concurrently, each at its own pace. The relative speed of processes may arbitrarily change at any time, unless they synchronize with each other using the synchronization primitive called *await*, or if *constraints* are specified in the system. The await statement can

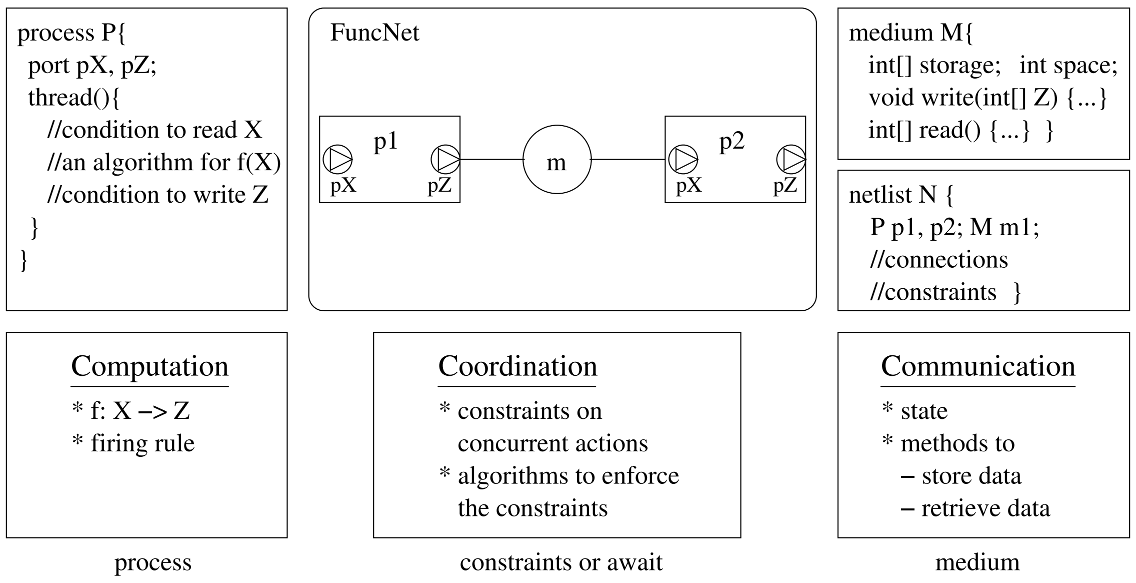


Figure 1.3: *An example of MMM specification*

be used to make a process wait until some condition holds and establish critical sections that guarantee mutual exclusion among different processes. To limit the behavior of processes, a designer can also specify high-level LTL (Linear Temporal Logic) [59] or LOC (Logic of Constraints) [19] constraints and leave the implementation of these constraints to the detail design stage.

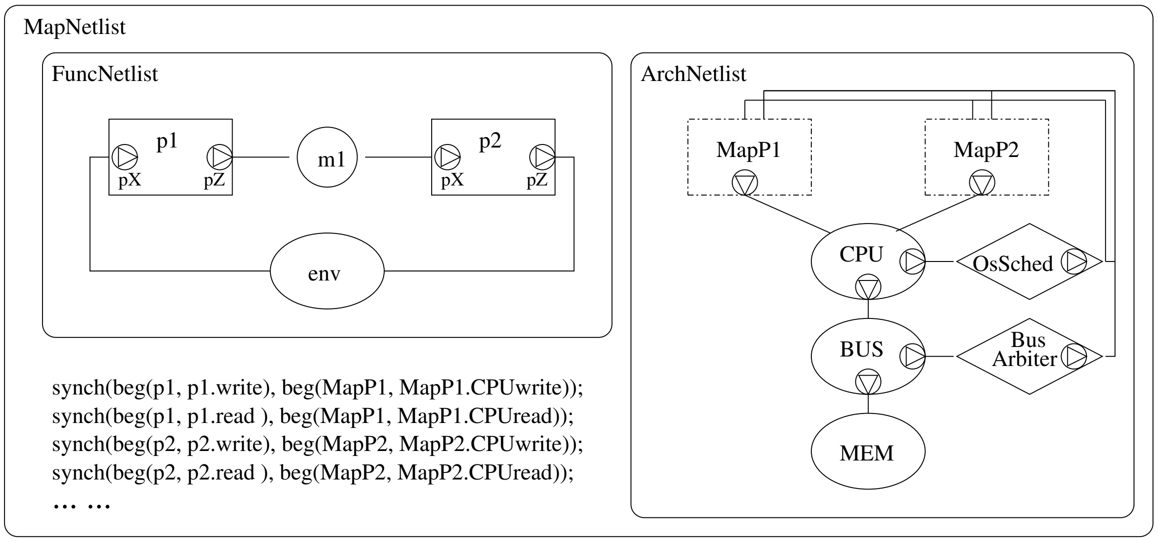
The await statement is used to establish mutually exclusive sections and synchronize processes. It contains one or more statements called *critical sections*, each controlled by a triple (*guard*; *testlist*; *setlist*), where the guard is a Boolean expression, and the testlist and setlist denote sets of interface methods that other processes can call. A critical section is said to be *enabled* if its guard is true, and none of the interface methods in the testlist are being executed at that moment. A critical section may start executing only if it is enabled. In addition, while the critical section is being executed, no interface methods included in the setlist can begin their executions. Whenever an await is encountered in the execution flow, one and only one of the enabled critical sections is executed. If no critical section is enabled, the execution blocks. If more than one critical sections are enabled, the choice is non-deterministic.

Function, Architecture, and Mapping

The function-architecture separation and mapping are natively supported in the Metropolis Meta-Model language. The function and architecture of a system are defined independently at a high level of abstraction. The function is then mapped to the architecture in order to arrive at a given implementation.

Both the function and architecture of a system are modeled as separate networks of processes communicating through media. In an architectural network, resources are typically modeled with media, services that the architecture can provide are modeled with so called *mapping processes*, and arbitrators among multiple architectural resources are modeled with *quantities*. A third network can be defined to encapsulate the functional and architectural networks and to *relate* the two by synchronizing events between them with *synch* constraints.

Figure 4.6 shows a mapping network *MapNetlist* that correlates the functional network *FuncNetlist* with the architectural network *ArchNetlist*. The functional network includes two processes *p1* and *p2* communicating through media *m1* and *env*. The architectural network contains media *CPU*, *BUS* and *MEM*, and the corresponding mapping processes. The synch constraints are used to synchronize the events from the functional processes and the mapping



```
synch(beg(p1, p1.write), beg(MapP1, MapP1.CPUwrite));
synch(beg(p1, p1.read ), beg(MapP1, MapP1.CPUread));
synch(beg(p2, p2.write), beg(MapP2, MapP2.CPUwrite));
synch(beg(p2, p2.read ), beg(MapP2, MapP2.CPUread));
... ...
```

Figure 1.4: *Function-architecture mapping*

processes. Schedulers *OsSched* and *BusArbiter*, which are modeled with quantities, coordinate the architectural resources and provide performance models to the architectural network. During execution, architectural media and mapping processes can request the quantitative annotations from the quantities.

1.5 Thesis Overview

In this thesis, we present a complete study of system level verification and analysis techniques based on formal specification of design constraints. The focus is to automate the verification and analysis process at different stages of the design flow. The rest of the thesis is organized as follows. In the next chapter, a complete study of LOC formalism is presented. The syntax, semantics, and verifiability of LOC are discussed in detail. In addition, the verification and analysis algorithms for LOC formulas are proposed.

In Chapter 3, we focus on assertion-based simulation verification and analysis for system level designs. We discuss how assertion languages based on mathematical logics such as LOC and LTL are used in simulation verification for both functional and performance design constraints. Furthermore, LOC formulas are shown to be useful in design exploration with performance and power trade-off analysis by automatically generating quantitative distribution analyzers. The techniques are demonstrated with case studies of a network processor architecture design.

A simulation-based deadlock analysis mechanism is presented in Chapter 4. We show that, for certain common design constraints such as deadlock or starvation, a built-in detection and analysis method based on simulation is more efficient to use than general assertion languages or formal methods. The causes of deadlock problems are analyzed, and data structures and algorithms are proposed for simulation time deadlock monitoring and analysis. The experiments are done within the Metropolis framework and used to show the effectiveness of the approach.

In Chapter 5, we propose a formal verification methodology for system level designs. In this approach, an existing software formal verification tool is utilized as the back-end verification engine, and system specifications are automatically translated into lower level verification models. A designer is then allowed to verify formally specified design constraints, and to refine the design or the constraints according to the verification results. In addition, an automatic abstraction propagation technique is proposed to simplify verification models. By implementing this methodology, a verification back-end tool is integrated in the Metropolis framework, and its usefulness and effectiveness are demonstrated through several case studies.

Chapter 6 concludes the thesis and summarizes the contributions of this work.

Chapter 2

Logic of Constraints

In this chapter, we introduce our quantitative constraint formalism, Logic of Constraints (LOC). LOC is particularly suited for specification and simulation analysis of performance constraints at the transaction level, where only the events observable from the system and their annotations are considered, as will be shown later in this chapter.

2.1 Introduction to LOC

The LOC formalism is compatible with a wide range of functional specification formalisms that describe a system as a network of components communicating through fixed interconnections. The observed behavior of the system is usually characterized by sequences of values observed at the interconnections. LOC is a formalism designed to reason about traces from the execution of a system. It consists of all the terms and operators allowed in sentential logic, with additions that make it possible to specify system level quantitative functional and performance constraints without compromising the ease of analysis. LOC can be used to specify many common and useful real-time performance constraints.

- rate, e.g. "$Display$'s are produced every 10 time units":

$$t(Display[i+1]) - t(Display[i]) = 10 \ \ , \tag{2.1}$$

- latency, e.g. "$Display$ is generated no more than 25 time units after $Stimuli$":

$$t(Display[i]) - t(Stimuli[i]) \leq 25 \ \ , \tag{2.2}$$

- jitter, e.g. "every $Display$ is no more than 4 time units away from the corresponding tick of the real-time clock with period 10":

$$\mid t(Display[i]) - (i+1) * 10 \mid \leq 4 \ \ , \tag{2.3}$$

- throughput, e.g. "at least 100 $Display$ events will be produced in any period of 1001 time units":

$$t(Display[i+100]) - t(Display[i]) \leq 1001 \ \ , \tag{2.4}$$

- burstiness, e.g. "no more than 1000 $Display$ events will arrive in any period of 9999 time units":

$$t(Display[i+1000]) - t(Display[i]) > 9999 \ \ . \tag{2.5}$$

In addition, LOC can also be used to specify quantitative functional constraints such as the data consistency, e.g. “the input data should be the same as the output data”:

$$data(input[i]) = data(output[i]) \ . \tag{2.6}$$

It should be emphasized that time is only one of the possible annotations. Any value that may be associated with an event (e.g. power, area, data value) can be used as an annotation. In the case of concurrent events, the values of time annotation should be the same. The indices of instances of the same event denote the strict order as they appear in the execution trace. There is no implied relationship between instances of different events. LOC can be used to express relationship between the annotations of the different instances of the same event (e.g. rate), or instances of different events (e.g. latency).

The latency constraint above is truly a latency constraint only if the $Stimuli$ and $Display$ are kept synchronized. Generally, we will need an additional annotation that denotes which instance of $Display$ is “caused” by which instance of the $Stimuli$. If the $cause$ annotation is available, the latency constraint can be more accurately written as:

$$t(Display[i]) - t(Stimuli[cause(Display[i])]) \leq 25 \ , \tag{2.7}$$

and such an LOC formula can easily be analyzed with simulation. However, it is the responsibility of the designer, the program, or the simulator to generate such an annotation.

A constraint formalism is not meaningful unless there exists a clear and efficient path to verification. An efficient simulation-based approach is proposed for analyzing LOC formulas (see Section 2.4). C++ trace checkers are automatically generated from LOC formulas. The checkers analyze the simulation traces and report any constraint violations. In most cases, the traces are scanned only once and memory usage is very low. The automatic checker generation is parameterized, so it can be customized for fast analysis for specific verification environments (e.g. memory limitation). The choice of C++ for the checkers is a matter of convenience. It allows us to tightly integrate the checkers with the SystemC [7] simulator for runtime monitoring. No major difficulty exists to generate checkers in HDLs for integration with hardware simulators, or in Java for concurrent execution with the software simulators.

A simulation-based approach can only disprove the LOC formula (if a violation is found), but it can never prove it conclusively, as that would require analyzing the design space exhaustively. However, for small but important designs or library modules that will be instantiated many times across different designs, it may be necessary to formally prove the desired properties. Formal verification is more expensive though the designers can be more confident about the result. It should be used only for small but important design modules (e.g. Task Transition Level (TTL) channel [32]), possibly in concert with simulation verification of the entire system. An exact verification algorithm exists for a broad class of LOC formulas (see Section 2.5). However, due to the high complexity of this algorithm, an alternative is provided in this study. We propose a formal verification approach where LOC formulas are translated into verification models in Promela (Spin's modeling language [42]) and LTL formulas. This approach is complete for a restricted subset of LOC. It can also be applied to a wider subset, but results might then be inconclusive, i.e. the verification is only partial.

While similar in spirit to the hardware embedded assertion languages, our LOC formalism and simulation verification approach are indeed useful in at least three fundamental aspects. First, Logic of Constraints is designed for specifying all quantitative performance and functional constraints, not just functional ones. This means that one can easily specify requirements on timing or power consumption of the systems being designed, in addition to those on the functional correctness. Second, LOC can be used to specify performance constraints effectively, while many LOC properties cannot be expressed with LTL directly. Third, system level functional and performance constraints written in LOC can be automatically and efficiently synthesized into static checkers, runtime monitors, or formal verification modules.

2.2 LOC Syntax and Semantics

Here we give an informal overview of LOC syntax and semantics. Full details are given in Appendix A. The basic blocks of LOC formulas are terms, which can be either:

- constants, or
- integer variable i (the only index variable that can appear in an LOC formula), or
- expressions of the form $a(e[n])$, where a is an annotation name, e is an event name, and *index expression* n is an integer-valued term, or
- combination of simpler terms using usual arithmetic operators.

We interpret $a(e[n])$ as the value of annotation a of the n-th occurrence of event e. All other terms are interpreted naturally. Terms can be combined using relational operators to create atomic LOC formulas. Finally, LOC formulas are standard Boolean expressions over atomic formulas.

LOC formulas may contain only one index variable, namely i. Having only one index variable may seem very restrictive, but so far we have not found a natural constraint that required more than one. In effect, the ability of defining annotations allows one to specify formulas that otherwise require more than one index variable. On the other hand, having only one index variable enables efficient simulation monitoring.

Models of LOC formulas contain a sequence of occurrences for each event name in the formula. Such structures are called *annotated behaviors*. Each occurrence may be annotated with some annotation, but we do not require each annotation appearing in the formula to be defined. This feature is important for our design methodology, where performance requirements are specified early in the process, even though they can be evaluated much later, when many implementation details are set.

Given an annotated behavior, the formula is evaluated for each value of index variable i. This is done in quite a standard fashion, except that we need to consider the fact that some terms may not be defined (either because there are only finitely many occurrences of an event, or because an annotation is not defined for an existing event occurrence). To deal with this, the third logical value $undef$ is introduced. In general, all operators (including Boolean) return $undef$ if one of their operands are $undef$. The only exceptions are conjunction with

$false$ (which is $false$), and disjunction with $true$ (which is $true$). Finally, the annotated behavior satisfies the formula if it does not evaluate to $false$ for any value of i.

2.3 Expressiveness of LOC

In this section, we discuss the expressiveness property of LOC especially in its relationship with the well known Linear Temporal Logic (LTL). It should be noted that LTL is defined on the state transition level where any change at the system state is accounted for, while LOC works on a higher abstraction level, in which only the events observable from the system and their annotations are considered. This apparent difference, however, is just a technicality, because it is not difficult to hide state transitions so that LTL and LOC are defined over the same kind of objects.

Through several examples and claims, it is concluded that LOC and LTL are incomparable and have different domains of expressiveness.

Claim 1 *There are LOC formulas that can be expressed with LTL.*

Since both LOC and LTL contain basic Boolean expressions, a subset of LOC constraints that specify simple global Boolean conditions can be expressed in LTL also. For example, the constraint, “the annotation $data$ of the event $Display$ is always greater than 100”, is expressed in LOC as:

$$data(Display[i]) > 100 \ . \tag{2.8}$$

If we use a variable $Display\_data$ to store the value of $data$ in the design, and use a flag $Display\_occur$ to indicate that an instance of the event $Display$ occurs, this constraint can be easily expressed in LTL as:

$$\mathsf{G}\ (Display\_occur \Longrightarrow (Display\_data > 100))\ . \tag{2.9}$$

Claim 2 *There are LOC formulas that cannot be expressed with LTL.*

Many quantitative constraints that can be easily expressed by LOC are not suitable for LTL. Specifically, when more than one events need to be compared in the same constraint (e.g. the latency constraint), LTL is not expressive enough to be used. For example, the data consistency constraint:

$$data(input[i]) = data(output[i]) \tag{2.10}$$

requires comparing each instance of $output$ with the instance of $input$ with the same instance index. After the n-th $input$ occurs, it is unknown when the n-th $output$ will occur, i.e. the number of $input$ instances that may occur before the n-th instance of $output$ is arbitrarily large. Therefore, this constraint cannot be modeled by a finite-state system, and it is impossible to express it using any formalism based on ω-regular languages, such as LTL or PSL.

It is interesting to note that there are simple LOC formulas that cannot be expressed by LTL even though they are ω-regular. For example, the property "the value of event A on every even occurrence is 1", can be expressed by LOC formula $data(A[2i]) = 1$, as well as with a simple two-state automaton, but it is well known that it cannot be expressed by LTL [64].

To show that some LTL formulas cannot be expressed in LOC, we first recall that any property can be expressed as a conjunction of a *safety* and a *liveness* property. Safety properties are those which can always be shown violated by a finite trace. For example, any execution that does not satisfy the property "the value of A is never 1" must have a finite prefix which ends with the value of A being 1. On the other hand, liveness properties can never be violated by a finite trace. For example, the property "for every request there is a response" can never be violated by a finite trace because there is always a chance that a response may come some time in the future. [1]

Claim 3 *LOC can express only safety properties.*

Indeed, if a trace does not satisfy an LOC formula, then there must exist an i for which the formula is false. We can evaluate all index expressions for that value of i. Since there can only be finitely many of these expressions, there must exist some point in the execution such that, for that particular i, the formula does not refer to any event occurrence beyond that point. Clearly, the execution prefix up to that point is sufficient to disprove the property.

On the other hand, LTL is capable of expressing some liveness properties, for example $\mathsf{GF}\ (A)$, i.e. "A occurs infinitely often".

Conclusion: From claims (2) and (3), we conclude that LOC and LTL are incomparable.

[1]To disprove a liveness property, we need to show that the system can enter an infinite cycle in which there are unfulfilled requests.

Generally, LOC is designed for the specification of quantitative performance and functional constraints at the transaction level where system events and their annotations are considered. Because of the use of index variable i, LOC is beyond the finite automata domain. On the other hand, LTL is suitable for the specification of functional constraints, and can effectively express the temporal patterns for system state transitions. Because of this difference, LOC can express important properties that cannot be expressed with LTL, on which the traditional property specification languages are based.

In fact, it has been shown that LOC is incomparable with any formalism capable of expressing ω-regular properties. From the theoretical point of view, it may be interesting to establish whether LOC can express all regular properties, i.e. whether LOC is more expressive than WS1S. However, for the methodology proposed here, that question is hardly relevant, because LOC is proposed as a complement to and not a replacement for existing property languages capable of expressing regular properties.

2.4 Checking LOC Formulas with Simulation

In simulation verification, we automatically generate simulation trace checkers from LOC formulas. In the LOC checker, we use a linear-time algorithm to check the simulation trace, which could be infinite, and see if an LOC formula can be satisfied for all possible values of index i. Although the algorithm is linear in time, memory space usage is dependent on the formula heavily. To reduce the running time, we try to scan the whole trace only once and store the annotation information that is expected to be useful in the future. Therefore, a memory recycling procedure has to be invoked frequently to release unnecessary memory space to obtain space efficiency.

The algorithm of LOC checking progresses based on the index variable i. Each LOC formula instance is checked sequentially with the value of i being 1, 2, ... etc. A formula instance is a formula with i evaluated to some fixed positive integer value, e.g. $Display[30] - Display[29] = 10$ is the 29th instance of the formula (2.1). Starting with i equal to 1, the LOC checker scans the trace sequentially. If any relevant data is read in, the checker stores it into a queue and checks the formula in the following manner (Algorithm 1).

Algorithm 1 *Check an LOC formula.*

```
procedure CHECK_LOC_FORMULA()
  while can evaluate formula instance i do
    evaluate formula instance i;
    i++;
    memory recycling
  end while
end procedure
```

The time complexity of the algorithm is linear in the size of the trace since evaluating a particular Boolean expression takes constant time. The memory usage, however, may become prohibitively high if we try to keep the entire trace in the queue for analysis. As the trace file is scanned in, the checker attempts to store only the useful annotations, and in addition, to evaluate as many formula instances as possible, and to remove from the memory parts of the annotations that are no longer needed (memory recycling).

For many LOC formulas (e.g. constraints (2.1), (2.3) - (2.5) in Section 2.1), the algorithm uses a fixed amount of memory no matter how long the traces are (see Table 2.1).[2] Memory efficiency of the algorithm comes from being able to free stored annotations as their associated formula instances are evaluated. This ability is directly related to the choice made in designing LOC. From an LOC formula, we can conservatively identify what annotation data will not be useful anymore once all the formula instances with indices less than a certain number are all evaluated. For example, consider an LOC formula:

$$t(\mathit{Display}[i+10]) - t(\mathit{Stimuli}[i+5]) < 300 \ , \tag{2.11}$$

and let the current value of i be 100. Because the value of i increases monotonically, we know that event $\mathit{Display}$'s annotation t with index less than 111 and event $\mathit{Stimuli}$'s annotation t with index less than 106 will not be useful in the future, and their memory space can be released safely. Each time an LOC formula is evaluated with a new value of i, the memory recycling procedure is invoked, which ensures minimum memory usage.

As described in Section 2.2, the LOC semantics allows us to evaluate an LOC formula even if some of its expressions are not defined. When an annotation with a particular index value is not yet available from the trace, or when the index has an invalid value (e.g. negative value), the Boolean expression that contains this annotation is evaluated to $undef$. The entire LOC formula could then be evaluated according to the standard three-value logic [54]

[2]The verification of the constraint (2.2) may also have constant memory usage if the given trace has a certain regular structure.

evaluation. For example, given the following LOC formula:

$$t(A[i+10]) > 100 \ \vee \ t(B[i-5]) < 300 \ , \tag{2.12}$$

let the current value of i be 10. If we know, from the trace, that the value of $t(A[\mathit{20}])$ is 200, the formula can already be evaluated to true even if the value of $t(B[\mathit{5}])$ is still not available at this point in the simulation (trace). Thus LOC formula instances can be evaluated as soon as possible, which further minimizes the memory usage. Also, if we let the current value of i be 4, -1 is then an invalid index for annotation t of event B. The expression $t(B[-\mathit{1}]) < 300$ is evaluated to undef, and the whole formula can be evaluated to true if the evaluation of $t(A[\mathit{14}]) > 100$ is true, or undef otherwise.

2.4.1 Runtime Monitoring

The static trace checking technique, as described above, assumes that a simulation trace is first generated and the subsequent LOC checking parses the trace and looks for constraint violation. How the trace is generated is immaterial as long as the format is correctly specified in the definition file. The trace file for a realistic design, however, can frequently occupy several gigabytes of disk space. It may be desirable to compile the checker as a runtime monitor to run concurrently with the simulator through a Unix pipe. Alternatively, the checker can be compiled into the compiled-code simulator for higher efficiency and tighter integration. As an example of such tight integration, the checker generator has been extended to generate LOC checkers as SystemC modules [7]. During the simulation, other SystemC modules (representing the design) can pass the events and annotations directly to the monitor modules through channels. A case study of this approach is reported in Section 2.4.3. Runtime monitoring is more efficient than static checking, but then obviously the simulation need to be repeated if some new formula need to be checked later. Furthermore, the trace is no longer kept so any debugging has to rely solely on the error report.

2.4.2 Dealing with Memory Limitation

Despite the memory efficiency for most LOC formulas, some LOC formulas may require high memory usage that the verification environment cannot support. To deal with the case of preset memory limitation, another extension has been added to the checker generator. Generally, the checker tries to read the trace and store the annotations only once. However, if the preset memory limit has been reached, it stops storing the annotation and instead, scans the rest of the trace looking for needed events and annotations for evaluating the current formula instance (with the current value of i). After freeing some memory space, the algorithm resumes storing annotations and reading the trace again from the same location. The analysis time can certainly be impacted (see the case study in Section 2.4.3) and may no longer be of linear complexity. However, the verification can continue and the constraint violations can be checked under the memory limitation of the verification environment.

2.4.3 A Case Study of FIR Filter

We use a register transfer level model of a *finite impulse response (FIR)* filter written in SystemC to show how LOC can be used to efficiently check real time performance constraints. Figure 2.1 shows a 16 tap FIR filter that reads in samples when the input is valid and writes out the result when output is ready. The filter design is divided into a control FSM and a data path. The test bench feeds sampled data of arbitrary length, and the output is displayed with the simulator.

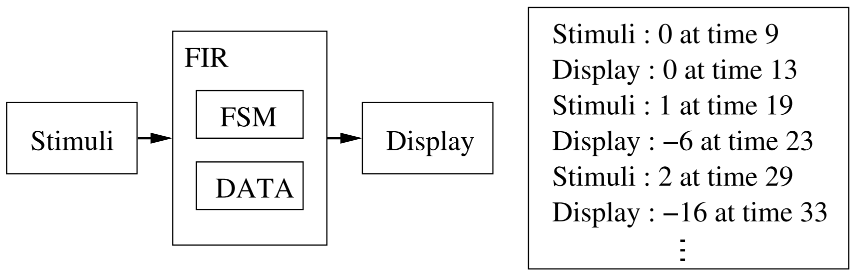


Figure 2.1: *A FIR design and its simulation trace*

We use our automatic trace checker generator to verify the properties specified in constraints (2.1) - (2.5) (in Section 2.1). The same trace files are used for all the analyses, and each constraint is checked one at a time. The time and maximum memory usage are shown in Table 2.1. We can see that the time required for analysis grows linearly with the size of the trace file, and the maximum memory requirement is formula dependent but stays fairly constant. Using LOC for common real-time constraint verification is indeed very efficient.

Given the large file size, runtime monitoring (see Section 2.4.1) may reduce the total verification time (simulation and checking) since no trace file needs to be actually generated. For

Table 2.1: *Costs of checking formulas (2.1)-(2.5) on FIR*

Lines of Trace		10^5	10^6	10^7	10^8
C1	Time(s)	1	8	89	794
	Memory	28B	28B	28B	28B
C2	Time(s)	1	12	120	1229
	Memory	28B	28B	28B	28B
C3	Time(s)	1	7	80	799
	Memory	24B	24B	24B	24B
C4	Time(s)	1	7	77	803
	Memory	0.4KB	0.4KB	0.4KB	0.4KB
C5	Time(s)	1	7	79	810
	Memory	4KB	4KB	4KB	4KB

the latency constraint (the formula (2.2)), we implement the checker as a SystemC module, and the simulation trace is no longer written to a file but passed to the monitoring module directly. Table 2.2 lists CPU times used for simulation, trace checking, and simulation with runtime monitoring for the formula (2.2) on the traces of different lengths. For the trace size of 100 million lines, the static checking approach requires 1404 seconds of simulation time and 1229 seconds of checking time for a total of 2633 seconds. Runtime monitoring requires only 1420 seconds for both simulation and monitoring. If a simulation trace is really long (e.g. hundreds of gigabytes), runtime monitoring can significantly save CPU time compared to off-line trace checking.

Table 2.2: *Time usage of simulation and checking formula (2.2) on FIR*

Lines of Trace	10^5	10^6	10^7	10^8
Simulation w/o Runtime Monitoring (s)	1	14	148	1404
Static Trace Checking Only (s)	1	12	120	1229
Simulation w/ Runtime Monitoring (s)	2	14	145	1420

We also verify constraint (2.7) to illustrate verification with memory limitation since this constraint is particularly expensive to check in terms of memory usage. Table 2.3 shows that the simulation time grows linearly with the size of the trace file. However, due to the use of an annotation in an index expression, memory can no longer be recycled and we see that it also grows linearly with the size of the trace file. Indeed, since we will not know what annotation will be needed in the future, we can never remove any information from the queue. If the memory is a limiting factor in the simulation environment, the analysis speed must be sacrificed to allow the verification to continue, as discussed in Section 2.4.2. The result is shown in Table 2.3 where the memory usage is limited to 50KB. We see that the analysis takes more time when the memory limit has been reached. Information about trace pattern can be used to dramatically reduce the running time under memory constraints. Aggressive memory minimization techniques and data structures can also be used to further reduce time and memory requirements. For most LOC formulas and simulation traces, however, the memory space can be recycled and the memory requirements are small.

Table 2.3: *Costs of checking constraint (2.7) on FIR*

Lines of Trace ($\times 10^4$)		2	3	4	5
Unlimited Memory	Time(s)	<1	<1	<1	1
	Mem(KB)	40	60	80	100
Mem Limit (50KB)	Time(s)	<1	61	656	1869
	Mem(KB)	40	50	50	50

2.5 Formal Verification of LOC Formulas

Although our trace analysis enables efficient verification of LOC formulas in a simulation environment, formal verification may still be valuable and sometimes even necessary. We propose to apply formal verification to small designs that are re-used many times, such as library modules. Because they are small, formal verification is practically possible. On the other hand, they are intended to be used in many environment, some of which will be designed long after the module itself is designed and verified. Therefore, it is hard to imagine all simulation scenarios that need to be verified. It is better to characterize the modules with a set of constraints that it satisfies. This will not only increase the confidence in the correctness, but these constraints can be used as a precise specification of a design's behavior as well. The lack of such a specification is a major source of design errors, because informal specifications of library modules are often ambiguous and misunderstood.

Unfortunately, it is undecidable whether a system satisfies an LOC formula, even if some strong restrictions are placed on the system specification and the formula (see Section 2.6). On the positive side, for a significant subset of LOC, it is possible to decide whether a finite-state system satisfies an LOC formula. The decision procedure is based on constructing a formula of Presburger arithmetic that is satisfied if and only if the formula is violated by some behavior of the system. The LOC subset that can be verified in this way includes all formulas described in Section 2.1, except the latency constraint (2.7).

Manipulating Presburger formulas is very expensive in practice, so we propose an alternative formal verification approach based on existing finite-state model checking tools. Our approach represents a complete verification procedure for a subset of LOC that defines ω-regular properties. We will show in the next section that rate (2.1), throughput (2.4), and burstiness (2.5) belong to this subset, but other formulas in Section 2.1 do not. The proposed approach may still be applied to these formulas, but the procedure is incomplete in this case, because it can terminate with an inconclusive result.

The simulation approach described in Section 2.4 suggests our formal verification approach. A trace checker can be interpreted as an automaton accepting executions. We could thus use existing model-checking tools to verify that each execution of the system is accepted by the trace checker. In the example shown in this chapter, the translation was manual. However, there is no technical difficulty in automatically generating such descriptions in a language understood by a model checking tool through modifying our trace checker generator.

The only significant difference between a simulation trace checker and an automaton description suitable for model checking is that the former can rely on dynamic memory allocation to store trace data that may be needed, while the latter must have all memory space statically allocated. Unfortunately, as we have shown in Section 2.3, for some LOC formulas it is not possible to determine memory requirements *a priori*. Our approach is to fix the memory size anyway and to designate special states where checking the formula would require allocating additional memory, but none is available. Such a state may or may not be reached during the reachability analysis. If it is, the result of the formula verification is inconclusive. More precisely, the verification of an LOC formula can have one of three outcomes:

- a counter-example is found showing that the system does not satisfy the constraint,
- the constraint is satisfied, all reachable state are searched without finding a counter-example, or reaching a state where memory is exhausted,
- inconclusive, analysis finds no counter-examples, but states where memory is exhausted are reachable.

For example, the latency constraint:

$$t(Display[i]) - t(Stimuli[i]) \leq 25 \tag{2.13}$$

cannot be modeled by any finite automata because there can be arbitrarily many occurrences of $Stimuli$ before x-th occurrence of $Display$ (intuitively, we assume that $Display[x]$ always occurs after $Stimuli[x]$). However, if we limit the number of stored time stamps of $Stimuli$ to, say, 50, then we can simultaneously check the following two constraints:

P1: There are never more than 50 occurrences of $Stimuli$ between x-th occurrences of $Stimuli$ and $Display$.

P2: If **P1** holds, then (2.13) holds.

Obviously, if **P1** and **P2** both hold, then so does (2.13), and if **P2** is $false$, so is (2.13). However, if **P2** holds, but **P1** does not, the result is inconclusive.

To specify **P1** and **P2**, assume that the trace checker keeps 51 most recent time stamps for $Stimuli$ and $Display$ in arrays $Display\_t$ and $Stimuli\_t$ such that x-th time stamp is stored at position $(x \bmod 51)$ of the array. Also assume that variable $Display\_i$ and $Stimuli\_i$ (which take values from 0 to 50) keep the index of the most recent time stamps in the arrays. Finally, assume that binary variables $Display\_occur$ and $Stimuli\_occur$ are $true$ when $Display$ and $Stimuli$ occur, respectively, and that integer variable $diff$ counts the difference between the numbers of occurrences of the $Stimuli$ and $Display$ events, i.e. it is initialized to 0, incremented on each $Stimuli\_occur$, and decremented on each $Display\_occur$. Then, **P1** can be specified with the following state predicate:

$$diff \leq 51 \ . \tag{2.14}$$

Constraint (2.13) can be expressed as follows:

$$Display\_occur \implies Display\_t[Display\_i] - Stimuli\_t[Display\_i] \leq 25 \ , \tag{2.15}$$

and finally **P2** can be expressed as follows:

$$Assumption\ (2.14) \Longrightarrow Formula\ (2.15) \ . \tag{2.16}$$

2.6 Complexity of Verifying LOC Formulas

In this section, we address the following fundamental question: How hard is it to check if a system satisfies an LOC formula? This question has many versions, depending on how the system is represented, and what subset of LOC formulas is being considered. We present answers for several versions. Some versions of the problem are undecidable, and some are decidable, but with very complex algorithms. These “negative” results are used to justify the development of efficient algorithms which may not always give the full answer. These algorithms, based either on simulation, or partial formal verification, are described in previous sections.

In the most general case, systems are represented by arbitrary programs, and annotations can be of any type. This case is clearly expressive enough to encode the halting problem [44], so checking LOC formulas is undecidable in this case.

The first restriction we consider is to limit system specification to a *infinitely-valued* finite-state system, where the number of states of a system is finite, but value domains of annotations can be infinite. Unfortunately, this case is also undecidable. To show this we can encode two counter machines using a finite-state system, two integer annotations to represent counters, and an LOC formula to ensure that counters are incremented or decremented as necessary.

The next restriction we consider are so-called *finitely-valued* finite-state systems, where annotations and event values are required to be finitely valued. With regards to annotation specification, three cases will be considered:

(1) annotations completely undefined,

(2) annotation must satisfy certain axioms, expressed by an LOC formula,

(3) annotations defined by a finite state system.

The third case is typical of later design stages. At that point annotations can be considered as part of event values, so we will not study it separately.

The first case is typical at the beginning of the design process, where constraints on annotations are stated, but nothing is yet known about their actual values. At that point, annotations are uninterpreted functions, but they still have to satisfy constraints of equalities. For example, the formula $\overline{f(e[3i]) = f(e[i+2])}$ is not satisfied by any behavior in which e occurs at least 3 times.

We consider the second case because, even if the values of annotations are not known, some constraints, captured by axioms, may be. Consider, for example, time annotations. All possible timing annotations share certain constraints, e.g. time can never decrease. Just from these basic constraints of time, we could deduce some system constraints, which are then valid for any timing. Therefore, it is useful to be able to express constraints that all annotations of certain type must have. Specifying axioms could be done in many ways. For example, an extended version of LOC is used for this purpose in Metropolis [18]. However, the following results state that checking an LOC formula is undecidable even if annotation axioms are restricted to the basic LOC.

Theorem 1 *It is undecidable whether a finitely-valued finite-state system with LOC axioms satisfies an LOC formula with a single event indexed by expression i.*

As usual, the proof proceeds by reducing a known undecidable problem to LOC checking. The details are given in Appendix B.

At first glance, it may appear that checking an LOC formula ϕ for a finite state system with annotation axioms α may be reduced to checking that the system satisfies implication of ϕ by α without any axioms. Unfortunately, this approach does not work, and to see why we will for a moment make quantification over i appear explicitly in the syntax. Thus, the axioms can be written as $\forall i : \alpha$, and the formula can be written as $\forall i : \phi$. Solving the problem requires checking $(\forall i : \alpha) \Longrightarrow (\forall i : \phi)$, but LOC can only express $\forall i : (\alpha \Longrightarrow \phi)$, which is not the same. In fact, this seemingly minor restriction makes the problem decidable, as stated by Theorem 2.

We now turn our attention to the case without axioms, where annotations are either completely unconstrained or folded into event values.

Theorem 2 *It is decidable whether a finitely-valued finite-state system without annotation axioms satisfies an LOC formula, in which all index expressions are of the form $ai + b$, where a and b are integer constants, and variable i appears only in such expressions and linear inequalities.*

The proof consists of a decision algorithm. To describe the algorithm, we need some notation. An *event expression* is an LOC term of the form val$(e[\tau])$, or of the form $f(e[\tau])$, where τ is an integer-valued term, e is an event name, and f is an annotation. Note that conditions in Theorem 2 restrict τ to be a linear expression, i.e. it must be of the form $ai + b$, where a and b are constants. The *value domain* of an event expression is the set of values it can take, i.e. it is the value domain of e if the expression is of the form val$(e[\tau])$, and it is the value domain of f if the expression is of the form $f(e[\tau])$.

Given an LOC formula ϕ, we use $\mathcal{E}_\phi$ to denote the set of event expressions appearing in it. An *interpretation* of a set of event expressions is a function that assigns to each expression in the set a value from its value domain. Since Theorem 2 requires the system to be finitely-valued, there can be only finitely many distinct interpretations of $\mathcal{E}_\phi$. Given an LOC formula ϕ, and an interpretation I of $\mathcal{E}_\phi$, we use ϕ_I to denote the formula obtained from ϕ by replacing each event expression ϵ in ϕ by the value $I(\epsilon)$. We call ϕ_I an interpretation of ϕ. Note that because ϕ_I contains no event expressions, $\mathcal{V}^n_{(\beta,A)}[\![\phi_I]\!]$ actually depends only on n and must be either $true$ or $false$.

The conditions of Theorem 2 also insure that ϕ_I is a formula in *Presburger arithmetic*. Such formulas consist of linear inequalities of integer variables combined with usual Boolean connectives and quantification of variables [30]. Presburger formulas can be evaluated to $true$ or $false$ by choosing values for all free integer variables. LOC formula interpretations can have only i as a free variable, and we will use $\phi_I(n)$ to denote the value of ϕ_I when i is set to n.

Assume, for example, a system with two binary events, x_1 and x_2, and a formula ϕ:

$$(\text{val}(x_1[3i]) = \text{val}(x_2[i])) \Longrightarrow (i \geq 5) \ . \tag{2.17}$$

It has two binary event expressions, val$(x_1[3i])$ and val$(x_2[i])$, hence it has four interpretations. To denote interpretations, we use 00, 01, 10, and 11, where the first number represent the value of val$(x_1[3i])$, an the second number represents the value of val$(x_2[i])$. It is easy to check that $\phi_{01} = \phi_{10} = true$ and $\phi_{00} = \phi_{11} = (i \geq 5)$.

It is not hard to check that LOC formula interpretations have the following property:

$$\left(\forall \epsilon \in \mathcal{E}_\phi : \mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!] = I(\epsilon)\right) \Longrightarrow \left(\mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathcal{V}^n_{(\beta,A)}[\![\phi_I]\!] = \phi_I(n)\right) \quad . \tag{2.18}$$

In words, if behavior (β, A) and integer n agree with interpretation I on the values of all event expressions, then they agree also on the value of the whole formula. In addition, formula ϕ_I is both a Presburger formula (because it has no events nor indexing) and an LOC formula (because it has no quantifiers and its only free variable is i), so it may be evaluated in both ways, but the two values are always the same.

To check whether a system satisfies an LOC formula, we will combine formula interpretations with Presburger formulas characterizing the system, and we will reduce the original problem to checking satisfiability of the combined formula. That will complete the proof, as there are known algorithms to check satisfiability of a Presburger formula. In the following Lemma, we establish that it is indeed possible to construct a Presburger formula characterizing a finitely-valued finite-state system. The construction is described in Appendix B.

Lemma 1 *For a given finitely-valued finite-state system with no annotation axioms, and a given LOC formula ϕ, it is possible to construct, for each interpretation I of $\mathcal{E}_\phi$, a Presburger formula SYS_I in which i is the only free variable, such that for all integers n, $SYS_I(n)$ is true if and only if there exists an annotated behavior (β, A) of the system such that $\mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!] = I(\epsilon)$ for all $\epsilon \in \mathcal{E}_\phi$.*

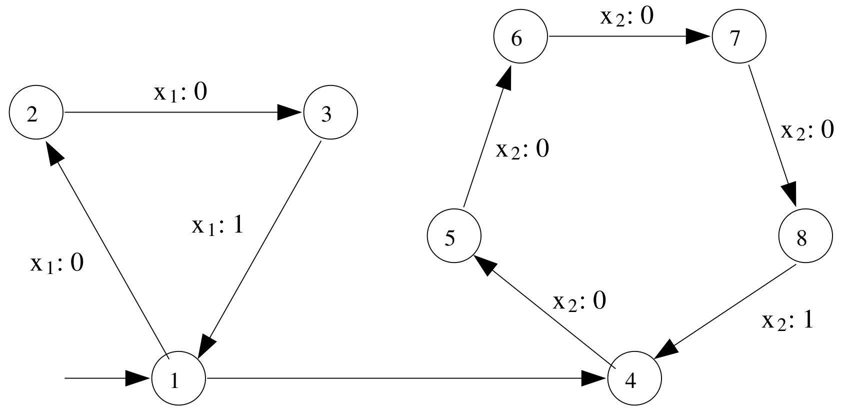


Figure 2.2: *A system generating 1 for every third value of x_1 and every fifth value of x_2*

Consider, for example, the system shown in Figure 2.2. It has eight states, two binary valued events, x_1 and x_2, and no annotations. A transition label of the form $x_k : v$ indicates that x_k is generated with value v on that transition. The system in Figure 2.2 satisfies formula (2.17), because $x_1[3i]$ is always 1, and $x_2[i]$ is 0 for all $i < 5$. With respect to interpretations of (2.17), one can easily verify that $SYS_{00} = SYS_{01} = false$, because $x_1[3i]$ is never 0, and SYS_{11} and SYS_{10} are[3] $(\exists j > 0 : i = 5j)$ and $(i > 0) \wedge \overline{(\exists j : i = 5j)}$ respectively, because every fifth values of $x_2[i]$ is 1.

Theorem 3 *For a given finitely-valued finite-state system with no annotation axioms, and a given LOC formula ϕ, let formulas SYS_I satisfy the property from Lemma 1, for each interpretation I of $\mathcal{E}_\phi$. The system satisfies ϕ if and only if the the following Presburger formula is* not *satisfiable:*

$$\bigvee_I SYS_I \wedge \overline{\phi}_I \ , \tag{2.19}$$

where the finite disjunction ranges over all interpretations of $\mathcal{E}_\phi$.

[3]We use $\exists j > 0 : \phi$ to abbreviate $\exists j : (j > 0) \wedge \phi$.

To show one direction, assume that the system does not satisfy the property, i.e. assume that there exists an annotated behavior (β, A), and an integer n such that, $\mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = false$, or equivalently $\mathcal{V}^n_{(\beta,A)}[\![\overline{\phi}]\!] = true$. Let I be the interpretation induced by (β, A) and n, i.e. set $I(\epsilon)$ to $\mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!]$ for all $\epsilon \in \mathcal{E}_\phi$. By Lemma 1, $SYS_I(n)$ is true, and by (2.18) so is $I(\overline{\phi})(n)$, so the formula is satisfiable.

For the other direction, assume that the formula is satisfiable, and let I and n be such that both $SYS_I(n)$ and $I(\overline{\phi})(n)$ are true. By Lemma 1, there exists an annotated behavior (β, A) such that $\mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!] = I(\epsilon)$ for all $\epsilon \in \mathcal{E}_\phi$, and by (2.18) $\mathcal{V}^n_{(\beta,A)}[\![\overline{\phi}]\!] = I(\overline{\phi})(n) = true$, implying that $\mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = false$, i.e. the system does not satisfy the property.

For example, the negation formula (2.17) has the following interpretations: $\overline{\phi}_{01} = \overline{\phi}_{10} = false$ and $\phi_{00} = \phi_{11} = (i < 5)$, so for the system is Figure 2.2, formula (2.19) becomes

$$(\exists j > 0 : i = 5j) \wedge (i < 5) \;\; ,$$

which is clearly not satisfiable.

Theorem 3 provides a constructive way of reducing the original problem to satisfiability of a Presburger formula. Theorem 2 then follows as a simple corollary. The described algorithm proves decidability, but it has a very high complexity. The number of interpretations may be exponential in the size of formula, and the best known algorithm for checking satisfiability of Presburger formulas is doubly exponential in the worst case. There may be cases in practice that are much better than the worst case, but it is still unlikely that the proposed algorithm will have a wide-spread use. It is therefore reasonable to search for alternative, more efficient verification algorithms, applicable to some reasonable subset of LOC. In Section 2.5, a couple of approaches along these lines has been proposed. But here we show that several approaches that one may consider are in fact not feasible.

Each LOC formula defines a language consisting of annotated behaviors that it satisfies. If we could construct an automaton with the same language, we could reduce LOC verification to the language containment problem, which has known algorithms linear in the number of states of the system and the property automaton. Indeed, this approach is possible for a very limited subset of LOC (as shown in Section 2.5), but languages of many simple LOC formulas cannot be represented by a finite-state automaton. Here are a few example:

- two events, all index expression just i, e.g.

$$\text{val}(x[i]) = \text{val}(y[i]) \ \ ,$$

- a single event, all index expressions linear, e.g.

$$\text{val}(x[i]) = \text{val}(x[2i]) \ \ ,$$

- a single event, and a single event expression, e.g.

$$\text{val}(x[i^2]) = 1 \ \ .$$

In the examples above we assume all events to be finitely valued. Still, it is not hard to show, using the pumping lemma for regular sets [44], that none of the formulas above define a regular language. Note that first two examples satisfy the conditions of Theorem 2 and could be checked with the proposed algorithm.

Another approach might be to use a class of automata that is more expressive than finite-state ones. For example one may consider pushdown automata that can define context-free languages. Unfortunately, this is not possible in general, either. For example, if event x takes values from $\{0, 1, 2, 3\}$, the formula:

$$\begin{aligned}
&(\text{val}(x[i]) = 0 \implies (\text{val}(x[i+1]) = 0 \vee \text{val}(x[i+1]) = 1))\wedge \\
&(\text{val}(x[i]) = 1 \implies (\text{val}(x[i+1]) = 1 \vee \text{val}(x[i+1]) = 2))\wedge \\
&(\text{val}(x[i]) = 2 \implies (\text{val}(x[i+1]) = 2 \vee \text{val}(x[i+1]) = 3))\wedge \\
&(\text{val}(x[i]) = 3 \implies (\text{val}(x[i+1]) = 3))\wedge \\
&((\text{val}(x[i-1]) = 0 \wedge \text{val}(x[i]) = 1) \implies \\
&\quad (\text{val}(x[2i-1]) = 1 \wedge \text{val}(x[2i]) = 2 \wedge \\
&\quad \text{val}(x[3i-1]) = 2 \wedge \text{val}(x[3i]) = 3))
\end{aligned}$$

defines the language:

$$\{s : s \text{ is a prefix of } 0^n 1^n 2^n 3^* \text{ for some } n \geq 0\} \ ,$$

for which it is easy to show that it is not context-free (e.g. see Example 6.1 in [44]).

One approach to generating an automaton for an LOC formula is to buffer event values. Once all the values needed to evaluate the formula for a particular value of i are in the buffer, the formula can be evaluated for that value of i. Once all values of i that need a particular event value are evaluated, the event value can be removed from the buffer. The results above indicate that the buffer sizes cannot be bounded in general. However, one may hope that for a specific finite-state system, a suitable bound can be found. Ideally, a bound may be found for any finite-state system.

For example, any implementation of a FIFO queue needs to satisfy the data consistency property (2.6) , i.e. the i-th value retrieved from the FIFO must match the i-the value put into it. Clearly, we cannot represent this property with a finite-state automaton, as we cannot bound in general the difference between the number of $input$ events and the number of $output$ events. However, for any particular FIFO implementation, this bound can be easily established, it is just the size of the FIFO. Thus, the size of the buffer in the checking automaton need not be bigger than the size of the FIFO. One may hope that this reasoning generalizes to any similar property and any finitely-valued finite-state system.

To the best of our knowledge, it is not known whether a bound on buffers can be found for any finite-state systems. However, we will use an example to show that even if such a bound can be found, it will sometimes be too big for an efficient verification algorithm. In general, the example is a finitely-valued finite-state system that may generate n different binary events $x_1, \ldots, x_n$, and has $p_1 + \cdots + p_n$ states, where $p_1, \ldots, p_n$ are first n primes. The system has n loops, and the k-th loop has p_k states. The system first circles through the first p_1 states, generating x_1 with value 0 $p_1 - 1$ times followed by generating x_1 with value 1 once. At the end of the loop there is a choice of repeating it or moving to the next loop. The system in Figure 2.2 is actually a part of such a system for $p_2 = 3$ and $p_3 = 5$. The language generated the system with n loops consists of all prefixes of strings defined by regular expression:

$$(x_1 : 0^{p_1-1}\ x_1 : 1)^+\ (x_2 : 0^{p_2-1}\ x_2 : 1)^+\ \ldots (x_n : 0^{p_n-1}\ x_n : 1)^+ \ .$$

Now, consider the LOC formula $\overline{\text{val}(x_1[i]) = \text{val}(x_2[i]) = \cdots = \text{val}(x_n[i]) = 1}$. (For readability and conciseness, we abbreviate formulas of the type $\tau_1 = \tau_2 \ \wedge \ \tau_2 = \tau_3$ to $\tau_1 = \tau_2 = \tau_3$.) It is not satisfied, but the smallest value of i that violates it is $p_1 * p_2 * \cdots * p_n$. Since the system generates all x_1's before generating any other events, all $p_1 * p_2 * \cdots * p_n$ values of x_1 (and $x_2, \ldots, x_{n-1}$ for that matter) would have to be buffered. Therefore, the size of the buffer have to be at least exponential in the number of states of the checked automaton, implying that the number of states of the checking automaton has to be at least doubly exponential. More practical approaches are needed.

Chapter 3

Simulation Verification and Analysis Based on Formal Assertions

Simulation is the primary verification method at all the stages of the design flow, from the system level down to the transistor level. With formal specification of design constraints (i.e. assertions), designers are allowed to precisely specify what they want to check or analyze for a design in verification, and the simulation verification and analysis process can therefore be automated. This chapter focuses on assertion-based simulation verification.

3.1 Methodology of Simulation Verification and Analysis

Figure 3.1 illustrates the methodology of the simulation verification based on formal assertions with automatic trace checker generation. Designers are responsible for the specification of design constraints with certain formal languages such as LTL and LOC. Automatic tools

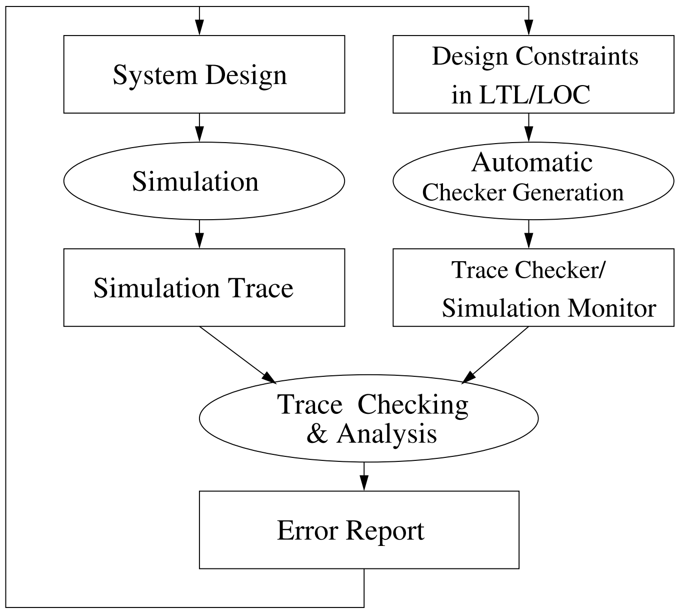

System Design
Design Constraints in LTL/LOC
Simulation
Automatic Checker Generation
Simulation Trace
Trace Checker/ Simulation Monitor
Trace Checking & Analysis
Error Report

Figure 3.1: *Simulation verification and analysis based on formal assertions*

are utilized to generate simulation monitors or static checkers for trace analysis. Simulation traces can then be checked during or after simulation, and design errors are reported if there is any constraint violation. According to an error report, designers can either correct the original design or revise the constraint specifications until the trace analysis passes the verification. In this simulation verification and analysis methodology for system level designs, the state transitions are modeled as event occurrences. This is consistent with transaction abstraction since only the events ordering are considered, not their tick-by-tick cycle level behavior. [1]

As shown in Section 2.3, LTL and LOC have different domains of expressiveness and indeed complement each other quite well. According to our experience, most functional constraints, such as mutual exclusion, non-starvation, and safety, can be easily expressed

[1]To handle cycle level analysis, designers only need to output clock ticks as events.

with LTL. On the other hand, LOC is more suitable for expressing quantitative performance constraints such as rate and latency, and transaction level functional constraints such as I/O data consistency. In this study, the formal specification of design constraints are mainly based on these two logics. We leverage an existing tool FoCs [14] to generate checker cores for LTL formulas and then use our tool to automatically generate wrappers that are necessary for simulation monitors and stand-along trace checkers. Since simulation sessions are finite, the linear temporal operators are interpreted over finite system executions by checking the conditions only up to the end of executions. An automatic tool set [13] has been developed to generate trace checkers and simulation monitors for given LOC formulas according to the algorithms and data structures presented in Section 2.4.

3.2 Simulation Verification in Metropolis

The assertion-based simulation verification methodology has been integrated in the Metropolis design framework. From formal specification of LOC or LTL constraints in MMM, run-time monitors or static checkers can be automatically generated along with simulation models in the integrated framework. Various functional and performance constraints can then be checked during or after simulation. In this section, two design examples are used to demonstrate the methodology implemented in Metropolis. The first is a system level design for set-top video processing, *Picture-in-Picture (PiP)*, which is originally specified with YAPI [49]. PiP is partially respecified and simulated with Metropolis. The other one is a

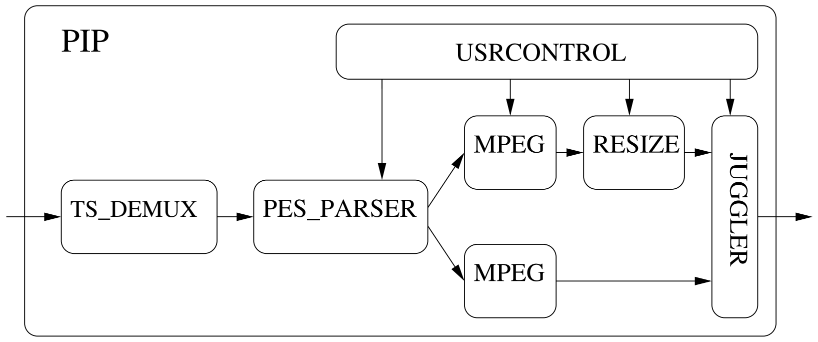


Figure 3.2: *Picture-in-Picture design*

high level model of function-architecture mapping. We use the generated trace checkers to verify a wide variety of functional and performance constraints.

3.2.1 A Picture-in-Picture Design

Figure 3.2 shows the PiP design. TS_DEMUX demultiplexes the single input transport stream (TS) into multiple packetized elementary streams (PES). PES_PARSER parses the packetized elementary streams to obtain MPEG video streams. Under the control of the user (USRCONTROL), decoded video streams can either be resized (through RESIZE) or directly feed to JUGGLER that combines the video frames to produce the picture-in-picture videos. The entire description consists of approximately 19,000 lines of Metropolis and YAPI code. With the sample input stream we used, it produced 120,000 lines of output representing header information for the processed frames.

In the transaction-level design of PiP, where time is still not available, we can check both functional and performance constraints with proper annotations output from the simulation. In the component RESIZE of PiP, the video frames processed are in interlaced format with

```
                        ⋮
WINDOW_DATA_OUT 23483 87000
WINDOW win_params_update x_begin: 12 y_begin: 6
RESIZE field_start  field_count: 2 size:  6720
                        ⋮
WINDOW win_params_update x_begin: 12 y_begin: 6
USRCONTROL write pixels_out: 144
RESIZE field_start  field_count: 3  size:  10368
                        ⋮
USRCONTROL write lines_out: 64
THSRC_CTL_OUT finfo_write value: 12876
RESIZE field_start  field_count: 4  size:  14016
                        ⋮
```

Figure 3.3: *PiP simulation trace*

alternating fields of all odd lines, then all even. The frame size should only change after a complete frame, each of which has 2 fields, is produced. Therefore, the field sizes of paired even and odd fields should be the same. This constraint can be specified as an LOC formula:

$$\begin{aligned} &size(\mathit{field\_start}[2i+2]) - size(\mathit{field\_start}[2i+1]) = \\ &\quad size(\mathit{field\_start}[2i+1]) - size(\mathit{field\_start}[2i]) \qquad , \end{aligned} \tag{3.1}$$

where $\mathit{field\_start}$ is an event, at which RESIZE starts to output a new image field. The annotation $size$ is the cumulative number of pixels processed by RESIZE. Figure 3.3 shows snapshots of the PiP trace. The generation of the checker for this LOC formula and the actual checking on the simulation trace take less than 1 minute of CPU time.

Another functional constraint we are interested in is that the number of the fields the RESIZE component reads in should be equal to the number of fields it produces. Two local counters, one at RESIZE's input part and one at its output part, provide these annotations. After a piece of video is processed, these two counters need to be compared to see if the constraint is satisfied. The LOC formula used to check this constraint is:

$$\mathit{field\_cnt}(in[i]) = \mathit{field\_cnt}(out[i]) \ . \tag{3.2}$$

The events in and out are generated by the input and output parts of RESIZE respectively whenever they finish processing a whole piece of video. The annotation $\mathit{field\_cnt}$ represents the number of fields processed by the input and output parts of RESIZE. The generation of the checker for this formula and the actual trace checking take less than 1 minute of CPU time.

We can also check performance constraints such as latency. The latency issue in RESIZE relates to the timely response to the size specification from the user. Since PiP is specified at the behavioral level, no detail timing information is available. We therefore specify a bound (e.g. 5) on the number of fields processed between reading a new size specification ($\mathit{read\_size}$) and the actual change in the output video image size ($\mathit{change\_size}$):

$$\mathit{field\_cnt}(\mathit{change\_size}[i]) - \mathit{field\_cnt}(\mathit{read\_size}[i]) \leq 5 \ , \tag{3.3}$$

where event $\mathit{read\_size}$ is generated whenever RESIZE reads a new size specification from USRCONTROL, and event $\mathit{change\_size}$ is generated whenever the size of the output video image is actually changed. The annotation $\mathit{field\_cnt}$ is the value of a global counter that is

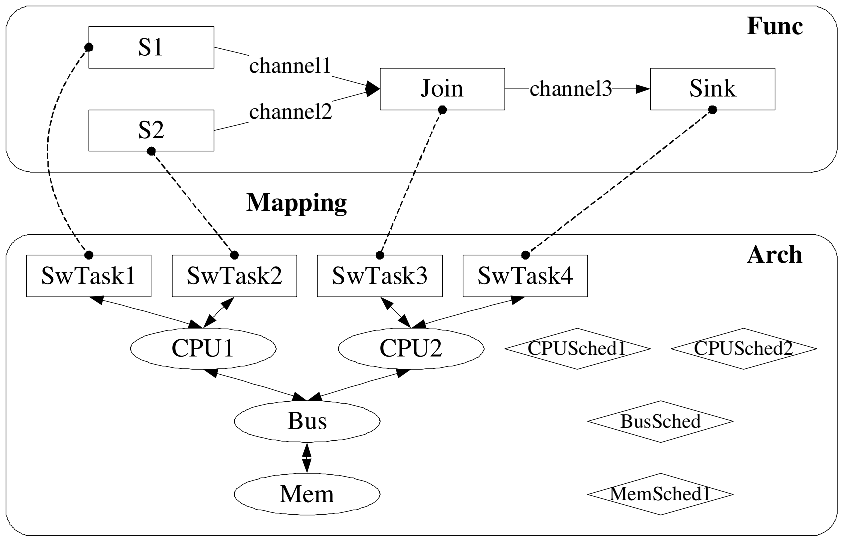


Figure 3.4: *A function-architecture mapping model*

incremented by one whenever RESIZE processes a new frame field. The generation of the checker for this LOC formula and the actual trace checking also take less than 1 minute of CPU time.

3.2.2 A Function-Architecture Mapping Model

In the platform-based design, mapping is the key procedure that correlates the function to the architecture of a design. In this design example (as shown in Figure 3.4), two source processes (S1 and S2) write data into two independent channels. A separate process (Join) then reads data items from both channels, manipulates them, and then sends the result data to another process (Sink) through another channel. In the abstract architecture model, there are two CPU/RTOS units, a bus unit, a memory unit, and a quantity manager (i.e. scheduler) for each architectural unit. [2] A CPU unit can be shared among several software tasks that may

[2]An architectural unit is modeled as a medium in Metropolis.

request services from it. When more than one service request is issued to a CPU, arbitration is needed. The mapping procedure synchronizes the processes in the function model and the mapping processes (representing software tasks) in the architecture model. In this example, functional processes S1 and S2 are mapped to mapping processes SwTask1 and SwTask2, respectively, which are associated to CPU1, and the other two processes are mapped to CPU2. The CPU quantity managers implement a non-preemptive static-priority dynamic scheduling policy. The two CPU units are connected to the bus, and the bus is connected to the memory unit. During simulation, the functional events are time-stamped through the architecture model, and thus various performance constraints can be analyzed. With the sample input we used, the simulation took 14 minutes and produced a 1.1G trace file with 2.36$\times 10^7$ lines.

We analyze the throughput of the model by using an LOC formula:

$$time(Sink\_read[i+100]) - time(Sink\_read[i]) \leq 5.0 \times 10^{-5} \ , \tag{3.4}$$

where event $Sink\_read$ represents the read operation by process Sink. The formula passes the trace verification in less than one minute, which means process Sink can perform at least 100 read operations in every time period of 5.0 ns.

Similarly, we can check the latency between the source processes and process Sink by checking their events representing write and read operations respectively:

$$time(Sink\_read[i]) - time(S1\_write[i]) \leq 1.5 \times 10^{-7} \ , \tag{3.5}$$

and

$$time(Sink\_read[i]) - time(S2\_write[i]) \leq 1.5 \times 10^{-7} \ . \qquad (3.6)$$

We can also analyze the processing delay of process Join using the formula:

$$time(Join\_write[i]) - time(Join\_read[i]) \leq 5.0 \times 10^{-7} \ . \qquad (3.7)$$

It should be emphasized that timing is only one of the possible annotations we can use in LOC to analyze quantitative constraints of a design. Any values associated with events can be used as annotations to check corresponding constraints (e.g. data value or power).

In addition, LTL formulas can be used to verify temporal constraints of the events generated by different processes (e.g. the event order). For example, the constraint that process Join cannot read before both source processes write, and process Sink cannot read before process Join writes can be verified with the formula:

$$\mathsf{G}\ ((\neg Join\_read\ \mathsf{U}\ (S1\_write\ \wedge\ S2\_write)) \wedge (\neg Sink\_read\ \mathsf{U}\ Join\_write)) \ . \qquad (3.8)$$

Given the trace size, all these constraints formulas can be analyzed within one minute.

3.3 Verification for Network Processor Architectures

We also integrate our assertion verification methodology into the design flow for high performance network processors. Based on Intel IXP1200 [3] network processor model, in-house designers have been putting together a new architecture which is capable of higher throughput, lower latency, and lower cost. The processor model is parameterized, so that a whole range of different architectures can be explored. Using our assertion verification methodology, designers were able to write assertions and automatically generate trace checkers or simulation monitors throughout the design process to check functionality and performance characteristics. Bugs were subsequently found and corrected.

3.3.1 Introduction to Network Processors

As Internet gets more and more complicated with the rise of new protocols and services, so does the cost of new equipment and upgrades. A network processor (NPU) is a base hardware platform that provides high performance and flexible programming capabilities, which allows it to address many market segments and a wide range of applications. As a result, the cost of upgrade can be reduced and developing cycles for new protocols and data types can be shortened. Therefore, NPUs are poised to replace expensive and inflexible fixed-function silicon application-specific integrated circuits (ASICs).

A number of challenges for NPU implementation are already evident. Performance and power dissipation are the most important among them. While high performance is achieved by increasing the working frequency and the degree of parallelism, power dissipation has been also increased significantly. For example, in a typical router configuration, there may be one or two NPUs per line card. A group of line cards, e.g. 16 or 32, are generally placed within a single rack or cabinet. Thus, the aggregated heat dissipation becomes a big concern, given that each NPU typically consumes around 20 Watts and the operating temperature can reach as high as 70^oC [10]. On the other hand, with the demand of performance scaling, NPU's clock frequency is increasing and more computation engines will be put on an NPU. Table 3.1 shows the power and performance changes in three Intel IXP family NPUs [3,9,10]. Note that the power dissipation increases as the complexity of NPU increases. This trend brings significant challenges for the NPU design.

System level modeling with executable languages such as C/C++ or other modeling frameworks have been crucial in designing large electronic systems. Unfortunately, most cycle-level accurate simulators only report performance and power data for worst and/or average cases. These data pose limitation on power/performance analysis. For example, an NPU's performance and power dissipation are closely related to the workload, namely the incoming packet rate. The workload is usually unbalanced, in temporal scale or geographical scale, which may cause extreme high power dissipation occasionally. The unbalanced workload provides opportunities for power and performance tuning. Therefore, the power and performance distribution patterns are important complements to average/worst-case data in the design exploration.

Table 3.1: *Power and performance of Intel IXP NPUs*

Description	IXP1200	IXP2400	IXP2800
Performance(MIPS)	1200	4800	23000
Media Bandwidth(Gbps)	1	2.4	10
Frequency of ME(MHz)	232	600	1400
Number of MEs	6	8	16
Power(W)	4.5	10	14

The methodology proposed in Section 3.1 is implemented for verifying and analyzing basic functional and performance constraints of an NPU design. It will be shown that the assertion-based analysis methodology is also very suitable for transaction-level or cycle-level design exploration, specially in power and performance analysis for NPU designs. From formally specified assertions, trace checkers and distribution analyzers are automatically generated to validate or analyze simulation traces. Designers do not need to write separate reference models or scripts to scan through the traces. So it is very efficient in design exploration for large systems with high complexity and functionality such as NPU designs.

3.3.2 Network Processor Model

Intel IXP1200 [3] is chosen as the reference model due to its overwhelming popularity in the network processing applications. Given normal-size IP packets, it can achieve up to 2.2 Gbps routing bandwidth. Being sold commercially, the processor model has been made available to the public in order to help the designers build systems based on IXP1200. The basic architecture of the processor is shown in Figure 3.5. IXP1200 consists of a StrongARM core, 6 multi-threaded processing units, which are called microengines, and controllers of peripheral units. The StrongARM core initializes the program store of the microengines and loads necessary data into memory before enabling the microengines. Each of the six microengines runs up to four threads concurrently. Thus, a total of up to 24 threads can be programmed to receive, process and transmit IP packets. The controllers of SRAM, SDRAM and IX Bus units serve the processor as interfaces to off-chip SRAM, typically used to store

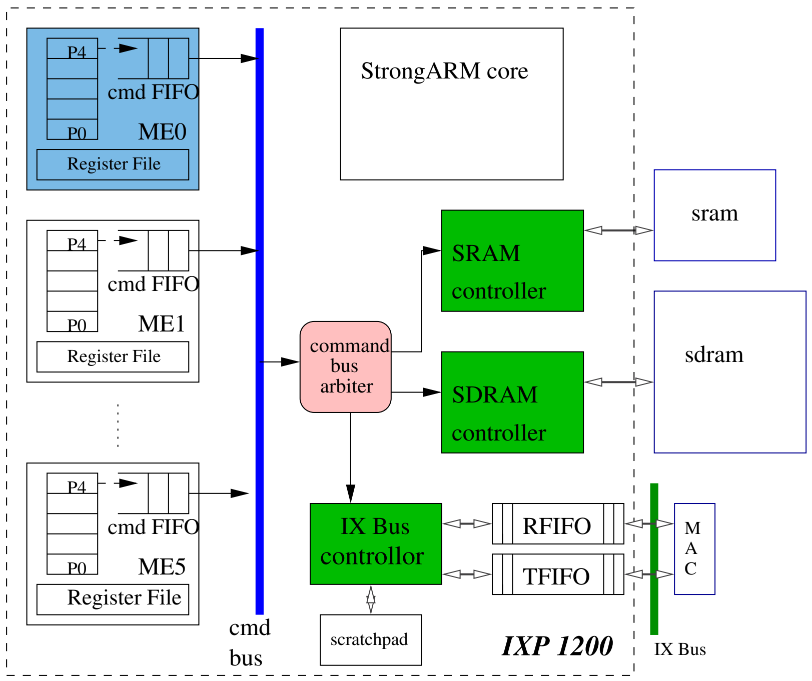


Figure 3.5: *IXP1200 architecture*

forwarding table, SDRAM, typically used to store IP packets, and network devices through the IX Bus.

The threads in a microengine share common ALU, pipelining, and scheduling units. Inside a microengine, each thread has an independent set of registers including general purpose registers, local control registers, SRAM transfer registers and SDRAM transfer registers. The microengine's instruction set architecture contains 33 categories of instructions. Because each instruction may have a number of different operations, the total number of op-codes implemented is around 120. For example, the $sram$ instruction has operations such as $read$, $write$, $push$ and pop, each of which corresponds to a unique op-code. Each microengine has a 5-stage pipeline (P0 through P4): Instruction Fetch, Instruction Decoding, Operand Fetch, Instruction Execution, and Write-Back.

In each microengine, memory references, which are called commands, are issued to a two-entry command FIFO. The commands are then sent to the command bus and scheduled by the command bus arbiter. Based on the priority of commands, the command bus arbiter selects one or more reference commands among the six command FIFOs and move them to the corresponding memory controllers. The SRAM controller handles all SRAM reference commands issued from the microengines. Each SRAM reference command is enqueued, dequeued, committed and finally done. SDRAM reference commands are handled similarly by the SDRAM controller.

In this study, the NePSim simulator [52] is used to model NPU architectures. NePSim is based on Intel IXP1200 and includes a cycle-accurate architecture simulator and a power estimator. All the configurations in NePSim are parameterizable. When the architectural parameters (e.g. number of microengines, number of threads in each microengine, number and length of FIFO queues, size of the caches, scheduling policies) are set to be that of the IXP1200, the behaviors of the two processor models are expected to be very similar. When we vary the parameters, the functional "correctness" is expected to be maintained while the performance attributes are expected to change, trading off one metric against another. Ultimately, we can arrive at a design that is most suitable to the applications at hand.

3.3.3 Experimental Settings

In order to verify the processor architectural models, a set of architectural execution events that occur during simulation are generated as simulation traces. They include instructions en-

Table 3.2: *List of events for verification*

Event Type	Comments
$pipeline$	an instruction enters the pipeline
$sram\_enq$	an SRAM access request is enqueued
$sram\_deq$	an SRAM access request is dequeued
$sram\_done$	an SRAM access request is committed
$sdram\_enq$	an SDRAM access request is enqueued
$sdram\_deq$	an SDRAM access request is dequeued
$sdram\_done$	an SDRAM access request is committed
bus_issued	a bus request is issued
bus_done	a bus request is committed
ip_lookup_start	an IP address lookup starts
ip_lookup_done	an IP address lookup finishes
$receive$	an IP packet is received
$forward$	an IP packet is forwarded

tering or leaving the pipeline, memory reference commands being put into or removed from the command queues in memory controllers, signals being generated from or consumed by functional units and threads. Table 3.3.3 lists the events that we are interested in for the verification studies. To differentiate events generated by different microengine, different threads, and different architecture models, each event could be appropriately prefixed and suffixed. For example, $m2\_t1\_pipeline\_IXP$ represents the pipelining event from the microengine 2, thread 1 of the Intel IXP1200 model. An event is annotated with timing, identification, and other quantitative information. As presented in this section, each event instance is associated with four annotations, $cycle$, pc, $addr$, and $data$, where $cycle$ is used to measure time in clock cycles, and pc is the PC (Program Counter) value for the current instruction. Depending on the event, $addr$ may represent memory address or next PC (NPC) address, and $data$ may represent data read from memory or ALU operation result.

```
                            ⋮
1731  68  00120100  00000000  m2_t2_pipeline
1732  34  0000FFF8  00000000  m5_t1_receive
1733  19  00000300  00000000  m0_t0_sram_enq
1733  30  00000300  00000000  m1_t0_pipeline
1733  30  00000300  00000000  m2_t0_pipeline
1733  30  00000300  00000000  m3_t0_pipeline
1733  20  00000300  00000000  m0_t0_sram_deq
1734  69  0000003F  0000050C  m2_t2_pipeline
1735  34  00000300  00000518  m5_t2_pipeline
1736  35  0000001B  0000051C  m5_t2_forward
1736  36  00178000  00000520  m5_t2_pipeline
1737  72  00020100  00000524  m2_t2_pipeline
1737  20  00000300  00000000  m0_t0_sram_done
                            ⋮
```

Figure 3.6: *NePSim simulation trace*

Traces from NePSim are taken “as is” and fed into automatically generated trace checkers. Log traces from Intel IXP1200 are preprocessed in a straightforward manner before being fed into trace checkers so that each event has the above annotations on the same line, though there is no difficulty in writing a separate trace format for IXP1200. A snapshot of the simulation trace from NePSim is shown in Figure 3.6.

In the following verification studies, we run a trie-based route lookup benchmark [60] on NePSim and Intel IXP1200 processor models. The benchmark program is an infinite loop which continuously look up the route for a list of IP addresses. The trie data structure is stored in SRAM and accessed through the SRAM instructions.

3.3.4 Verification Studies

We present three categories of assertion verification throughout the design process in this section. First, we would like to know whether NePSim, with parameters equal to those of

IXP1200, would achieve the same functionality and "similar" performance. We then vary the design parameters (i.e. number of microengines, number of threads, configuration of the microengines, amount of caches, ..., etc,) and in each case, check functional correctness of the new design with functional assertions. Varying the design parameters obviously affect the performance. We use performance assertion checking to determine the performance of a particular design given a particular simulation input.

Checking with Reference Model

Using LOC, we can formally and accurately specify both functional equivalence and performance similarity of two designs. [3] We run the same benchmark on NePSim and Intel IXP1200 models, and use the simulation trace from IXP1200 as the reference trace. First, we check if the NePSim model is functionally equivalent to the reference model. The primary function of the model resides in the forwarding table lookup for IP packets being processed, which involves correct reading from the SRAM. More specifically, we want to check the following constraint:

"For each SRAM access on NePSim and IXP1200, the associated memory address and data should be the same, and all the SRAM references are executed with the same order."

This constraint can be specified with an LOC formula:

$$addr(sram\_enq[i]) = addr(sram\_enq\_IXP[i]) \wedge data(sram\_done[i]) = data(sram\_done\_IXP[i]) \tag{3.9}$$

[3]The requirements for checking functional equivalence and performance similarity are directly from the designers.

We run the benchmark on NePSim and IXP1200 for one million cycles to obtain traces of about 3×10^5 lines. Both models are configured with a single working microengine and a single working thread so that the packet processing order is deterministic. With the automatically generated trace checker, we show that this formula pass with the given benchmark trace within 6 seconds of CPU time (see Table 3.3.) All the trace checkings presented in this section were run on our Athlon 1.5GHz Linux machine with 1GB memory, though the simulation sessions were run by the designers on their own machines. We report time and memory usage only for the trace checking operations. In a preliminary version of the design, the functional equivalence checking was violated. The error report helped the designers identify a real bug in the SRAM request scheduling algorithm.

Comparing the simulation trace from NePSim to the reference IXP1200, we also want to make sure that the instruction pipelining behavior of NePSim is "similar" in performance to that of IXP1200. More specifically, this constraint requires:

"On the two models, all the instructions in the benchmark are executed with the same order, and the execution time of every pipelining instruction by NePSim is no more than certain clock cycles away from the execution time of the corresponding instruction by Intel IXP1200."

This constraint can be specified with an LOC formula:

$$\begin{gathered}(pc(pipeline[i]) = pc(pipeline\_IXP[i])) \wedge \\ (|cycle(pipeline[i]) - cycle(pipeline\_IXP[i])| \leq A \cdot i + B),\end{gathered} \tag{3.10}$$

where A and B are constants. The second part of the formula holds if and only if, for a particular pipeline event, the difference in time of occurrence in NePSim and Intel IXP1200 is within $A \cdot i + B$. As simulation progress, the difference accumulates, which is reflected by $A \cdot i$. The difference in startup time is accounted for by constant B. If the two designs are truly identical, both values should be zero. The designers decided that, to account for the differences in the two designs, the acceptable values of A and B should be 0.05, and 8, respectively. The formula failed almost immediately, and after going through the error report and debugging the NePSim design, it was found that the SRAM access latency was not modeled correctly. Once the error was fixed, the performance assertion passed (see Table 3.3). This performance margin is sufficient for designers to declare that NePSim and IXP1200 are similar in performance.

With the same formula (3.10) and substituting for the $pipeline$ event, we can check the performance similarity of other critical events such as $sram\_enq$, $sram\_done$, and $sdram\_enq$, with different acceptable values of A and B, determined by the designers.

Functional Verification

Due to the non-determinism in thread handling within the network processor models, it is difficult to perform deterministic functional equivalence trace checking when there are more than one thread enabled. For normal multi-thread operations, functional constraint verification, based on both LTL and LOC, can be very useful. Designers can write their functional assertions in LTL or LOC. For LTL assertions, we use FoCs to generate the assertion checking code in C++, and our tool then generates the necessary wrappers for trace checking.

To verify the normal operation of the NePSim processor model, We configure it with 6 working microengines (4 of them (m0 - m3) used for forwarding table lookup and 2 of them (m4, m5) used for IP packet transmission) and 4 working threads for each microengine, and run the benchmark on NePSim for one million cycles. We want to check a non-starvation constraint for the SRAM controller of NePSim:

“Once an SRAM access request from a thread (e.g. thread 0) of a microengine (e.g. microengine 0) is enqueued, it must be eventually committed within the next 300 SRAM related event occurrences.”

This constraint can be specified with an LTL formula:

$$\mathsf{G}(m0\_t0\_sram\_enq \rightarrow \mathsf{X}[1:300](m0\_t0\_sram\_done)) \ . \tag{3.11}$$

To check this constraint, we only produce the events that are related to SRAM references to get a trace of 2.8×10^5 lines. The parameterized wrapper generator can easily generate this assertion for all threads in all microengines, and for SDRAM controller and IX bus controller.

Another important constraint of the memory access scheduler is the correct occurring order of the events $sram\_enq$, $sram\_deq$, and $sram\_done$, which requires that “after an SRAM request by a thread (e.g. thread 1) of a microengine (e.g. microengine 0) is issued and put into the scheduling FIFO, it cannot be done before it is dequeued”. This constraint of occurring order can be specified with a formula:

$$\mathsf{G}(m0\_t1\_sram\_enq \rightarrow \neg\, m0\_t1\_sram\_done \,\mathsf{U}\, m0\_t1\_sram\_deq) \ . \tag{3.12}$$

Table 3.3: *Verification results for functional assertions*

Formula	Formula Instances	Trace Lines	Mem	Time
(3.9)	10267	3×10^5	40KB	6s
(3.10)	295582	3×10^5	64.8 KB	7s
(3.11)	5690	2.8×10^5	0.4KB	77s
(3.12)	5739	7.0×10^6	50 Bytes	24s
(3.13)	5708	7.0×10^6	12 Bytes	59s

Note that if the simulation trace ends, the verification of the formula will be interpreted on a finite trace. For example, formula (3.12) will not be violated if $sram\_enq$ occurs and then neither $sram\_done$ nor $sram\_deq$ occurs when the trace ends.

Using LOC, we can specify data consistency constraints for different functional units. For example, when an SRAM access request is put into to the scheduling FIFO by a thread (e.g. thread 2) of a microengine (e.g. microengine 1) and then eventually committed, the memory address it refers to should be the same. We express this constraint with the LOC formula:

$$addr(m1\_t2\_sram\_enq[i]) = addr(m1\_t2\_sram\_done[i]) \ , \tag{3.13}$$

where the annotation $addr$ is used to represent the referenced memory address. With the automatically generated trace checkers, formula (3.11) - (3.13) are checked with no error. The verification results are listed in Table 3.3.

Performance Assertions

The goal of design exploration for network processor is to find an architecture which would perform “better” than the existing model. It is therefore very important to be able to analyze quantitative constraints of a design. With LOC, we can express the performance requirements or expected quantitative features. In this section, we continue with the parameter setting of 4 microengine for IP address lookup and 2 microengines for IP packet transmission. For each microengine doing IP address lookup, we experiment with either running 2 threads or 4 threads. As a consequence, we compare the performance metrics for an 8-thread processor model against the one with 16 threads. We run our benchmark on both configurations for one million cycles, and get traces of about 3 million lines.

One primary function of the network processor is to perform IP address lookup, which requires very frequent access to SRAM. Therefore, we want to check the latency between an SRAM access request enqueued and when it is committed. We first check the SRAM access latency from a thread (e.g. thread 0) of a microengine (e.g. microengine 2) for the two configurations. We consider the maximum latency constraint, which can be expressed with the following LOC formula:

$$cycle(m2\_t0\_sram\_done[i]) - cycle(m2\_t0\_sram\_enq[i]) \leq l1 \ . \tag{3.14}$$

We iteratively search for the smallest $l1$ that will allow the traces to pass the performance assertion (e.g. with a simple bi-partition approach on the range). For the 8-thread configuration, we were able to set $l1$ = 50, and the assertion can pass the trace checking without any error. For the 16-thread configuration, in order to make the assertion pass, we have to increase the $l1$ to 100. More threads can cause more memory access contention, and degrade the latency for individual memory accesses. See Table 3.4 for a summary of the result.

The total number of running threads can actually affect the latency for individual IP address lookups. The maximum latency of IP address lookups in a thread (e.g. thread 1) of a microengine (e.g. microengine 0) can be specified with an LOC formula:

$$cycle(m0\_t1\_ip\_lookup\_start[i]) - cycle(m0\_t1\_ip\_lookup\_done[i]) \leq l2 \ . \tag{3.15}$$

For the 8-thread configuration, we set $l2$ to be 900 for the assertion to pass. For the 16-thread configuration, $l2$ needs to be 1200. Using the formula (3.15), we have explicitly shown that the 8-thread configuration has lower latency for individual IP address lookups than the 16-thread configuration.

Of course, latency does not tell the whole story. Throughput is an equally important design characteristic for network processors. More threads should achieve better overall throughput. At the instruction level, we can check the throughput of pipelining instructions for the processor using the LOC formula:

$$cycle(pipeline[i + 10000]) - cycle(pipeline[i]) \leq t1 \ , \tag{3.16}$$

which requires that within $t1$ cycles, at least 10000 instructions need to be issued to the pipeline of the processor. For the 8-thread configuration, we need to set $t1 = 4200$ for the assertion to pass. This corresponds to a minimum throughput of 2.3 instructions per cycle. For the 16-thread configuration, $t1$ need to be set to 3500, which corresponds to a minimum throughput of 2.8 instructions per cycle. The 16-thread configuration has better instruction throughput according to the analysis using the performance assertion (3.16).

The overall performance of the network processor is measured by the throughput of IP packet forwarding, which can be expressed with the following LOC formula:

$$cycle(forward[i + 1000]) - cycle(forward[i]) \leq t2 \ . \tag{3.17}$$

In order for the performance assertion to pass, We need to set $t2 = 3.7 \times 10^5$ for the 8-thread configuration, and set $t2 = 3 \times 10^5$ for the 16-thread configuration. If we assume the NePSim processor is running at 200MHz, we get the throughput for IP packet forwarding of 5.4×10^5 packets/sec and 6.6×10^5 packets/sec for the 8-thread and 16-thread configuration, respectively. Given the average packet size of 64 bytes, the routing throughput will be 2.8 Gbps and 3.3 Gbps respectively for the two configurations. Indeed, the designers need to trade off latency and throughput for any given application to achieve the best design. LOC assertion checking allows them to quantitatively analyze the performance of a system level specification. During the design process, the designers can also experiment with increasing the number of microengines, changing the size of scheduling FIFOs, or putting more caches between storage hierarchies. All these design space explorations may bring various performance trade-offs, which can be easily specified and analyzed by the formal performance assertions.

The verification results of these LOC performance assertions are listed in Table 3.4. Since a typical simulation session can take half an hour or longer, the CPU time and memory usage for the trace checkers are trivial by comparison. Without them, however, it becomes very

Table 3.4: *Verification results for performance assertions*

Formula	Configuration	Parameters	Time	Memory
(3.14)	8-thread	$l1 = 50$	18sec	12Bytes
	16-thread	$l1 = 100$	23sec	16Bytes
(3.15)	8-thread	$l2 = 900$	46sec	8Bytes
	16-thread	$l2 = 1200$	44sec	8Bytes
(3.16)	8-thread	$t1 = 4200$	20sec	40KB
	16-thread	$t1 = 3500$	26sec	40KB
(3.17)	8-thread	$t2 = 3.7 \times 10^5$	44sec	4KB
	16-thread	$t2 = 3 \times 10^5$	44sec	4KB

difficult for the designers to conclude anything about the design except in very vague terms (e.g. "looks good"). Our assertion-based verification methodology is indeed efficient for dealing with large designs.

3.4 Performance and Power Analyis for Network Processor Architectures

In this section, we focus on the assertion-based design exploration of dynamic voltage scaling techniques in NPU architecture models. In order to efficiently analyze the power-performance trade-offs among different DVS policies with different parameter settings, we use LOC to specify assertion formulas for power and performance distributions. With automatically generated distribution analyzers, we can compare their power and performance characteristics and identify optimal configurations in their large design spaces.

To automate quantitative distribution analysis that is common in design exploration, we extend the LOC assertions by introducing 3 more operators $\bowtie$, $\lhd$ and $\rhd$. To analyze the distribution of some quantity over certain ranges, we can use a formula, in the form of $quantity \bowtie \{min, max, step\}$, to automatically generate a corresponding analyzer. An *analysis period* is specified with a triple $\{$*min, max, step*$\}$, where *min* and *max* are lower and upper bounds, and the interval between these two values is divided into bins of width *step*. For example, given a formula:

$$(time(forward[i+100]) - time(forward[i])) \bowtie \{40, 80, 5\} \ , \tag{3.18}$$

an assertion analyzer is generated to evaluate the left hand side with *i* being 0, 1, 2, ... , and report the percentage of formula instances whose values fall within the ranges of ($-\infty$, 40], (40, 45], ..., (75, 80], (80, $+\infty$). If we replace the operator $\bowtie$ with $\lhd$ or $\rhd$, the ranges become ($-\infty$, 40], ($-\infty$, 45], ..., ($-\infty$, 75], ($-\infty$, 80] or [40, $+\infty$), [45, $+\infty$), ..., [75, $+\infty$), [80, $+\infty$), respectively.

3.4.1 Experimental Settings

In this set of experiments, we use the same network processor simulator NePSim described in Section 3.3 and choose four representative networking applications to explore different architectural features of the NPU model, i.e. *ipfwdr, url, nat,* and *md4*. The application *ipfwdr* is an IP forwarding software provided in Intel's SDK. The routing table is stored in the SRAM, and the output port information is stored in the SDRAM. The program *url*

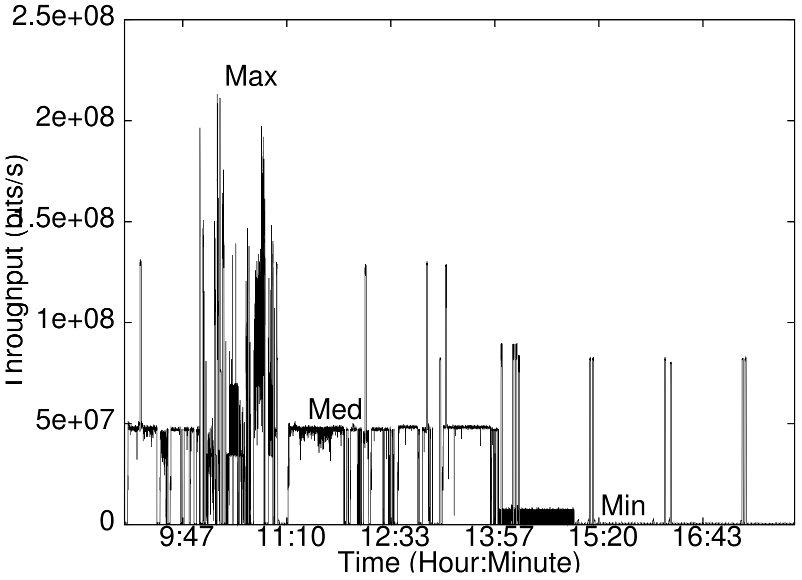


Figure 3.7: *Distribution of IP packets*

routes packets based on URL requests. It checks the payload of packets frequently, so it needs a large number of SRAM and SDRAM accesses. In *nat* (network address translation), each packet only needs an access to SRAM for looking up the IP forwarding table. The *md4* provides a 128-bit digital signature algorithm. It moves data packets from SDRAM to SRAM and accesses SRAM multiple times for computation. So it is both memory and computation intensive. Memory accesses, specially SDRAM accesses, have long latency. They lead to long idle time for MEs, which in turn shows up as lower power and throughput. Computation-intensive benchmarks, those that do not wait on memories, will tend to show higher power consumption.

The simulation data that we use follows IP traffic patterns in a real world edge router from NLANR [12]. Figure 3.7 shows a day time distribution of IP packet arriving rates. Due to the limited simulation speed of NePSim, it is too expensive to run the whole trace. We sample a

Table 3.5: *List of events and event annotations for performance and power analysis*

Events	Details
pipeline	an instruction enters the execution pipeline
forward	an IP packet is forwarded
fifo	an IP packet is put into the processing queue
Event Annotations	Details
cycle	number of core clock cycles elapsed from the beginning
time	simulated time elapsed from the beginning
energy	cumulative energy consumed
total_pkt	total packets received or transmitted
total_bit	total bits received or transmitted

few seconds of real traffic in high, medium, and low arriving rates as individual inputs to the simulator.

Traces generated from simulation contain a set of architectural execution events that occur frequently during simulation and a set of values related to power and performance, which are called event annotations. In this set of experiments, three types of events, $pipeline$, $forward$, and $fifo$, are mainly used, as explained in Table 3.5. In a simulation trace, the events are prefixed to differentiate different microengines (MEs), threads, or configurations. For example, *m2_pipeline* represents a pipeline event from ME2. Each event is associated with five annotations, also explained in Table 3.5. A snapshot of a trace file generated by NePSim simulator is shown in Figure 3.8.

3.4.2 Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) [23] is a popular low power technique that has been employed widely for microprocessors, resulting in significant power and energy savings. DVS

```
cycle  time   energy    total_pkt  total_bit  event
              ... ...
365    1.573  0.773932  1          4096       m2_pipeline
365    1.573  0.768133  2          9216       m3_pipeline
368    1.586  0.794108  1          4096       forward
368    1.586  0.784506  3          11264      m5_pipeline
369    1.590  0.809369  2          9216       forward
              ... ...
```

Figure 3.8: *NePSim simulation trace for performance and power analysis*

exploits the variance of a processor's utilization, reducing voltage and frequency (VF in short) when the processor has low activity and increasing VF when the peak performance is required. Dynamic power consumption is proportional to $C \cdot Vdd^2 \cdot \alpha \cdot f$, so reducing voltage ($Vdd$) and frequency ($f$) can significantly reduce power consumption.

Although many DVS algorithms appear in literature, the unsolved difficulty is how to derive the optimal settings from external observations, for example, by monitoring the workload or idle time. In this section, we will use assertion-based methodology to study and find out optimal DVS parameters in NPUs.

3.4.3 Power Analysis

We first use our assertion-based analyzer to check the maximum power and power distribution for the NPU model. We simulate our 4 benchmarks, each of which is executed for 8×10^6 cycles with an unlimited packet arriving rate.

Long period of high power consumption can increase the temperature to the extend of damaging the chips themselves. Therefore, we check a constraint for the maximum power

Table 3.6: *Power values for the 4 benchmarks*

	ipfwdr	md4	nat	url
MAX	1.45	1.7	1.7	1.65
MIN	0.6	0.3	0.6	0.3

consumption in the six microengines: “the power consumption within every 5 instructions pipelined should be smaller than a threshold value *a*”. The constraint can be specified with an LOC formula:

$$\frac{eng(pipeline[i+5]) - eng(pipeline[i])}{time(pipeline[i+5]) - time(pipeline[i])} \leq a \tag{3.19}$$

The number of 5 is the window size we used to observe the power. The window is sliding, so all instances will be checked. It doesn’t change the results if the window size is 10 or 100. The checker executes in less than 1 minute of CPU time. The threshold value *a* in the formula (3.19) is changed gradually, and we get the maximum and minimum power consumption in 5-pipeline-event time windows (Table 3.6). The characteristics of different benchmark result in different min/max power. *nat* has highest maximum and minimum values. This is because it has no SDRAM accesses, so there is no long latency for memory access and the MEs are kept busy running.

Besides checking whether the NPU consumes power within a safe range, we are also interested in how the power values are distributed. We want to know whether it stays close to the average value, or spreads over a wide range. Formula (3.19) is extended for distribution

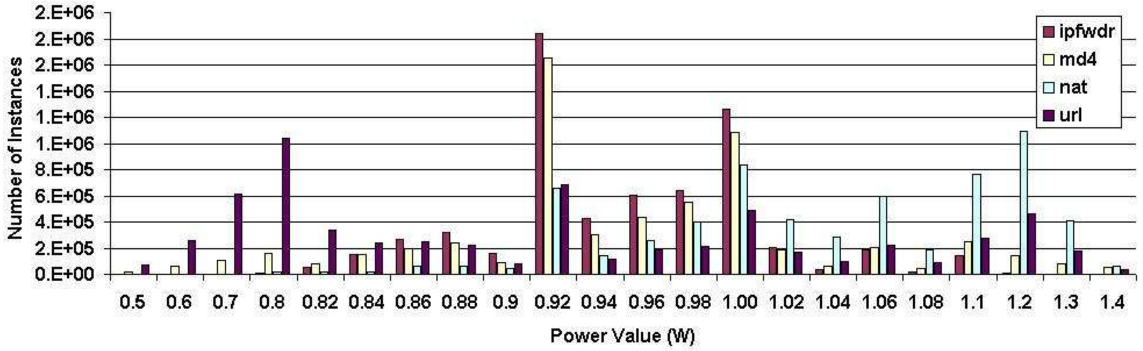


Figure 3.9: *Power distribution graph for 4 benchmarks*

analysis as follows:

$$\frac{eng(pipeline[i+5]) - eng(pipeline[i])}{time(pipeline[i+5]) - time(pipeline[i])} \bowtie \{0.40, 1.40, 0.01\} \tag{3.20}$$

Figure 3.9 shows the power distributions for the 4 benchmarks generated from the assertion analyzer. [4] We can see that all the benchmarks show a high percentage of power values between 1.00W to 0.90W. The benchmarks *ipfwdr* and *md4* have 28% and 26% of total formula instances (i.e. *i*'s) with power between 0.90W and 0.92W. Another frequent range is between 0.98W and 1.00W, which is caused by some frequently used instruction patterns, e.g. common computation operations. We can also see that the NPU is working around $\pm$10% of the average power for around 70% of the total simulation time. The minimum and maximum power consumptions rarely appear. This is a favorable situation to the chips since they will not become too hot by running in high power for short spurts.

[4]For clearer presentation, infrequent ranges are merged in the graph.

3.4.4 Design Exploration for DVS

In a real system with DVS, the frequency and voltage are adjusted dynamically according to the processing workload. A DVS scheduler relies on the history information of workload to make decisions. In an NPU design, two types of information can be used for this purpose, network traffic load and processor idle time. We call the two DVS policies traffic-based dynamic voltage scaling (TDVS) and execution-based dynamic voltage scaling (EDVS). The two policies are usually not combined since monitoring both traffic load and processor idle time on a chip is expensive in terms of area and power.

We analyze the power-performance trade-offs of DVS policies by varying the window size and threshold for voltage/frequency scaling, and search for optimal points in the design space. We also compare the two DVS policies under different design requirements.

Traffic-based Dynamic Voltage Scaling

TDVS uses the total traffic load detected at the 16 device ports as the control parameter for scaling. If the traffic volume in the previous time window is smaller or larger than a particular threshold value, we scale down or up the VF of the processor by one step, until a lower or upper bound is hit. The lower and upper bounds of VF, similar to those used in Intel XScale [8], are from 400MHz to 600MHz and 1.1V to 1.3V. We set the frequency step to 50Mhz and compute the voltage as in XScale. In order to match higher NPU frequency, we scale the speed of SDRAM, SRAM and ixbus to 1.3 times of those in IXP1200.

To estimate the power in TDVS, we modified NePSim's power estimation module to include the power overhead, a 32-bit adder. The adder is used to accumulate the packet sizes in each monitor window, and compare the traffic volume with the threshold. Note this adder is only used when a packet comes in, much less frequently than the ALUs in ME pipelines. From the experiment results, we find the overhead is less than 1% of total power.

TDVS reduces the power, but it may adversely affect the performance. The clock cycle becomes longer if Vdd is decreased, so the NPU takes longer time and possibly more energy to get the same amount of work done. The trade-off motivates us to analyze both power consumption and performance of the NPU with different TDVS policies applied. The goal is to find the optimal points in the design space for each benchmark. We use the following LOC formula to analyze the power consumption distribution:

$$\frac{energy(forward[i+100]) - energy(forward[i])}{time(forward[i+100]) - time(forward[i])} \triangleright \{0.5, 2.25, 0.01\} \ . \tag{3.21}$$

The left hand side of the formula calculates the average power consumption for each 100 packets forwarded.

To study the performance of the processor with various configurations, we analyze the distribution of the transmitting throughputs using the following formula:

$$\frac{(total\_bit(forward[i+100]) - total\_bit(forward[i]))/10^6}{time(forward[i+100]) - time(forward[i])} \triangleleft \{100, 3300, 10\} \ . \tag{3.22}$$

The left hand side of the formula calculates the average forwarding bit rate in Mbps for each 100 packets forwarded.

Table 3.7: *Voltage scaling values*

Frequency (Mhz)	600	550	500	450	400
Voltage(V)	1.3	1.25	1.2	1.15	1.1
Traffic Threshold(Mbps)	1000	916	833	750	666

With the two formulas, we search for the optimal settings of TDVS policies. In TDVS, two main types of parameters that need to be carefully tuned are the traffic thresholds and window size. For each TDVS policy, the traffic thresholds are a set of volume numbers that control the voltage scaling in different VF combinations. With the frequency and voltage reduced, the traffic threshold is also lowered to match the reduced ME processing capability. Taking *ipfwdr* as an example, if we choose a top threshold of 1000Mbps for the normal frequency of 600MHz and other thresholds for reduced VFs are decided as listed in Table 3.7. In our experiments, we use the benchmark *ipfwdr* to compare the TDVS policies with four different top thresholds, 800, 1000, 1200 and 1400 Mbps.

The window size decides how long a traffic history is used to make voltage scaling decisions, and it also directly affects the overall performance of the TDVS policy. For example, if the window size is set to 20k clock cycles, the average traffic volume in the previous 20k cycles is compared to the current threshold to decide whether the VF needs to be changed. If a window size is too large, it may smooth the peak traffic with low traffic and miss a good chance to reduce power; If window size is too small, VF may change too frequently, which incurs more penalty and eventually hurts the performance. In the experiments, the penalty for each voltage scaling is 10μs [52], which is equivalent to 6000 cycles at the normal fre-

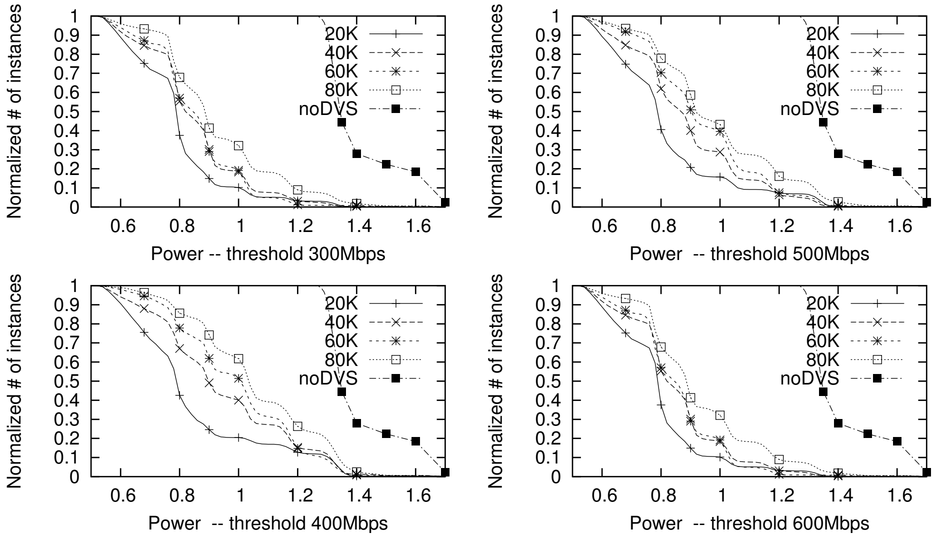


Figure 3.10: *Power under different design points with TDVS*

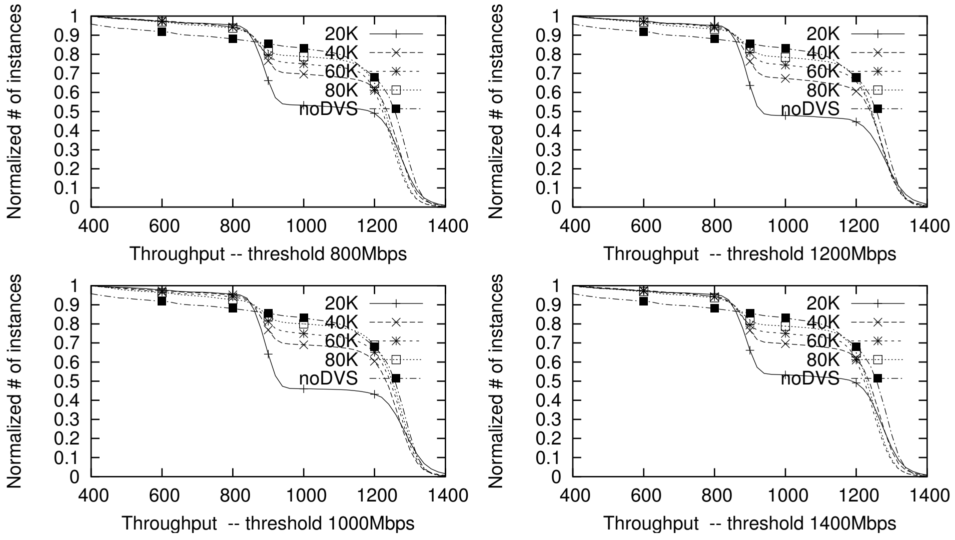


Figure 3.11: *Throughput under different design point with TDVS*

quency of 600MHz. We compare 4 different window sizes for *ipfwdr*, ranging from 20k to 80k cycles.

We run the simulation 8×10^6 cycles for each TDVS configuration. Using the automatically generated distribution analyzer with the formulas (3.21) and (3.22), we compare the power and performance distributions with different TDVS policies or no TDVS enabled. The distributions for the power and performance are plotted in Figure 3.10 and Figure 3.11 respectively. Each subgraph shows the power or throughput distribution with a particular top threshold and different window sizes. In the power distribution graphs, the horizontal axis represents possible power values and the vertical axis represents the percentages of assertion instances that are smaller than particular power values. Similarly, in the throughput distributions, the vertical axis represents the percentages of assertion instances that are larger than particular throughput values.

From Figure 3.10, we can see that compared with no TDVS policy, the power saving by TDVS is obvious no matter what threshold or window size is chosen. And in most cases (except with window size of 20k), the performance degradation is small (from Figure 3.11). So it is shown that TDVS is a very successful power saving technique. We also see that TDVS configurations with smaller window sizes have lower power consumption but worse throughput, regardless the threshold values. When window size is small, e.g 20k, the TDVS policy becomes very aggressive. The VFs are changed very frequently, and as a result, the 6000-cycle penalties almost consume 30% of the window time. That is the reason why there is dramatic drop in throughput for window sizes of 20k. On the contrary, for 80k window sizes, they still achieve certain power savings with almost no performance loss.

To compare the results of different thresholds more clearly and look for a best TDVS policy for *ipfwdr* with an optimal threshold-window size combination, we generate 3-D graphs for power and performance distributions in Figure 3.12 and Figure 3.13. A vertex on the

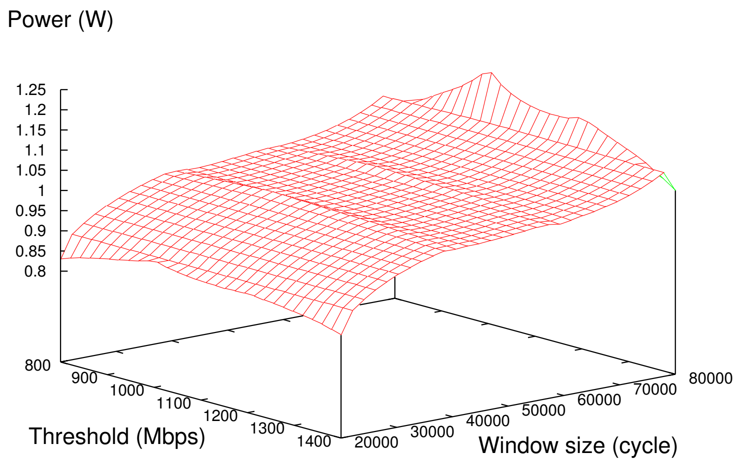


Figure 3.12: *Power under different design points with TDVS*

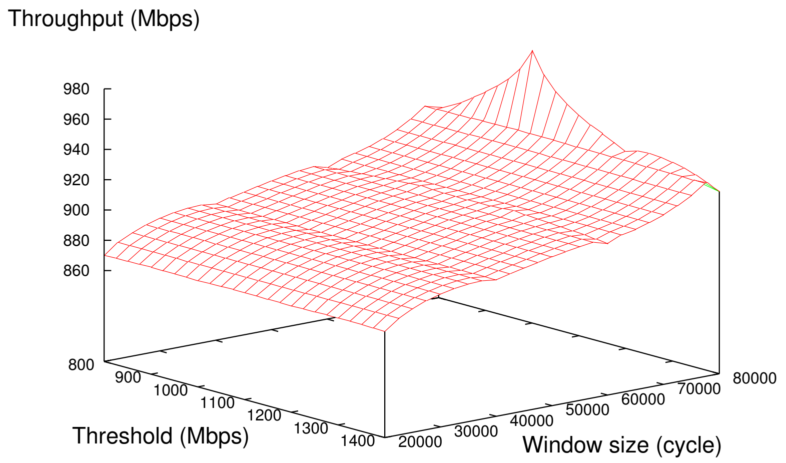


Figure 3.13: *Throughput under different design points with TDVS*

surface shown in Figure 3.12 represents that 80% of formula (3.21) instances are lower than a power value for a particular threshold and window size. Similarly, a vertex on the surface in Figure 3.13 represents that 80% of formula (3.22) instances are higher than a throughput value for a particular threshold and window size. As shown in Figure 3.12, for a particular window size, the threshold of 1000Mbps has higher power than others, and this trend becomes more significant as the window size increases. As shown in Figure 3.13, if the window size is small, the performances for different thresholds are similar; as the window size becomes larger, the performance for 1000Mbps threshold becomes much better than others.

Based on above analysis, if performance has a higher priority in the design, we should choose threshold of 1000Mbps and 80k window size with limited power savings. On the other hand, if saving power is more important, the configuration with 1400Mbps of top threshold and 40k of window size is preferred.

Execution-based Dynamic Voltage Scaling

In execution based dynamic voltage scaling (EDVS), the idle time of a microengine is used as the control parameter for voltage scaling. When the idle time is longer or shorter than a certain percentage of an observed period, the VF of the microengine is scaled down or up by one step, until a lower or upper bound is hit. Note that in EDVS, each ME changes its VF independently.

Intuitively, ME idle time is usually seen to be proportional to the workload, which makes TDVS and EDVS almost the same. However, this is not really the case in the NPU model. Even if an ME does not process packets during low workload, it will actively execute instructions to poll the buffers and status registers to check new packets. In the NPU model, the idle time of an ME is mainly introduced by long latency of memory accesses since an

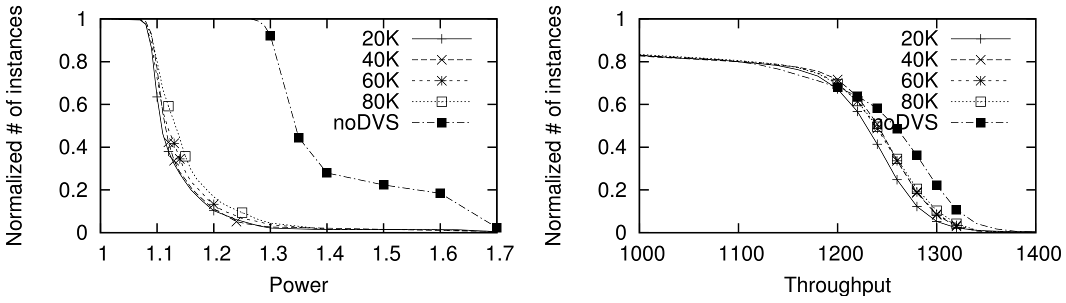


Figure 3.14: *Power and performance distribution for EDVS*

SDRAM access can take as much as 100 clock cycles. If all the threads in an ME are waiting for memory accesses to be completed, we consider the ME idle.

To analyze EDVS policies, the idle time thresholds and window sizes are the main parameters we study and others are configured as those used in TDVS. We use the assertion-based distribution analyzer to find the good idle time thresholds by analyzing the distribution of the idle time in simulations. It is observed that for receiving MEs, in around 90% of the total simulation time, idle time is either under 5%, or between 30% and 40%. For transmitting MEs, idle time is almost always under 5%. The microengines seem working under only two statuses, either busy or idle. Here we simply choose the idle time threshold value as 10%, i.e. if the idle time of an ME is longer or shorter than 10% of an observed period determined by the window size, its VF may be changed. We study three different window sizes, 20k, 40k and 60k and still use *ipfwdr* as the example benchmark.

We run the simulation 8×10^6 cycles for each EDVS configuration and plot the distributions of throughput and power in Figure 3.14. From the power distribution graph, we observe that power dissipation generally drops from 1.5W to 1.15W for most cases with EDVS ap-

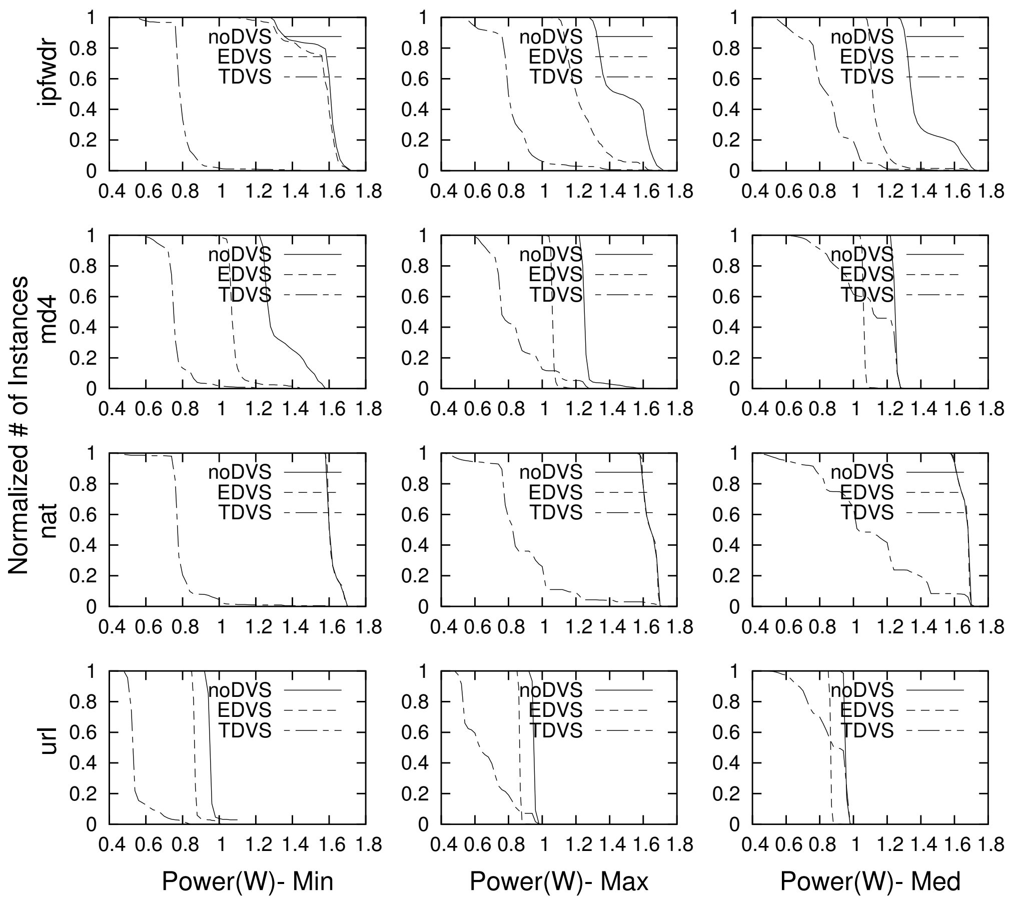


Figure 3.15: *Energy comparisons for employing DVS*

plied, achieving around 23% of power saving. Meanwhile, there is nearly no performance degradation from the throughput distributions. In EDVS, each ME changes its VF independently and the transmitting MEs never scales down their VFs due to their low idle time.

Comparison between TDVS and EDVS

We have shown that both TDVS and EDVS are capable of saving power with little performance impact. Now we are ready to compare the two policies, and find which one is better given a particular power or performance requirement.

We sample the real traffic file in three periods with high, medium, and low traffic volumes respectively. We simulate all four benchmarks with the optimal configurations (from previous analysis) for two DVS policies and compare the power distributions in Figure 3.15. We don't show the throughput performances and only note that in all cases EDVS has no significant performance loss while TDVS never drops more than 2-5% compared to the original NPU model with no DVS applied.

Overall, TDVS has more power savings than EDVS. But as the traffic volume becomes higher, power savings by TDVS are reduced quickly, while EDVS has a more steady reduction under every situation. EDVS has better results for memory intensive benchmarks. We observe that *ipfwdr* shows the most power savings if traffic volume is medium or high. This is because *ipfwdr* needs to check routing tables in SRAM and the output port information in SDRAM for each packet. There are plenty of opportunities for EDVS. The benchmark *nat* shows no power savings from EDVS under every traffic patterns due to the fact that *nat* has very few memory accesses, and the MEs are kept busy.

In summary, if the power consumption is the dominant design factor, TDVS shall be a better choice. Otherwise, if performance is more important and packet loss needs to be avoided as much as possible, EDVS shall be used.

Chapter 4

Deadlock Analysis with Built-in Simulation Monitors

In the design of highly complex, heterogeneous, and concurrent systems, deadlock detection and resolution remains an important issue. Even with careful methodological guidance, it is still possible to introduce unintended and undesirable behaviors into function specifications, high level architecture models, or function-architecture mappings. Foremost among these are deadlock, livelock, and starvation. Being semantic in nature, their complete and precise characterization requires formal analysis or verification, which can only be done at a high level of abstraction due to the state space explosion problem. In this chapter, we look for a practical solution to deal with these design problems in realistic and complex system designs. A simulation based analysis methodology is proposed for the detection and elimination of these "semantic errors". Designers are responsible for coming up with simulation vectors and scenarios that are important and may lead to undesirable behaviors such as a deadlock. Our approach automatically analyzes the simulation status and reports deadlocks once they occur.

4.1 Introduction to Deadlock Analysis

Deadlock detection and resolution techniques have already been extensively studied in the areas of operating systems and database systems [28, 61, 48, 58]. In those domains, deadlock prevention is possible if particular resource allocation policies are applied. Deadlock avoidance is used as a part of scheduling algorithms to choose at least one possible execution path where no deadlock will occur. A resource allocation graph or state graph is usually used to analyze and identify deadlock situations for deadlock detection. Though it is possible to incorporate these techniques in a system design to eliminate deadlocks, they are not general enough to apply to arbitrary designs due to the design flexibility required by today's platform-based embedded system designs. Our deadlock analysis mechanism is integrated in the design framework (rather than the designs) to help designers analyze design errors while allowing full design flexibility.

In communications and concurrent software, various formal verification techniques are employed to exhaustively search deadlock situations in concurrent protocols [35, 34, 55, 42]. In essence, synchronization protocols at a high level of abstraction, either extracted from the design or defined *a priori*, are formally verified. In the latter case, lower level implementations are then developed manually keeping as close as possible to the higher level protocols. The current approaches suffer from at least three problems. Firstly, abstraction of synchronization protocols from a complex design is non-trivial and error-prone. Secondly, complex modern synchronization structures are becoming too complex, and their analysis also suffers from state explosion problem. Thirdly, when a protocol is formally verified *a priori*, it is still quite difficult to get designers to follow exactly the verified protocol, not to mention that the design flexibility is considerably reduced. Our approach is based on simulation, so it can handle real complex system designs.

In simulation verification, assertions that are based on temporal logics can be used to check safety properties in a certain period of execution [6]. However, temporal assertions have to be designed according to particular applications. They are usually used to check the overall behavior of a system, and not suitable for identifying the causes of those undesirable behaviors due to their “trace checking” nature. A general deadlock detection mechanism is proposed in [50] for discrete event simulation models. However, no implementation on real simulation models has been discussed in the literature. In emerging simulation environments for heterogeneous system level designs, an effective and efficient deadlock analysis tool that can be tightly integrated into the design methodology is needed, which is the main focus of this chapter.

While our deadlock monitoring approach can apply to any system level design environment, we focus our effort on the synchronization dependency and deadlock analysis for simulation in the Metropolis design framework [20]. Metropolis is a system level design framework for modern embedded systems. In the modeling language of Metropolis, Metropolis Meta-Model (MMM), a design is specified as asynchronous processes with communication specified with media and with its overall behavior limited by the synchronization constructs: function-architecture mappings, *await* statements, interface function calls, constraints, and schedulers. The function and abstract architecture of a system are specified separately and correlated by synchronization of functional events with architectural events (*mapping*). An *await* statement can be used to make a process wait until some conditions hold, to establish critical sections that guarantee mutual exclusion among different processes, and to prevent interface function calls by other processes. To limit the behavior of processes, designers can put high-level LTL (Linear Temporal Logic) or LOC (Logic of Constraints) constraints on the system specification without giving any specific scheduling algorithm, and leave the implementation to the lower levels of abstraction. Designers can also write their own schedulers in architecture models at a high abstraction level, which are called *quantity managers* in Metropolis. The high flexibility of the design platform allows designers to use different modeling constructs freely in a system design. Without a platform-supported systematic analysis mechanism, this flexibility can lead to vulnerability to design errors that may cause deadlocks.

We identify and analyze deadlock problems in Metropolis simulation. We propose a data structure called the dynamic synchronization dependency graph (DSDG) that reflects the runtime blocking dependencies among processes. We also devise an associated deadlock detection algorithm to monitor the simulation. The goal of the synchronization dependency analysis is to help the designer identify the components (e.g. processes and media) and synchronization constructs (e.g. *await* and *synch*) that are causing any deadlock problem and to provide an error trace or a history of dependency snapshots that show how the system arrives at this state. We use a real world Metropolis design, the *resize* component in a picture-in-picture (PiP) video processing system, to demonstrate the usefulness and effectiveness of the deadlock analysis approach. We also use a high level mapping model that includes a functional specification, an abstract architecture model, and mapping to illustrate how design problems from the function-architecture mapping can be analyzed.

4.2 Synchronization in Metropolis

In this section, we review the synchronization constructs in Metropolis Meta-Model and discuss how deadlock situations are caused by the synchronization mechanism in a concurrent system model.

4.2.1 Synchronization Constructs

The modeling constructs for synchronization in MMM include *synch* constraints, *await* statements, interface functions, quantity managers, and LTL and LOC constraints. Most of these synchronization constructs are not unique to MMM, and their counterparts are also used in other concurrent modeling languages.

In Metropolis, the function and architecture of a system are modeled as separate networks of processes communicating through media. In a functional network, functional processes run concurrently and communicate with each other through media. In an architectural network, computing and storage resources are modeled with media. Services that the architecture can provide are modeled with processes, which are called *mapping processes*. A function model is mapped to an architecture model as the events of functional processes and mapping processes are synchronized with *synch* constraints. A designer is allowed to implement particular schedulers as *quantity managers* to manage architectural resources and services in an architecture model. Quantity managers are basically scheduling media that implement a particular set of functions that can be invoked by processes to issue service requests. An architectural mapping process may be suspended by a quantity manager if it requests resources (quantities in Metropolis terminology) from it. The corresponding functional processes that are mapped to the mapping process can then be blocked through *synch* constraints.

A *synch* constraint is an alternative of a rendezvous used in the concurrent programming [41, 25]. It can specify that two events in two different processes must occur at the same time. If only one of the two events can be scheduled to occur, the process containing the event has to be blocked until the other event can occur also. A *synch* can also require that an event cannot occur until any of the other events occur. The execution of a process has to be blocked at a certain event until all the *synch* constraints containing the event are satisfied. For example, assume functional process p_0 and mapping processes p_1 and p_2 have events e_0, e_1 and e_2, respectively, and are synchronized by a *synch* constraint $synch(e_0 => e_1 || e_2)$, which requires that e_0 cannot occur until e_1 or e_2 occurs. This scenario may denote that a

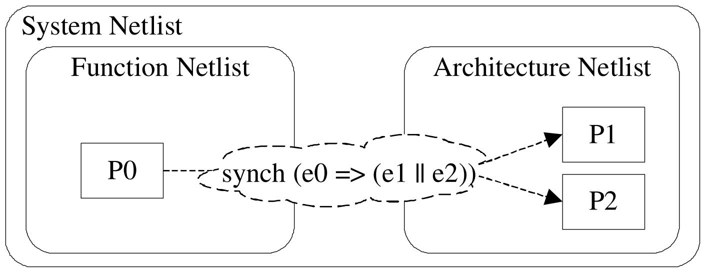


Figure 4.1: *An example of synch constraint*

functional process can not run until there are free computation resources in the architecture. The execution of p_0 may be blocked by either p_1 or p_2, as illustrated in Figure 4.1.

An *await* statement is used to establish mutually exclusive sections and to synchronize processes. It contains one or more statements called *critical sections*, each controlled by a triple (*guard*; *testlist*; setlist). The *guard* can be any Boolean expression, and the *testlist* and *setlist* denote sets of interfaces, which essentially work as integer semaphores that can be incremented or decremented. A critical section is said to be *enabled* if its *guard* is evaluated to true and none of the interfaces in the *testlist* has been set by other processes in the system. A critical section may start executing only if it is enabled. While the critical section is being executed, the "semaphores" specified in the *setlist* are incremented and can block other processes that require the semaphores. The interface function calls are also prevented if the interface is set by an *await*. If no critical section is enabled, the execution blocks. If more than one critical section are enabled, the choice is non-deterministic. For example, an *await* statement has two critical sections:

$$\begin{aligned}&await\ \{\\&\quad (foo();\ intf_{00};\ intf_{01})\ \{critical\_section_0;\}\\&\quad (true;\ intf_{10},\ intf_{11};\ intf_{10},\ intf_{11})\ \{critical\_section_1;\}\}\end{aligned}$$

The first critical section is enabled only if guard $foo()$ is evaluated to true and $intf_{00}$ is not set by other *awaits*. If a process enters this critical section, $intf_{01}$ will be set . The second critical section is enabled only if none of interfaces $intf_{10}$ and $intf_{11}$ is set by other processes. If a process enters this critical section, $intf_{10}$ and $intf_{11}$ will be set by the process. Note that an interface can be set by multiple processes at a given time and must be unset by all of them to be released.

A designer can also add general LTL and LOC constraints to a system to further restrict the behaviors of the system. We do not present these constraints directly here since their specification semantics are not for execution, and it is up to the simulator to make sure that the execution is consistent with the constraints.

4.2.2 Deadlock in Metropolis

Many different definitions can be found in the literature concerning deadlock. In our approach, we define deadlock for Metropolis designs as follows:

Definition 1 *A **deadlock** is a situation where two or more processes are blocked in execution while each is waiting for some conditions to be changed by others.*

Given the constructs considered in MMM, only the following situations may block the execution of a running process:

(1) A process has to wait for synchronization from other functional or architectural processes as required by one or more *synch* constraints.

(2) A process cannot execute an interface function due to the fact that the interface is included in the setlist of a critical section being executed in another process's *await* statement.

(3) A process is blocked at an *await* statement due to the unsatisfaction of all its guard/testlist conditions.

(4) A mapping process is suspended by a quantity manager when it is requesting some quantity from it but cannot be satisfied.

The interaction of these synchronization constructs can be quite complicated. A deadlock exists if and only if there exist dependency loops among the processes in a system. We will identify and analyze the deadlock situation and report the processes and the media to which they are connected. Livelocks and starvations are much harder to identify. Formal verification [55, 42] is required to conclusively identify them by searching for infinite cyclic executions and infinite blocking conditions, respectively. However, synchronization dependency analysis is still useful to provide a guide to the designer to help isolate the problem. Furthermore, while we do not attempt to analyze a system to make sure it is deadlock-free and only report deadlocks as they occur in a specific simulation, we can give heuristic guidelines and suggestions whether or not a deadlock is likely to occur in the future or with other simulation vectors, by showing the system dependency condition at any given simulation state.

4.3 Synchronization Dependency and Deadlock Analysis

In this section, we introduce a deadlock analysis methodology for system level designs. We propose a data structure called the dynamic synchronization dependency graph (DSDG) used in the Metropolis design environment for deadlock analysis. Though the graph is defined according to the execution of Metropolis Meta-Model constructs, it represents general synchronization characteristics in today's system level designs and can be easily applied to other languages and design environments. Once the synchronization dependencies are captured by the graph, an algorithm can be used to detect deadlock situations.

4.3.1 Deadlock Analysis Methodology

Our deadlock analysis methodology is illustrated in Figure 4.2. By integrating deadlock analysis tools in a simulation environment for system level designs, designers can efficiently analyze complex concurrent systems with simulation and quickly identify design problems that may cause deadlocks. The task of design analysis becomes much easier with the help of runtime synchronization information combined with regular simulation traces and static

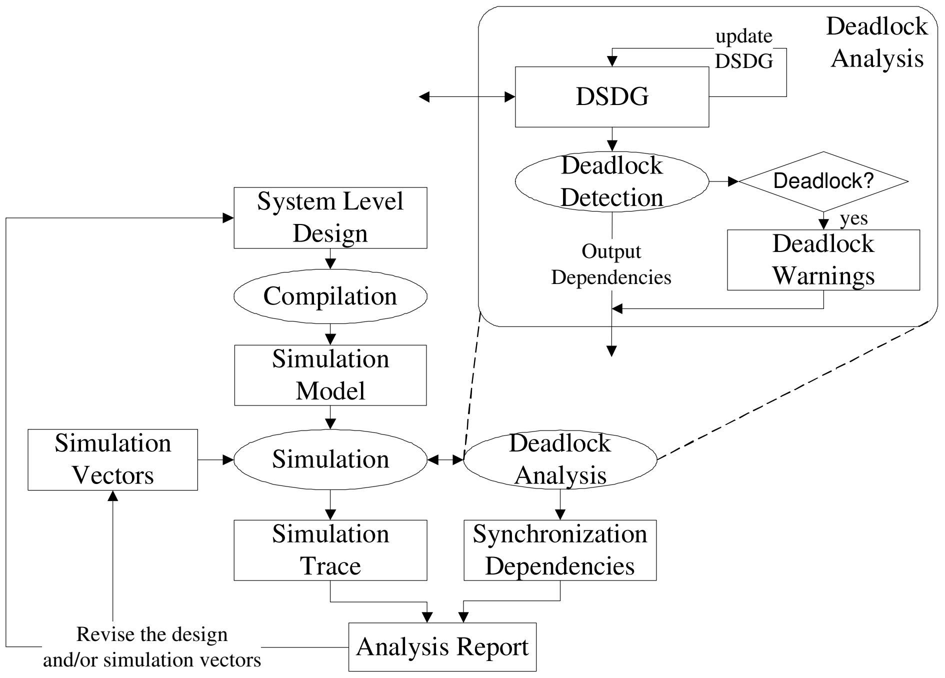


Figure 4.2: *Deadlock analysis methodology*

network structures. They can be used to guide a designer to revise the design to eliminate problems or modify simulation vectors to explore different execution paths looking for other design errors. This methodology allows full design flexibility and is able to handle large models. The details of the deadlock analysis mechanism will be discussed in the rest of this section.

4.3.2 Dynamic Synchronization Dependency Graph

Definition 2 *A* ***DSDG*** *(dynamic synchronization dependency graph) is a directed graph S=(V, E). V is a set of four categories of vertices representing processes in the network, or-dependency, and-dependency, and eval-dependency. E is a set of directed edges between vertices indicating dynamic synchronization dependencies.*

Algorithm 2 *Main procedure to build and update a DSDG.*

```
procedure UPDATE_DSDG()
  for each process $p_i$ in the system do
    if $p_i$ is unblocked by one or more synch. constructs then
      remove all the dependency vertices and edges from $p_i$ caused by those synch. constructs;
    end if
    if $p_i$ is blocked by one or more synch. constructs then
      UPDATE_PROCESS($p_i$);
    end if
  end for
end procedure
```

In a DSDG, each process in the network is represented by a process vertex. Other dependency vertices and edges are added or removed dynamically as dependencies between processes change in the execution. An *and-dependency* requires a process to be blocked until all the conditions become satisfied. An *or-dependency* indicates that as long as one of the conditions becomes valid, a process can be released. A *eval-dependency* is used to represent that a process is blocked by a guard of an *await* or by a quantity manager. Guards are not analyzed but simply evaluated to get the valuation of "true", "false", or "blocked". Similarly, quantity managers are invoked to decide if processes that are making requests need to be suspended or not. If a process is blocked by a guard or a quantity manager, it has dependencies on the processes that may change the evaluation of the guard or the quantity manager sometime later. A DSDG is automatically built and updated during the simulation, and it describes the status of dependencies among all the concurrent processes of a system at a particular execution state. If a process is actively running, there is no outgoing edge from it in the graph. If it is blocked or released, dependency edges and vertices will be added to or removed from the graph dynamically. A DSDG is built and updated with dependency

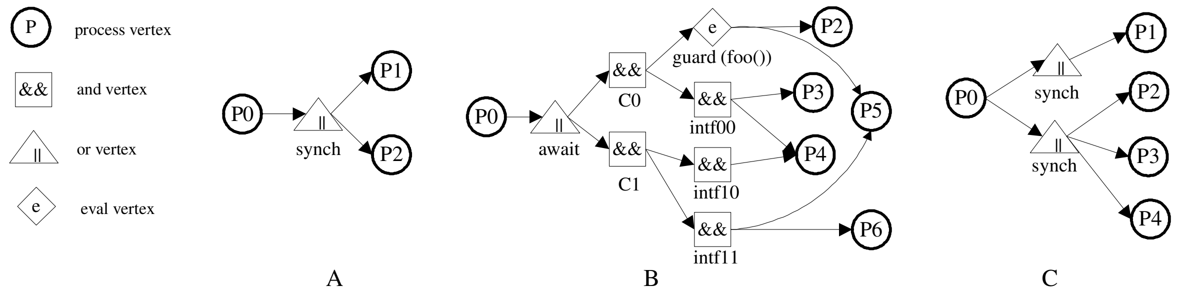


Figure 4.3: *DSDG examples*

Algorithm 3 *Procedure to handle a blocked process.*

```
procedure UPDATE_PROCESS($p_x$)
  for each synchronization construct that blocks $p_x$ do
    if $p_x$ is blocked by a synch constraint that requires its waiting for any of processes $p_1$, $p_2$, $\ldots$, and $p_n$ then
      add an or-dependency vertex $o_x$;
      CONNECT($p_x$, $o_x$, $\{p_i : i \in [1, n]\}$);
    else if $p_x$ is blocked by an interface function $I$ then
      add an and-dependency vertex $a_x$;
      CONNECT($p_x$, $a_x$, $\{$processes that prevent the interface $I\}$);
    else if $p_x$ is blocked by a quantity manager $Q$ then
      add an eval-dependency vertex $e_x$;
      CONNECT($p_x$, $e_x$, $\{$processes that are managed by $Q\}$);
    else if $p_x$ is blocked by an await then
      UPDATE_AWAIT($p_x$, $\{C_i :\ i \in [1, n]$ and $C_i$ is a critical section$\}$);
    end if
  end for
end procedure
```

vertices and edges dynamically added or removed using the procedures in Algorithm 2 to 5. Initially, V only includes all the process vertices, and E is set to $\emptyset$. During simulation, UPDATE_DSDG() is called to update S every time the synchronization dependencies of the system are changed. UPDATE_PROCESS() is called to update the DSDG for each blocked process, and UPDATE_AWAIT() is called for a blocking *await*. CONNECT() connects newly added vertices with directed edges.

Algorithm 4 *Procedure to connect newly added vertices.*

```
procedure CONNECT($src$, $mid$, $\{dest_i : i \in [1, n]\}$)
  add an edge from $src$ to $mid$;
  for i := 1 to n do
    add an edge from $mid$ to $dest_i$;
  end for
end procedure
```

Algorithm 5 *Procedure to handle a blocking await.*

```
procedure UPDATE_AWAIT($p_x$, $\{C_i : i \in [1, n]\}$)
  add an or-dependency vertex $o_x$;
  for each critical section $C_i$ ($1 \leq i \leq n$) do
    add an and-dependency vertex $a_i$;
    if the guard condition is evaluated to false then
      add a eval-dependency vertex $g_i$;
      CONNECT($a_i$, $g_i$, {processes that may change the guard});
    else if the evaluation of the guard is blocked then
      recursively call UPDATE_PROCESS($a_i$) to add dependency vertices and edges as if $a_i$ is a
      blocked process;
    end if
    for each prevented interface $intf_{ij}$ in $C_i$'s testlist do
      add an and-dependency vertex $a_{ij}$;
      CONNECT( $a_i$, $a_{ij}$, {the preventing processes});
    end for
  end for
  CONNECT($p_x$, $o_x$, $\{a_i : i \in [1, n]\}$);
end procedure
```

Figure 4.3A shows an example DSDG of a process p_0 being blocked by a constraint $synch(e_0 => e_1 || e_2)$, which requires that e_0 cannot occur until e_1 or e_2 occurs. Figure 4.3B shows an example of a process p_0 being blocked by an *await*. The *await* statement has two critical sections C_0 and C_1. Assume that, in C_0, the guard is evaluated to be false and is accessible by p_2 and p_5, which is represented by a guard vertex. The interface $intf_{00}$ in its testlist is blocked by p_3 and p_4. In C_1, the guard is always evaluated to be true, but the interfaces $intf_{10}$ and $intf_{11}$ are blocked by other processes. Figure 4.3C shows an example where a process p_0 is blocked by 2 $synch$ constraints at the same time. Note that each dependency

Algorithm 6 *Deadlock detection.*

```
procedure DETECT_DEADLOCK($S$, $P$)
  search for simple cycles in $S$ from process vertices in $P$;
  let $\mathbb{L}$ = $\{L_i$=($V_i$, $E_i$)$\}$ be the set of all these simple cycles;
  if $\mathbb{L} = \emptyset$ then
    return NO_DEADLOCK;
  end if
  for each $L_i \in \mathbb{L}$ do
    if $L_i$ is already marked then
      continue;
    end if
    mark $L_i$;
    if each vertex in $V_i$ is either a process or and-dependency then
      the processes in $L_i$ are deadlocked, return;
    else
      $D$ := $\{$eval- and or-dependency vertices in $V_i$ that have two or more outgoing edges$\}$;
      $\mathbb{L}'$ := $\{L_i\}$;
      repeat
        find unmarked cycles in $\mathbb{L}$ that contains vertices in $D$;
        mark all these cycles;
        $D := D \cup$ $\{$eval- and or- dependency vertices with two or more outgoing edges in these
        cycles$\}$;
        $\mathbb{L}'$ := $\mathbb{L}' \cup$ $\{$these cycles$\}$;
      until $\mathbb{L}'$ becomes stable
      if $\exists$ vertex in $D$ that has an outgoing edge $\notin$ $\mathbb{L}'$ then
        continue;
      end if
      the processes in $\mathbb{L}'$ are deadlocked, return;
    end if
  end for
  return NO_DEADLOCK;
end procedure
```

vertex is labeled to indicate the exact location in the source code that it is corresponding to. This information can be made available for the designer to help identify the problem quickly.

4.3.3 Deadlock Detection Algorithm

Given a dynamic synchronization dependency graph $S = (V, E)$ and a set of processes that are blocked from running P, we use Algorithm 6 to detect deadlock situations. Generally, the algorithm traverses the graph, searches for cyclic dependencies, and determines deadlocked processes according to the and- , or- and eval-dependencies among processes. The algorithm not only decides if there is any deadlock but also identify all the processes and synchronization constructs that are involved in deadlock situations. The algorithm works incrementally, starting from the newly added dependencies, since the part of the graph not affected by the new dependencies has already been checked to be deadlock-free. In the worst case, the first step of the algorithm is to find all the simple cycles in the graph. Its complexity is $O(|V| \cdot (|V| + |E|))$ assuming that the adjacency-list representation is used for the graph. The rest of the algorithm will traverse all the simple cycles at most twice with a complexity of $O(|V|)$. If a simple cycle only contains process vertices and and-dependency vertices, then it is a deadlock. If a simple cycle also contains or- or eval- dependency vertices, there is a deadlock only if other edges from these or- or eval- dependency vertices all lead to cycles. Therefore, the complexity of the algorithm is $O(|V| \cdot (|V| + |E|))$. $|V|$ and $|E|$, the numbers of vertices and edges in a DSDG, are determined by the number of process instances, interface instances, critical sections of *await* statements and quantity managers in a system.

4.3.4 Implementation

The dynamic synchronization dependency graph and deadlock detection algorithm have been implemented in the simulator of Metropolis framework. During the simulation of a design, a dependency graph is built and updated as the dependency state of the system changes, i.e. as one or more processes in the system are blocked from running or released from blocking.

Whenever one or more processes are blocked from running, the deadlock detection algorithm is invoked to search the DSDG for any deadlock situation. Once a deadlock is detected in the simulation, the history of DSDG updates provides a trace that shows how the system execution goes into the deadlock. Due to the incremental nature of the DSDG update and deadlock detection algorithms, this simulation monitoring mechanism will not introduce significant overhead to the regular simulation.

More complex and hard-to-detect undesirable behaviors in a system are livelock and starvation. Informally, a livelock is a situation where two or more processes keep running and change their states in response to changes in others, but cannot reasonably complete their jobs. A starvation is a situation where a process is blocked due to some condition being unsatisfied and depends on other processes to change the condition. The other processes are still running, but will never make the condition satisfied. Though the DSDG data structure alone does not capture the complete state of a system, a history of dynamically updated DSDGs with other system state information kept can help catch livelock and starvation situations. Specially, a simple algorithm can be used to search any cyclic patterns in the history of DSDGs, which can provide a useful guide to designers to look for livelock or starvation. The simulation deadlock monitoring can also be combined with formal verification techniques to detect those subtle problems in the design automatically.

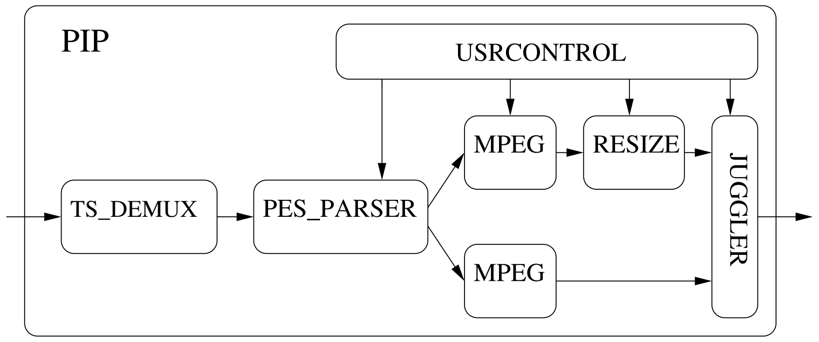


Figure 4.4: *Picture-in-Picture design*

4.4 Case Studies of Deadlock Analysis

In this section, we use two previous examples, a real design of a complex functional model for video processing, Picture-in-Picture (see Section 3.2.1), and a high level model of function-architecture mapping (see Section 3.2.2), to demonstrate the usefulness and effectiveness of our deadlock analysis approach for system level designs.

4.4.1 A Function Model for Video Processing

For convenience, the PiP design is shown again in Figure 4.4. TS_DEMUX demultiplexes the single input transport stream (TS) into multiple packetized elementary streams (PES). PES_PARSER parses the packetized elementary streams to obtain MPEG video streams. Under the control of the user (USRCONTROL), decoded video streams can either be resized (RESIZE) or directly feed to JUGGLER that combines the images to produce the picture-in-picture videos. RESIZE is the major component of PiP that computes and adjusts the size of MPEG video frames according to user inputs. It consists of about 9,000 lines of Metropolis

Meta-Model source code and contains 22 concurrent processes and more than 300 media. The video frames and control signals are passed between processes through around 80 communication channels specified with media. The communication channels are modeled at the task transition level (TTL) with bounded first-in-first-out (FIFO) buffers [32]. The mutual exclusion and boundary checking of the bounded FIFO buffer is guaranteed by a central protocol. To simulate the RESIZE unit, three additional processes are used to mimic user inputs (USER), send MPEG video streams to the unit (SOURCE) and absorb the data from it (SINK) as shown in Figure 4.5A.

In the simulation with our runtime deadlock monitoring mechanism enabled, a deadlock is reported immediately after TMUX_UV and TMEM_CTL_U block each other through two await statements and their synchronization dependencies are captured in the DSDG as shown in Figure 4.5C. As it turns out, there is a design error in process TMUX_UV, which fails to read all the data sent by TMEM_CTL_U.[1] The data in the bounded buffer of the channel between the two processes accumulates until the buffer becomes full. Then a deadlock occurs where TMEM_CTL_U is blocked waiting for the buffer space to be released by TMUX_UV while TMUX_UV is also blocked waiting for reading signals from TMEM_CTL_U. The designer can now focus on the two processes and the communication channels between them to identify and correct those design errors. A solution is to modify process TMUX_UV and make it absorb all the data from its input channels even if not all the data is useful. We observe that, without the deadlock detection mechanism, the simulation will continue and the

[1]As Figure 4.5B shows, process TMUX_UV gets video data from both TMEM_CTL_U and TMEM_CTL_V, combines two streams of data and sends them to its successor process.

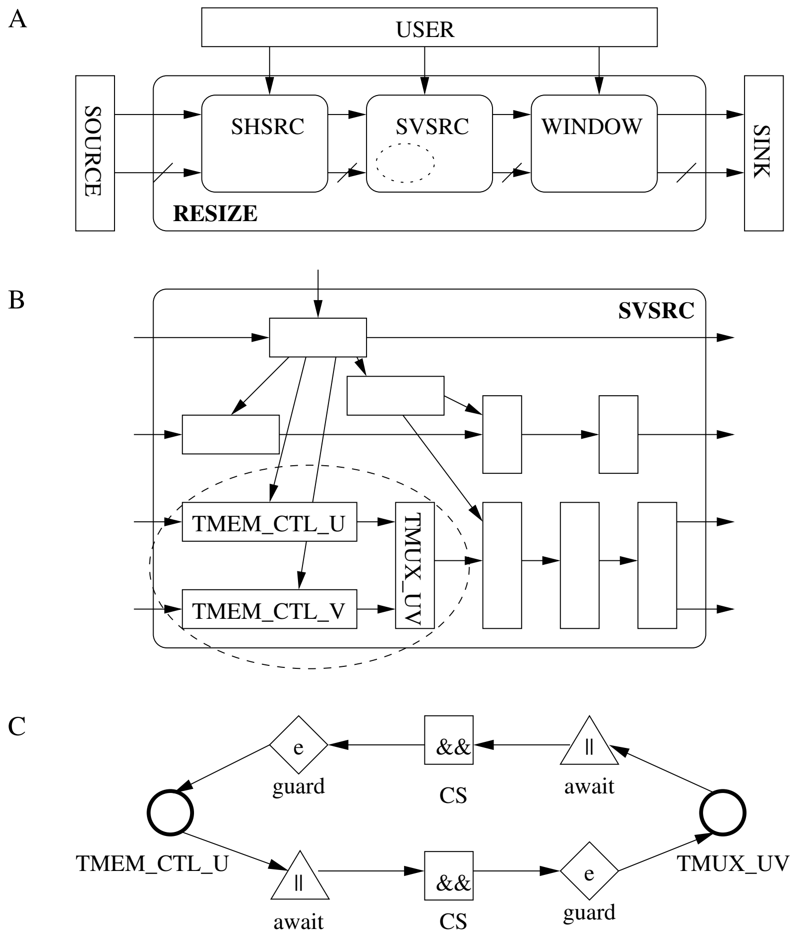
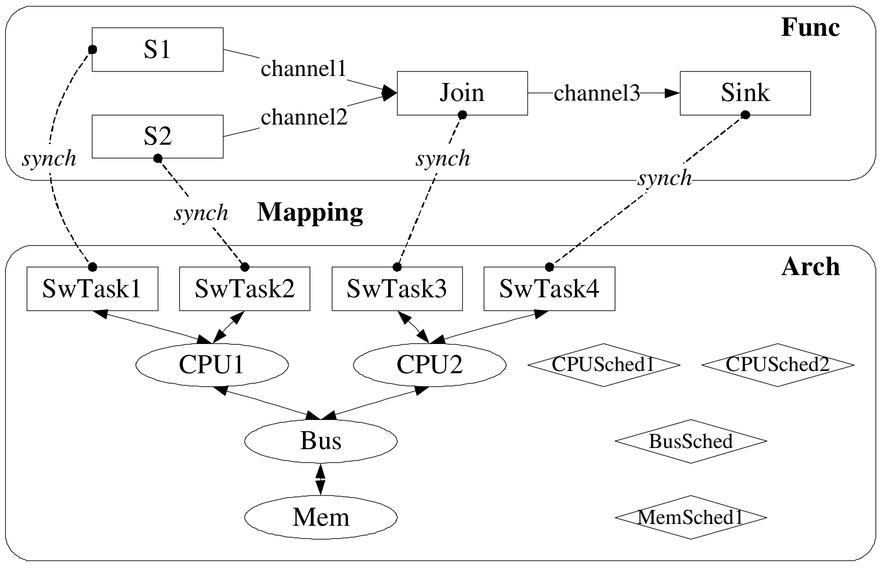


Figure 4.5: *RESIZE unit and its synchronization dependencies*

regular simulation trace won't show any apparent sign of deadlock until most of the processes in the system are eventually blocked. By that time, the simulation trace is long, and a large number of processes are blocked. Our approach automatically catch the deadlock as it first occurs. Designers can then focus on solving the deadlock without complicating themselves by the consequences of the deadlock.

A



B

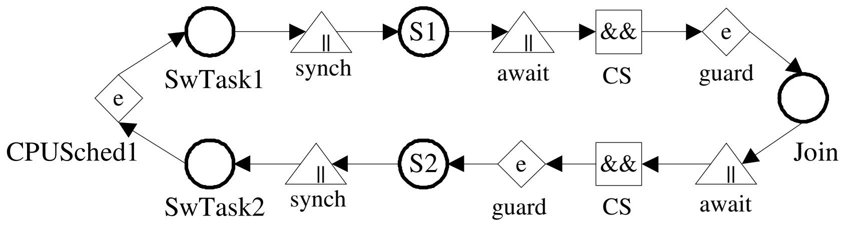


Figure 4.6: *A mapping model and its synchronization dependencies*

4.4.2 A Function-Architecture Mapping Model

In the platform-based design, the mapping is the key procedure that correlates the function to the architecture. In this design example (as shown again in Figure 4.6A), two source processes (S1 and S2) write the data into two independent channels. A separate process (Join) then reads data items from both channels, manipulates them, and then sends the result data to another process (Sink) through another channel. In the abstract architecture model, there are two CPU/RTOS units, a bus unit, a memory unit and a quantity manager (i.e. scheduler) for each architectural unit.[2] A CPU unit can be shared among several software tasks that may

[2]An architectural unit is modeled as a medium in Metropolis.

Table 4.1: *Summary of deadlock analysis case studies*

Example	RESIZE Unit	Mapping Model
Code Size	9000 lines	5900 lines
Processes	22	8
Media	300+	16
Deadlocked Processes	2	5
Time to Catch Deadlock	2min	$<$ 1min

request services from it. When more than one service request is issued to a CPU, arbitration is needed. The mapping procedure synchronizes the processes in the function model and the mapping processes (representing software tasks) in the architecture model. In this example (as shown in Figure 4.6A), functional processes S1 and S2 are mapped to mapping processes SwTask1 and SwTask2, respectively, which are associated to CPU1 and the other two processes are mapped to CPU2. The CPU quantity managers implement a non-preemptive static-priority dynamic scheduling policy. The two CPUs are connected to the bus and the bus is connected to the memory unit.

Our deadlock detection mechanism reports a deadlock within one minute of simulation. Due to the boundedness of the channels between processes, process S1 can not complete a task of writing data before Join reads from and releases the channel buffer. Therefore, with the current CPU scheduling policy, the deadlock occurs when S1 obtains the CPU service but cannot complete a writing task while Join is still waiting for data from S2 who cannot get CPU service. The deadlock situation involves five processes, two await statements, two *synch* constraints and a quantity manager as shown in Figure 4.6B. This analysis also suggests several possible deadlock resolutions. The deadlock can be resolved by making the channel buffer large enough to store all the data from a single writing task, increasing the number of CPUs, or changing the CPU scheduling policy. We also observe that such deadlocks only occur in the mapped design and are not inherent in the function specification or in the architecture model. The simulation and analysis results for this mapping model are also listed in Table 4.1.

Chapter 5

Formal Verification for System Level Designs

This chapter focuses on formal verification of embedded system designs, especially at higher levels of abstraction. We develop a verification methodology for designs that may go through different levels of abstraction and a translation mechanism from system design specifications to descriptions more suitable for formal verification engines. We devise solutions to many challenges encountered in semantically translating from an object-based system design language (i.e. Metropolis Meta-Model [18]) to a procedural verification modeling language (i.e. Promela [42]). In addition, an automatic abstraction propagation algorithm is used to simplify a design specification for specific design constraints. We use a set of realistic case studies to demonstrate our verification approach for system level designs.

5.1 Introduction to Formal Verification

Formal verification can be very powerful for catching errors early in the design process. Formal verification tools, notably model checkers (e.g. Spin [42], SMV [55]), are available to designers. Designers can describe their designs with the given formal language and the design constraints or properties they want to check with some logics (e.g. LTL [53, 59], CTL [26]). If a design constraint or property is found to be false for the design, an error trace is provided by the model checker to designers to help them modify the design or the constraint. The state explosion problem restricts the usefulness of exhaustive proof to protocols or other higher levels of abstraction. Approximate verification (e.g. an option available in Spin [42]) allows model checkers to automatically check a constraint with only a portion of the state space explored. Obviously, approximate verification does not prove that a constraint is satisfied for all conditions. A tool provides the estimated percentage of this partial exploration (i.e. confidence factor) and reports a bug if one is found on the partial state space searched.

One problem for formal verification is that a verification model needs to be written, often manually, from a specification model. This tedious process multiplies if designers wish to verify a constraint of a design as it goes through various abstraction/refinement operations. Our contribution is to fully integrate formal verification tools into the Metropolis framework. Verification models can then be automatically generated for all levels of the design, so designers no longer have to manually re-describe their design in a formal verification language each time a design moves from high levels of abstraction toward implementation. The central challenge in this approach is that verification languages, such as Promela used by Spin model checker [42], allow only simple concurrency modeling and are not amenable to system design specification where complex synchronization and architecture constraints are needed. Our translator automatically constructs a verification model from a specification model, taking care of all the system level constructs.

In constraint-driven verification, only a portion of a design may be relevant to passing or failing of a given constraint. The rest of the design may be simplified or removed, without changing the outcome of the verification. Based on this observation, a technique of automatic design abstraction and propagation is developed to abstract the original specification of a design and to simplify the corresponding verification model. Designers are also allowed to indicate what elements in the design are not relevant to the constraints being verified. They can apply these abstraction operations, freeing in particular, to variables, statements, and components. If the constraints are "safety" in nature (i.e. something bad will never happen), abstraction can only lead to verification results that are either exact or conservative (with possibly false negative result). There will never be a false positive result. We propose an automatic algorithm to propagate this abstraction to the rest of the design exactly (i.e. without introducing more false negatives or any false positives).

In the rest of this chapter, we introduce our verification methodology for system level designs, define translation algorithms for the main Metropolis Meta-Model constructs (such as processes, media, schedulers, await statements, dynamic objects, and mapping), propose design abstraction and propagation algorithms to simplify verification models, and use a set of case studies to demonstrate numerous aspects of verification before and after synthesis and mapping procedures. While we focus on a verification methodology in the context of Metropolis designs, the same approach can be easily applied to other abstraction/refinement design frameworks (e.g. SystemC [7]).

5.2 Formal Verification Methodology

The task of formal verification is to exhaustively search the state space of a system design and to check whether a particular design constraint is satisfied. After a system specification in Metropolis Meta-Model is translated to Promela description, one can use Spin to do model checking. Spin provides two powerful ways to specify constraints of a design: Assertion and LTL (Linear Temporal Logic) [53, 33]. Assertion is an annotation construct in Promela used to "assert" that a particular condition (e.g. space$>$3) must hold. LTL is strictly a superset of Assertion. Without loss of generality, we only deal with the LTL here.

The formal verification methodology for Metropolis is illustrated in Figure 5.1. MMM description is automatically translated into Promela description, and LTL constraints specified in MMM are checked using the model checker Spin. It is known that only a subset of LOC can be translated into equivalent LTL formulas and formally checked with Spin directly(see Section 2.5). For other LOC formulas, formal verification results may be inconclusive, i.e. the verification is only partial. A designer may perform any synthesis step (e.g. composition, decomposition, constraint addition, scheduler assignment), and a new Promela

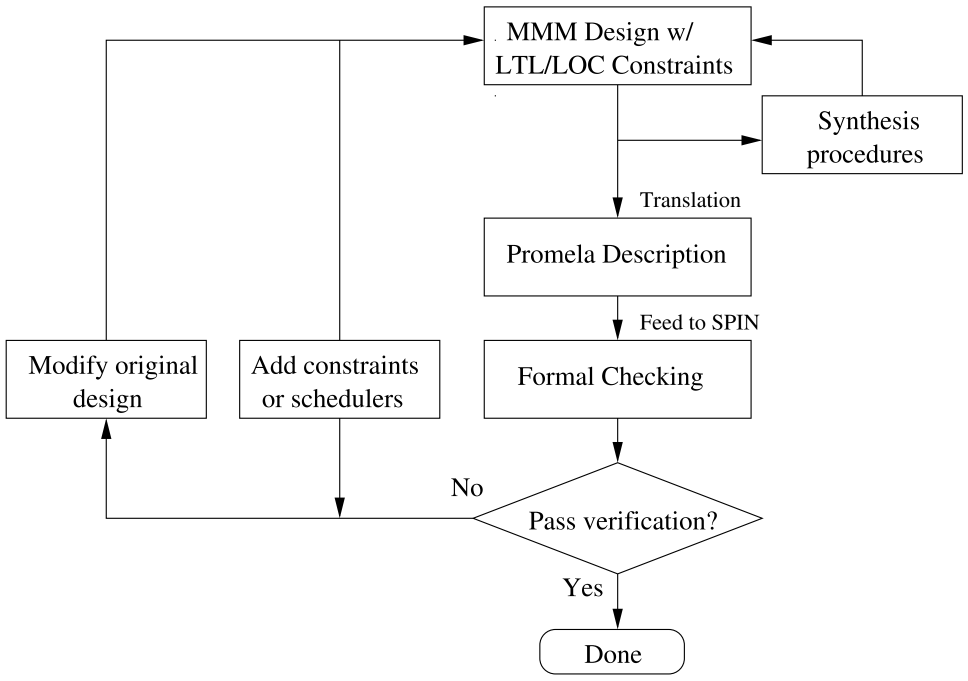


Figure 5.1: *Metropolis formal verification methodology*

code can be automatically generated to verify design constraints. If it does not pass, the error trace may be used to help the designer figure out whether the design needs to be altered. If a verification session runs too long, approximate verification can be used to explore a subset of the state space and report the probability that a constraint will pass. Obviously, a partial exploration cannot prove that a constraint is satisfied. However, it is our experience that a lot of "easy" bugs can be found within a relatively small amount of time and memory usage. If a Spin verification session continues to run after a long time, it is highly likely that the constraint will eventually pass.

Figure 5.2 shows a prototypical network of *m* producers and *n* consumers communicating through a single medium. The producers receive inputs from the environment, process the data in some way, and then output it to a medium of a single space. The consumers read

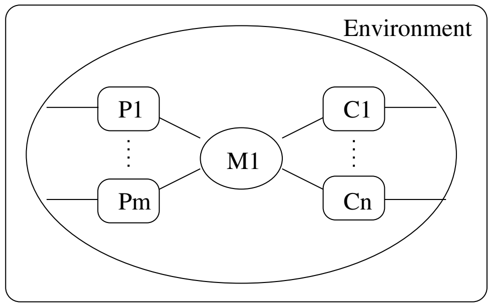


Figure 5.2: *Example of a bytelink meta-model*

in the data from that medium, process it, and then output to the environment. It is possible for all producers and consumers to execute concurrently. If we want to check the constraint, *"whenever a producer writes an item into the medium, there must be some space in the medium"*, it can be specified as an LTL formula:

$$\mathsf{G}\left((P_1\_write \vee \cdots \vee P_m\_write) \rightarrow M_1\_not\_full\right) \ , \tag{5.1}$$

and be verified with Spin after the specification model is automatically translated into Promela.

The same methodology can also be used for a verification-driven synthesis approach. If a constraint does not pass the verification, an error trace is generated and examined. Based on the error trace, the original design may be incorrect, or refinement need be applied to the original specification for it to have the desired constraint. At a higher level of abstraction, abstract constraints can be used to constrain the behavior so the design property can pass verification. At a lower level of abstraction, designers must ensure that these constraints are implemented. This may be achieved, for example, with schedulers on a particular platform.

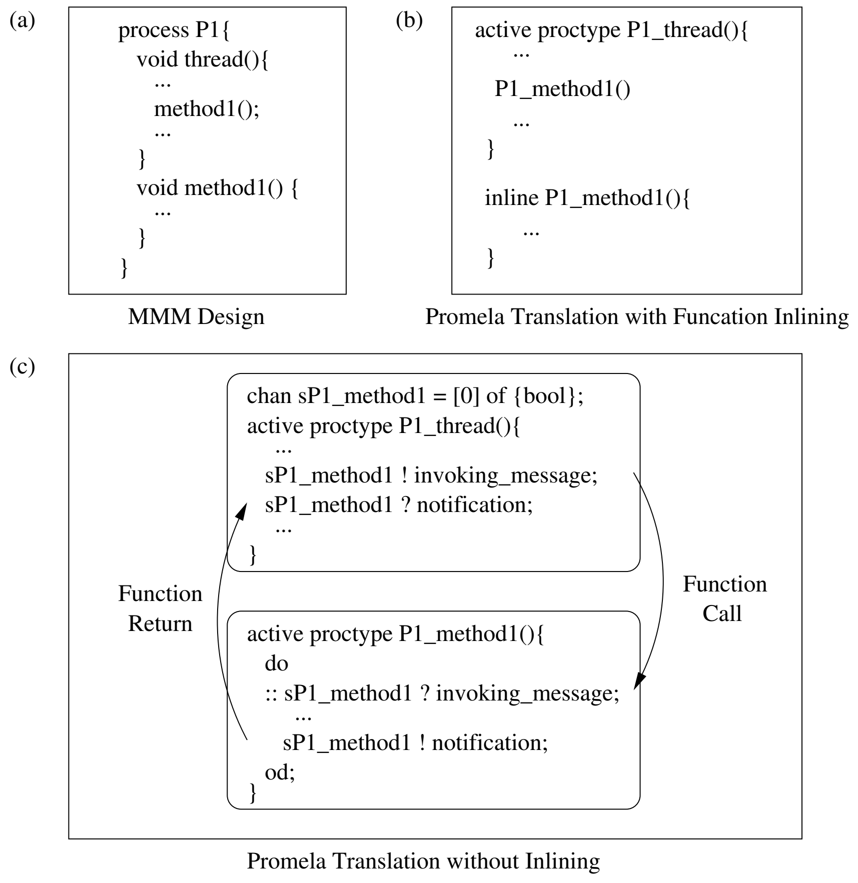
5.3 Translation from MMM to Promela

The Metropolis [18] design framework enables designers to represent and to manipulate their designs at multiple levels of abstraction and with multiple models of computation (MoC). Central to the framework is the Metropolis Meta-Model (MMM) representation. Different high-level languages, models of computation, design constraints, as well as specifications of system functions and architecture platforms can be represented in MMM while retaining their correct semantics. Constructs in MMM are chosen to facilitate the transformations and refinements between different abstraction levels. Incorporated into the Metropolis design environment is a set of back-end tools, with which one can simulate, synthesize, and verify a design at hand.

In Metropolis, the model checker Spin [42] is utilized as one of its back-end verification engines. A design specification in Metropolis Meta-Model is automatically translated into a verification model in Promela, the modeling language of Spin, and constraints of the design can then be formally verified with Spin. Four main issues in the translation from MMM to Promela are the modeling of MMM processes, interfaces and *await* statements for coordinations, dynamic objects, and function-architecture mapping. We do not believe that it will be profitable, at this stage, to develop a new model checker specific to a system level specification language. Instead, we rely on automatic translation, both to decouple this very complex problem and to make it easier for Metropolis to take advantage of the latest advancement from the formal verification community.

5.3.1 MMM Processes

In MMM, communication between processes is made by calling functions defined in the media. One way to model function calls in Promela is to to use active processes to model all instances of meta-model functions, which include all the member functions of processes, media and other objects in the meta-model. Each member function is translated into an active Promela process, which is instantiated and initiated at the very beginning of the execution, and a function call in MMM is translated as invoking an execution of the corresponding Promela process. The invocation is accomplished through message passing using a rendezvous channel (i.e. FIFO channel of size 0). Figure 5.3 illustrates this approach. Figure 5.3(a) shows the function thread() of process P1 making a call to a member function method1(). In Promela (see Figure 5.3(c)), this is interpreted as the process P1_thread sending a message to process P1_method1 through a rendezvous channel sP1_method1 (using operator !). The function return follows the same paradigm. When P1_method1 finishes, it sends back a notification message through the same channel to P1_thread. P1_thread receives the message (using operator ?) and continues its execution. Thus, the sequential execution flows and control transfers of the MMM processes are assured. Due to Spin's limitation on the number of running processes and its resource recycling mechanism [5], dynamically creating new processes is prohibitively expensive. Instead, all Promela processes, except the processes representing meta-model constructors and threads, are initialized at the beginning of execution as active processes blocked waiting for an invoking message from their calling processes through the corresponding rendezvous channels. Member variables of an MMM


(a)
MMM Design
(b)
Promela Translation with Funcation Inlining
(c)
Function Return
Function Call
Promela Translation without Inlining

```
process P1{
  void thread(){
    ...
    method1();
    ...
  }
  void method1() {
    ...
  }
}
```

```
active proctype P1_thread(){
    ...
  P1_method1()
    ...
}

inline P1_method1(){
    ...
}
```

```
chan sP1_method1 = [0] of {bool};
active proctype P1_thread(){
    ...
  sP1_method1 ! invoking_message;
  sP1_method1 ? notification;
    ...
}
```

```
active proctype P1_method1(){
  do
  :: sP1_method1 ? invoking_message;
      ...
    sP1_method1 ! notification;
  od;
}
```

Figure 5.3: *Translations of MMM functions*

process or medium are represented by global variables of Promela after they are renamed appropriately.

To further reduce the overall complexity of the verification, we use a translation approach that inlines all the functions into the process that calls them directly or indirectly. The translator simply pastes its translated code into the point of the invocation in the calling process (see Figure 5.3(b)). No process or channel is needed. In the situation of multiple level function calls, all the functions are inlined recursively so that one MMM process corresponds to only one Promela process. Thus the total number of Promela processes can be shrunk,

```
await {
   (guard1; testlist1; setlist1) {stmts1}
   (guard2; testlist2; setlist2) {stmts2}
     ...
   (guardk; testlistk; setlistk) {stmtsk}
}
```

```
do  //start of await
 :: atomic {
     if   //evaluation of guards and testlists
     ::(guard1 && intfc1_active == 0 && intfc1_exclusive == 0)
       -> intfc1_exclusive ++;         // set setlist 1
          awaitFlag_1 = true;          // select critical section 1
     ::(guard2 && intfc2_active == 0 && intfc2_exclusive == 0)
       -> intfc2_exclusive ++;         // set setlist 2
          awaitFlag_2 = true;          // select critical section 2
      ...
     ::(guardk && intfck_active == 0 && intfck_exclusive == 0)
       -> intfck_exclusive ++;         // set setlist k
          awaitFlag_k = true;          // select critical section k
   }
od;
 if  //enter and execute a critical section
 ::(awaitFlag_1 == true) ->
     //stmts1
 :: (awaitFlag_2 == true) ->
    //stmts2
    ...
 :: (awaitFlag_k == true) ->
    //stmtsk
 fi;
```

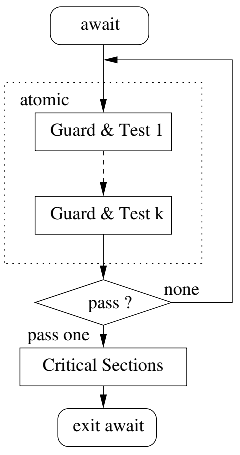


Figure 5.4: *Translation of an await statement*

which reduces the inherent complexity of the Promela program. With functional inlining, the verification becomes much more efficient regarding both time and memory usage. We use this as our standard translation method.

However, the translation approach without functional inlining is still useful as a debug mode, because it provides a detailed graphic trace that makes it much easier to trace function calls.

5.3.2 Interfaces and Await Statements

In MMM, an interface is used to define the I/O data ports of the process or medium and the I/O control points of the process or medium. To implement the control point, the MMM interface is used as a semaphore in the *setlist* and *testlist* of an *await* statement. We translate each interface into a pair of integer variables used as semaphores in Promela. The first variable, called *ACTIVE* is used to indicate whether the interface (and its member functions) are in active state (whether they are being executed). Another one called *EXCLUSIVE* indicates whether this interface semaphore is set (i.e. whether it is included in the *setlist* of some *await* statement that is currently executing). We use these variables as semaphores to signal that interface functions appearing in *testlist*'s are being executed and to prevent, when appropriate, interface functions appearing in *setlist*'s from being executed. Figure 5.4 illustrates how an *await* statement is translated in Promela. Promela constructs such as *atomic*, repetition *do-od* and case selection *if-fi* are utilized to guarantee the exact semantics equivalence. Specially, if the *await* statement has more than one critical sections that are enabled, one of them will be chosen non-deterministically and executed. This non-determinism is directly supported by Promela in *do-od* and *if-fi* statements.

5.3.3 Dynamic Objects

Another interesting aspect of MMM is the dynamic object (i.e. the reference type). For example, an array is a reference type in MMM, and its memory space could be allocated and changed dynamically at runtime. However, most model checkers (including Spin) only support static memory allocation, i.e. arrays have to be declared explicitly at design time. To solve the problem, we have to put some restrictions on the MMM code All the reference types have to be declared explicitly once and only once, so that they can be translated to Promela as static objects. An array declaration in MMM, "$int[]\ a\ =\ new\ int[12];$" can be translated to Promela as a static array "$int\ a[12];$". After the array a is declared in MMM, its reference cannot be changed any more. If the dimension of the MMM array is dynamic, e.g. "$int[]\ a\ =\ new\ int\ [n];$" where n is a variable, it is also translated to Promela as a static array "$int\ a[ARRAY\_MAX];$", where $ARRAY\_MAX$ is a constant set by the designer at compilation time. It is up to the designer to guarantee that $ARRAY\_MAX$ is always larger than or equal to the maximum value of n. Other dynamic objects such as class types in MMM are similarly translated to static data objects of Promela.

5.3.4 Function-Architecture Mapping

In MMM, the function of a system is specified as processes communicating through media. The architecture is represented as a set of media and mapping processes. Synchronization constraints are used to map the function to the architecture. To translate the function-architecture mapping, we need to use Promela to implement the MMM synchronization constraints that actually relate the function processes and the architectural mapping processes together. In Promela, we use a rendezvous channel (or synchronous channel) to synchronize two concurrent processes. An example of a synchronization constraint in MMM is as follows (see Figure 4.6):

"ltl synch(beg(P1, P1.write), beg(MapP1, MapP1.CPUWrite));",

The beginning of *P1*'s *write* and the beginning of *MapP1*'s *CPUWrite* are synchronized (both *write* and *CPUWrite* are function calls). In Promela, when *P1* and *MapP1* run to the points that need to be synchronized, one of them (e.g. *P1*) sends a synchronization signal to a rendezvous channel, and wait for the other process (i.e. *MapP1*). In this way, the events of the function processes and their corresponding mapping processes are synchronized and the mapping is realized.

5.4 Producer-Consumer Network

In this section, we present a set of case studies that consider a prototypical network of *m* producers and *n* consumers communicating through one or more media (see Figure 5.2 and 5.6). Producers receive inputs from the environment, process the data in some way, and then output it to a medium of a single space. Consumers read in information from that medium, process it, and then output to the environment. It is possible for all producers and consumers to execute concurrently. We verify constraints of the design before and after synthesis steps.

5.4.1 Verification of Data Integrity

Given a network of consumers and producers with one medium(see Figure 5.2), we want to check the constraint specified as formula (5.1) in Section 5.2, and we rewrite it here for convenience:

"Whenever a producer starts to write an item into the medium, there must be some space in the medium."

The constraint can be specified with LTL as:

$$\mathsf{G}\,((P_1\_write \vee \cdots \vee P_m\_write) \rightarrow M_1\_not\_full)\ , \tag{5.2}$$

where *write* indicates the condition that a producer initiates a write operation, and *not_full* indicates the condition that there is still some space in the medium.

Here we consider the case where m=2. This design has 102 lines of MMM source code and 670 lines of Promela code after translation. The formula is proved by Spin within one minute on a 1.5GHz Athlon machine with 1GByte of memory. The same setup is used for all case studies in this section. The detailed resource usage of this verification is listed in Table 5.1, where the states generated are the total number of unique global system states that are stored by the algorithm.

Another constraint we want to check is:

"When a consumer wants to read and there is no data in the medium and none of the producers has started to write, the consumer cannot finish reading until some producer starts to write."

This constraint can be specified with LTL as:

$$\begin{aligned} \mathsf{G}((&C_x\_start \wedge M_1\_empty \wedge \neg(P_1\_start \vee \cdots \vee P_m\_start)) \\ &\rightarrow ((\neg C_x\_end)\ \mathsf{U}\ (P_1\_start \vee \cdots \vee P_m\_start))) \qquad , \end{aligned} \tag{5.3}$$

where *start* indicates the condition that a consumer initiates a read operation or a producer initiates a write operation, and *end* indicates that they complete the operations. We consider the case where m=2. The constraint is verified by Spin within one minute of CPU time on the same machine. All relevant verification parameters are listed in Table 5.1.

Table 5.1: *Summary of verification for the producer-consumer network*

Constraint formula	(5.2)	(5.3)	(5.10)
Depth reached	26765	62839	1112111
States generated	120983	289828	1.49894e+07
State transitions	234014	561226	6.6521e+07
Total memory used	12.382 MB	42.584 MB (Partial Order Reduction)	101.626MB (Graph Encoding)
CPU time elapsed	<1s	1.49s	31m:54s

Finally, we want to prove the following constraint:

"If the consumers are able to keep reading data from the medium, then whenever a producer initiates a write, it will eventually complete the write":

In LTL, the constraint can be expressed as:

$$\mathsf{G}\,\mathsf{F}(C_1\_read \vee \cdots \vee C_n\_read) \rightarrow \mathsf{G}(P_x\_start \rightarrow \mathsf{F}P_x\_end) \;\; , \tag{5.4}$$

where *read* indicates the condition that a consumer completes a read operation, *start* indicates the condition that a producer initiates a write operation, and *end* indicates that the producer completes this write operation. Specifically, we consider the case where m=2, n=1 and x=1. Spin reports that the constraint does not hold. From the error trace using the debug mode (see Figure 5.5 [1]), we see that there is possibility of starvation. It is possible for P_2 to keep accessing the medium and prevent P_1 from ever be able to write.

[1]The numbers indicate the verification steps, and arrows indicate communications between processes through channels.

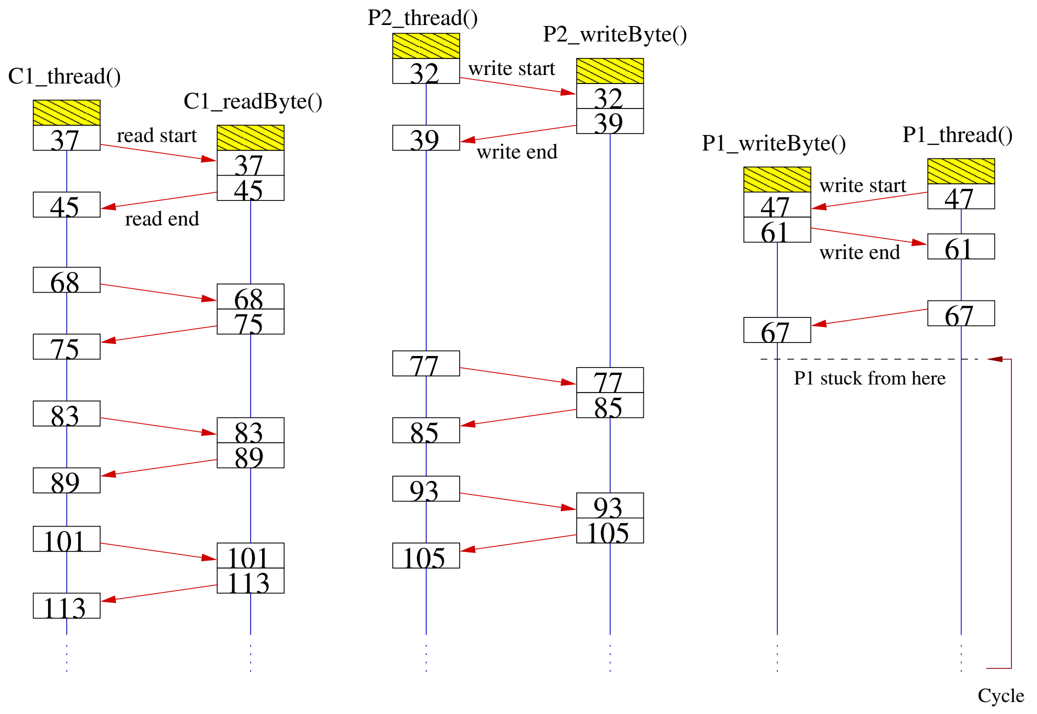


Figure 5.5: *Verification error trace produced by Spin*

5.4.2 Assumptions and Schedulers

In Metropolis, formal assumptions (specified with LTL or LOC) can be used to limit the possible behavior of a design. However, the downstream synthesis procedure must guarantee that assumptions are correctly implemented. If we want the constraint (5.4) (with m = 2, n=1, and x = 1) to hold, i.e. P_1 is not allowed to starve, we may specify the following assumption in the MMM design:

$$P_1\_start \wedge P_2\_start \longrightarrow \neg P_2\_end \; \mathsf{U} \; P_1\_end \;\; , \tag{5.5}$$

where *start* indicates the condition that a producer initiates a write operation, and *end* indicates that it completes the write operation. The assumption is trivially "translated" into Spin environment as the left-hand-side of an implication. Constraint (5.4) should be proved only for the cases where the assumption is satisfied. In other word, we prove the LTL formula:

$$Assumption\ (5.5) \rightarrow Constraint\ (5.4)\ . \tag{5.6}$$

Formula (5.6) is proved by Spin within one minute of CPU time.

In an architecture specification, assumption (5.5) can be implemented as a scheduler(or arbiter) that has a static-priority policy with P_1 having higher priority. After mapping the function to the architecture, We use Spin to prove that constraint(5.4) (with m =2, n = 1, and x = 1) holds in the presence of such scheduler. In addition, if we assign P_2 to have higher priority, the constraint fails. Another scheduling policy that can be proved to allow constraint (5.4) (with m =2, n = 1, and x = 1 or 2) to hold is round robin scheduling, where producers take turns accessing the medium.

5.4.3 Transformation and Refinement

Of course, system level synthesis procedures may not always be driven by the result of functional verification. For example, communication media may be combined to reduce the cost. MMM can be used to formally represent the design before and after a particular synthesis step. Consider the example in Figure 5.6. In (a), m media are used and producer-consumer

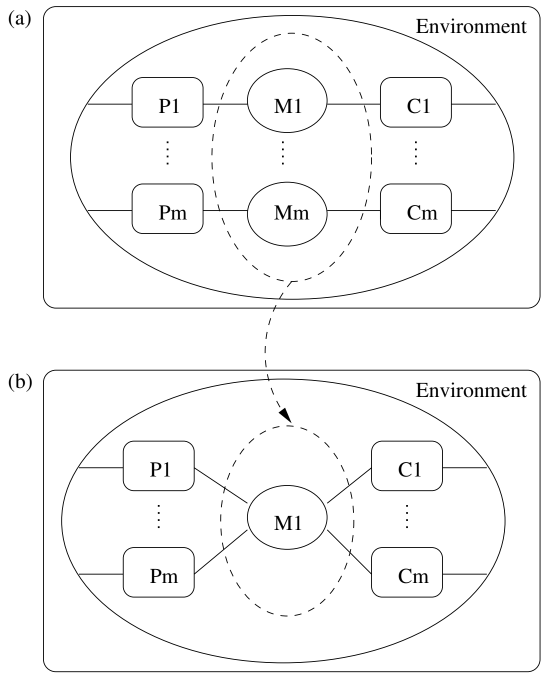


Figure 5.6: *Example of a design refinement*

data streams are running independently. It is trivial to verify that

$$\mathsf{G}(C_x\_start \wedge M_x\_empty \wedge \neg P_x\_start \rightarrow \neg C_x\_end \,\mathsf{U}\, P_x\_start) \;\;, \tag{5.7}$$

where $x = 1, \ldots, m$, *start* indicates the condition that a consumer initiates a read operation and a producer initiates a write operation, and *end* indicates that the consumer finishes its read operation. Now, let us consider Figure 5.6(b) where a single medium is used. It is derived from the network in Figure 5.6(a) through a structural composition. The constraint

$$\mathsf{G}(C_x\_start \wedge M_1\_empty \wedge \neg P_x\_start \rightarrow \neg C_x\_end \,\mathsf{U}\, P_x\_start) \tag{5.8}$$

is not guaranteed to be satisfied. Indeed, Spin verifies that the constraint does not hold within one minute of CPU time. The error trace shows that for the constraint to hold, an assumption must be added such that streams of data do not mix (i.e. if P_x write, then no consumer can read until C_x read):

$$\mathsf{G}(P_x\_write \rightarrow \bigwedge_{y \neq x} (\neg C_y\_read \,\mathsf{U}\, C_x\_read)) \ . \tag{5.9}$$

With these assumptions, the constraint may be verified by Spin using the LTL formula:

$$Assumption\ (5.9) \rightarrow Constraint\ (5.8) \ . \tag{5.10}$$

We verify the case where m=2. This design has 113 lines of MMM source code and 836 lines of Promela code after translation. The verification completes without error. Table 5.1 lists the detailed resource usage of the verification.

We also run a verification session with a dynamic scheduler of the following form: “if P_x writes, then no consumer can consume until C_x does”. As expected, the constraint is satisfied with similar complexity measurements. Through experimentation, we find that no round-robin scheduling nor any static priority real-time scheduler allow the constraint to pass.

5.5 Automatic Abstraction and Propagation

Working from a high abstraction level of a design, such as Metropolis Meta-Model, provides two pivotal advantages. First, an abstraction applied to a higher level specification will also make its lower level verification model more abstract as well. It is therefore advantageous to apply abstraction operations directly on the higher level model and simplify the higher level model as much as possible. Second, designers now have opportunities to specify abstraction operations on their own directly at a specification model according to their knowledge about the design.

It is obvious that only a portion of a design may be relevant to the passing or failing of a given constraint in constraint-based verification. The rest of the design may be simplified or removed, without changing the outcome of the verification. Unfortunately, identifying precisely what simplification or removal can be made correctly is as complex as the verification problem itself. Up until now, the process of design abstraction (i.e. the simplification of the design) is usually done by hand or left to the verification tools as they explore the reachable states and analyze the constraints. Based on these observations, we propose a technique of automatic design abstraction and propagation to simplify specification models and to lead to simpler verification models.

The automatic abstraction propagation consists of two separate operations, designer-driven propagation and constraint-driven propagation. Designers can specify free-able variables or statements according to their in-mind knowledge about the design, and then use the automatic propagation to exactly propagate them and abstract the entire design. Constraints being formally verified may themselves suggest an exact abstraction as well. The constraint-driven propagation can automatically free the variables and statements that are not relevant. No designer's interaction is required.

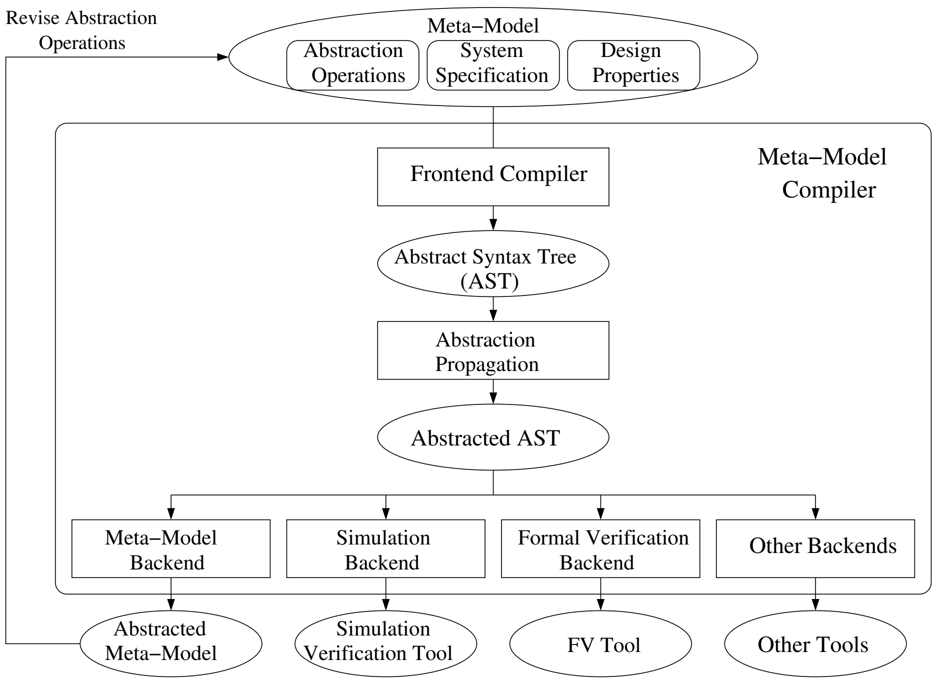


Figure 5.7: *Metropolis compiler architecture with abstraction propagation*

The implementation of automatic abstraction propagation in Metropolis is illustrated in Figure 5.7. If a regular verification session cannot complete or takes too much time to complete, a designer can turn on a compile-time flag to enable the abstraction, which can recognize the abstraction keywords and perform the automatic abstraction propagation to simplify the system design for verification. The abstraction propagation starts from the abstract syntax trees (ASTs), the intermediate representation of the Meta-Model language, uses on-demand traversal method to traverse the ASTs, and identifies the variables and statements that are eligible for abstraction according to the control and data dependencies in the design. Designers are allowed to specify more abstractions, and the tool will propagate them automatically to abstract the design as much as possible to speed up verification. The abstracted specification can then be verified by other verification tools more efficiently.

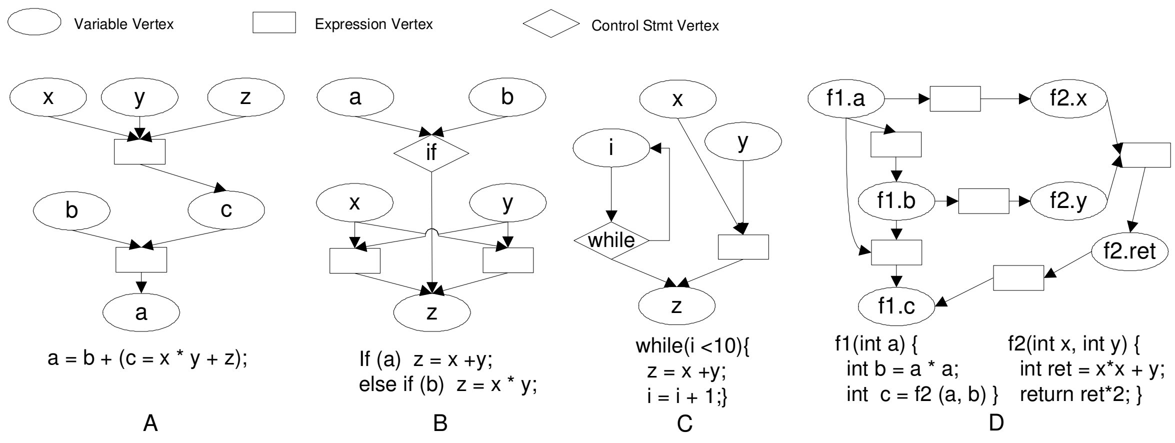


Figure 5.8: *CDDG examples*

5.5.1 Control and Data Dependency Graph

We use a control and data dependency graph (CDDG), built statically from the language syntax information of the original design, to automate abstraction propagation processes. The control and data dependency graph we use is a directed graph that has three types of vertices, representing variables, expressions and control statements respectively. More specifically, given an MMM specification, a CDDG is built according to the following general rules:

1) Each variable in the design corresponds to a vertex in the graph. For each assignment expression, there is a expression vertex to represent its right-hand side expression. For each variable in the right-side expression, there is an edge from the variable vertex to the expression vertex. There is also an edge from the expression to the left-side variable. Figure 5.8A shows an example of a complex assignment statement. Note that the operations on the variables are skipped and only dependencies are captured in a CDDG.

2) Each control statement is represented as a vertex in the graph. If a variable is in

the decision part of a control statement, there is an edge from the variable to the control statement; if a variable may change its value in the execution part of a control statement, there is an edge from the control statement to the variable. Control statements are further divided into three categories, branching statements such as *if* and *switch*, loop statements such as *while* and *for*, and synchronization statements such as *await* and *synch*. Figure 5.8B and Figure 5.8C shows examples of *if* and *while* statements respectively.

3) Function calls are generally treated as operations and expressions. The CDDG of a design specification is built as if all of its functions are flattened. Functions are connected together through passing arguments and returning values when they invoke each other. For a function, there is an expression vertex for each of its formal parameters and an expression vertex for the return value. There are edges between the variable vertices (in both invoking and invoked functions) and these expression vertices as variables are passed as arguments and return values are assigned to variables. As Figure 5.8D shows, function f_1 calls function f_2 by passing a and b as arguments and assigning the return value to c. So there are three expression vertices connecting the variables in two functions.

Note that vertices representing expressions don't contain any useful syntax information themselves and are only used to connect multiple variables to a variable or an expression as intermediate vertices. Though they are eventually removed to simplify the graph representation and traversal in the implementation, for the convenience of presentation, we still keep them in the illustrations. We define that a vertex v_j *depends* on a vertex v_i if there exists a directed path from v_i to v_j in a CDDG, where v_i and v_j represent either variables or control

Algorithm 7 *Designer-driven abstraction propagation*

```
1: $D' := D$
2: for each $v \in D$ do
3:     $D' := D' \cup \{u_i \in V$: there exists a path from $v$ to $u_i\}$
4: end for
5: Remove all the variables (including their operations) and statements in $D'$ from the design.
```

statements. In Figure 5.8A, variable a depends on variables b, c, x, y and z. In Figure 5.8C, variable i and the while loop depend on each other.

The number of vertices in a CDDG is the total number of variables, assignment expressions, formal parameters of functions and control statements. So the size of a CDDG is linear to the size of the original source code.

5.5.2 Abstraction Propagation Algorithms

Let $G = \{V,\ E\}$ be a control and data dependency graph that is built from a design specification, where V is the set of all the vertices and E is a set of dependency edges. In the designer-driven abstraction propagation, a designer can specify free-able variables and statements including variables and control statements, and automatically propagate them to the entire design. Assuming a set of variables and statements $D \subseteq V$ is chosen by the designer to start from for the designer-driven abstraction propagation, the algorithm is listed in Algorithm 7.

The algorithm searches for and then abstracts the variables and statements that depend on the designer's input in the entire specification. Using the example shown in Figure 5.8C, if a designer specifies that the while loop is free-able, then the whole while loop including the

Algorithm 8 *Constraint-driven abstraction propagation*

```
1: $L := P$
2: **for** each $v \in P$ **do**
3:     $L := L \cup \{$all the vertices that have a path to $v$ $\}$
4: **end for**
5: **for** each synchronization statement $s \in V$ **do**
6:     $L := L \cup \{s\}$
7:     $L := L \cup \{$all the vertices that have a path to $s$ $\}$
8: **end for**
9: Remove all the variables (including their operations) and statements in $V - L$ from the
   design.
```

variable i and the assignment statement of z will be totally abstracted and the abstraction can also be propagated to other variables and statements that depend on them. In the designer-driven abstraction propagation, the amount of false negative results due to the abstraction is decided solely by the designer's input. Its propagation is guaranteed to be exact and no false negative result will present as a consequence of the propagation.

Assume a set of variables $P \subseteq V$ is being checked in the constraints. The algorithm of the *constraint-driven abstraction propagation* is listed in Algorithm 8. The algorithm keeps what the constraints and synchronization statements depend on, and abstracts the rest of the design. Using the example shown in Figure 5.8C, if $P = \{x, y\}$, $V - L = \{z\}$, the code fraction is then abstracted to: "$while(i\ <\ 10)\ i\ =\ i\ +\ 1;$".

Note that the synchronization statements are not freed at this point even if they don't directly control the variables in the constraints. This is because a synchronization statement controls the execution of the processes in a concurrent system, and the complex interaction between processes make it difficult to free these synchronization statements exactly. The automatic abstraction propagation does not intend to handle the synchronization of concurrent

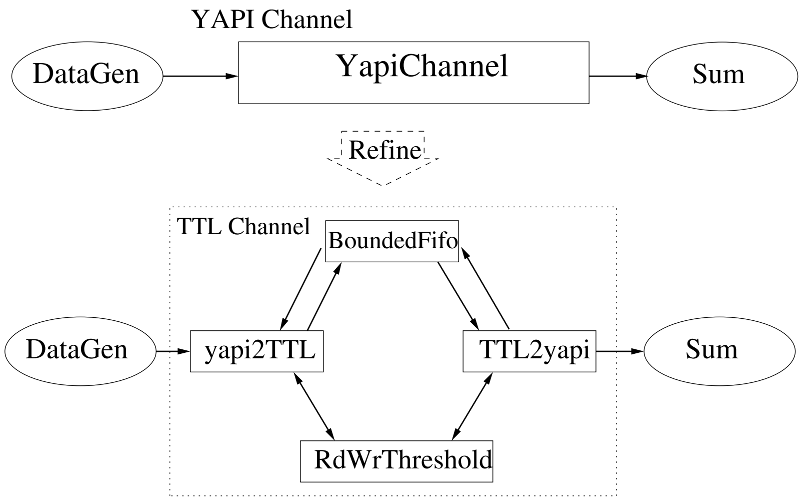


Figure 5.9: *YAPI and TTL channels*

systems and their abstractions are left to the designer by the designer-driven propagation. The algorithm also assumes that there are no non-terminating loops that may cause “dead” code.

Methodologically, the constraint-driven propagation should be applied first in the process of abstraction verification since it doesn’t need any interaction from the designer and will not introduce false negative results. Then a designer can apply several iterations of designer-driven abstractions to further abstract the design specification and simplify the verification problem as much as possible, even by introducing false negative results. The worst case for both algorithms is to traverse the entire CDDG $|V|$ times, so their complexity is $O(|V|^2)$ and they will introduce little overhead compared to the overall compilation time. The effectiveness of the automatic abstraction propagation we have proposed will be studied through a formal verification case study in the next section.

5.6 Formal Verification for TTL Channel

In this section, we use a realistic Metropolis design as an example to illustrate the usage of the formal verification mechanism in Metropolis and to demonstrate the effectiveness of the automatic abstraction propagation we have proposed in Section 5.5.

Y-chart Application Programmer's Interface (YAPI) is a popular model of computation for designing signal processing systems [49]. It is basically a Kahn process network [46] extended with the ability to non-deterministically select an input port to consume and an output port to produce. Within Metropolis, a library environment is set up such that any YAPI design can be written using constructs in the Metropolis library. Central to YAPI is the definition of communication channel and its refinement into Task Transition Level (TTL) [21, 32]. Figure 5.9 shows how a YAPI channel is refined to a TTL channel in Metropolis. A YAPI channel models an unbounded First-In-First-Out (FIFO) buffer, similar to Kahn process network. Asynchronously, writer processes write data into one end of the channel and reader processes read data from the other end of the channel. At the lower level (TTL), the channel is modeled with a bounded FIFO buffer. A central protocol is used to control the mutual exclusion and boundary checking of the bounded FIFO buffer. As Figure 5.9 shows, the TTL channel has a bounded FIFO ($BoundedFifo$) whose size is set at design time, and a control medium ($RdWrThreshold$) which implements a protocol to guarantee correctly writing to and reading from the FIFO buffer. To test the YAPI channel and its TTL refinement, we use a writer process ($DataGen$) to write a series of data into the channel and a reader process (Sum) to read the data from it.

Due to the boundedness of the TTL buffer, the writer process will block when there is not enough free buffer slots to write data, and the reader process will block when there is not enough data available in the buffer. The protocol implemented in the TTL channel controller($RdWrThreshold$) uses a threshold value to indicate if the writer or the reader can be unblocked. If there is a condition on which a process may be unblocked, the controller uses events *wakeup_reader* or *wakeup_writer* to signal unblocking. The detail of this algorithm can be found in [21]. The TTL channel model has 720 lines of code in Metropolis Meta-Model and about 2200 lines code in Promela after translation. The experiments presented in this section are all conducted with Spin 4.1.3 on a 3.0GHz Pentium 4 machine with 4GB of total memory.

5.6.1 A Deadlock Free Constraint

One important constraint we want to check on the TTL channel is that there should be no deadlock situation within the channel, i.e. once the writer starts writing data into the channel, it will finish writing eventually. This constraint can be specified as an LTL formula:

$$\mathsf{G}(datagen\_start \ \rightarrow \ (\mathsf{F}\ datagen\_finish)) \ , \tag{5.11}$$

where G is the *globally* operator, F is the *eventually* operator in LTL, and $\rightarrow$ is the Boolean imply operator.

Firstly, we try to verify a preliminary version of the TTL channel that contains a real bug causing a deadlock situation. Using Spin, the bug can be easily caught within less than

Table 5.2: *Summary of formal verification for TTL channel*

	Verification w/o abstraction	Manual abstraction	Designer-driven abstraction prop.	Constraint-driven abstraction prop.
state vector	432 bytes	352 bytes	232 bytes	188 bytes
depth reached	75607	74073	54359	33897
states generated	2.36686e+09	2.36607e+09	2.26572e+09	2.26481e+09
state transitions	3.65231e+09	3.60348e+09	3.42441e+09	3.54922e+09
memory usage	1094.545 MB	1091.66MB	1086.046 MB	1081.028 MB
CPU time usage	11h:48m:51s	10h:26m:24s	6h:41m:03s	5h:37m:24s
hash factor	3.62926	3.63046	3.79126	3.79278

*Optimization techniques, partial order reduction and bitstate, are applied.

one minute.[2] Then, after fixing the bug, we re-run the verification session and the revised TTL model can pass the formal verification without any error. The total CPU time used for the verification is a little less than 12 hours. Table 5.2 lists the details about the verification sessions for the non-deadlock constraint of the TTL model with and without abstractions and propagations applied.

Considering that the non-deadlock constraint only checks the control part of the TTL channel, its data-path can be abstracted to reduce the verification complexity. So we first manually free the data storages in both the writer process ($DataGen$) and the reader process (Sum) without using the automatic abstraction propagation. This abstraction saves about 12% of verification time, and requires modifying more than 10 statements throughout the original design. Then we use the designer-driven abstraction propagation to propagate these two abstractions to rest of the design. As a result, the internal data-path in the TTL channel is also abstracted and 43% of the verification time is saved.

[2]After the abstractions and their propagations are applied later, the bug in the preliminary TTL channel can also be caught within less than one minute. So the abstractions and their propagation are considered safe.

To show the effectiveness of the constraint-driven automatic abstraction propagation, we also apply it on the original design. It automatically frees not only the FIFO structure but also the buffers in other two connecting components ($yapi2TTL$ and $TTL2yapi$), which are directly connected to the FIFO, and their operations. From Table 5.2, we can see the constraint-driven abstraction propagation can save 52% of verification time without any human interaction.

Practically, the designer-driven and constraint-driven abstraction propagations complement each other and should be used together to simplify verification as much as possible.

5.6.2 Checking Data Consistency

When the writer *DataGen* writes a data into the TTL channel, it produces an event of $prepared$; when the reader *Sum* reads a data from the channel, it produces an event of $processed$. We use the annotation $data$ to represent the value of data written into or read from the channel. An important constraint that can be expressed with LOC is data consistency of the TTL channel, i.e. the input data of the TTL channel should be read from the channel in exactly the same order without a loss. The data consistency constraint is defined as:

$$data(prepared[i]) = data(processed[i]) \quad . \tag{5.12}$$

The TTL channel shown in Figure 5.9 is initially specified in Metropolis Meta-Model (MMM) [18]. From the MMM specification of the TTL channel design, we use the Metropolis backend tool to generate a corresponding Promela (Spin's modeling language) description [42], which can be verified by the model checker Spin for a particular LTL formula. The TTL channel design has 634 lines of MMM source code and 2049 lines of Promela code after translation.

From the discussion above, we know that the data consistency constraint (5.12) of the TTL channel cannot be expressed by LTL directly. Therefore, we have to assume that, "after the *x*-th write by *DataGen*, at most 31 writes can be done before the *x*-th read by *Sum*". [3] Then we use arrays $prepared\_data[32]$ and $processed\_data[32]$ to store the recent 32 pieces of data written by *DataGen* and read by *Sum* respectively. We also use $prepared\_i$ and $processed\_i$ (which take values of 0 to 31) to keep the index of the most recent data in the arrays. The assumption is written in LTL as:

$$\mathsf{G}(prepared\_occur \rightarrow prepared\_i \neq processed\_i) \ , \tag{5.13}$$

and it is verified to hold by Spin. The data consistency constraint is written in LTL as:

$$\mathsf{G}(processed\_occur \rightarrow prepared\_data[processed\_i] = processed\_data[processed\_i]) \ . \tag{5.14}$$

Because $processed[x]$ always follows $prepared[x]$, the data consistency only needs to be

[3]This assumption is derived from the actual buffer size of the TTL channel.

Table 5.3: *Summary of formal verification for data consistency*

Formula	(5.13)	(5.15)
Depth reached	51257	57221
States stored ($\times 10^8$)	2.2431	2.3156
State transitions ($\times 10^8$)	2.85523	3.09726
Total memory (MB)	735.098	819.517
CPU time	1h37m55s	3h03m18s
Hash factor	4.78686	4.63699

checked when an instance of $processed$ is occurring. The formula:

$$Assumption(5.13) \rightarrow Constraint(5.14) \tag{5.15}$$

is also verified to hold by Spin.

With the bitstate technique [43], Spin verifies the formulas (5.13) and (5.15) using about 1.5 hours and 3 hours of CPU time respectively on our 1.5GHz Athlon machine with 1GByte of memory. And all the other relevant verification parameters are listed in Table 5.3. From this case study (compared to the case studies in Chapter 3 and Chapter 4), we can clearly see the tradeoff between the simulation trace checking and the formal verification. The simulation trace checking is usually much more efficient in terms of memory and CPU time usage, but its verification results totally depend on the design of test cases for simulation. On the other hand, the formal verification is more expensive but the results are more confident. Therefore, it should be used for small but important design modules like the TTL channel.

Chapter 6

Conclusions

In this thesis, we have presented a comprehensive and complete study on verification and analysis methodologies for system level designs and have mainly based our approaches on formal specification of design constraints. Both simulation and formal verification techniques have been discussed for system designs with functional and performance constraints. LOC (Logic of Constraints) and LTL (Linear Temporal Logic) are two main formal languages that we use for constraint specification. The contributions of this work are summarized as follows.

We have extensively studied the verification aspects of our quantitative constraint formalism, Logic of Constraints. We compare LOC with LTL, a popular functional constraint specification formalism, find that LOC has a different domain of expressiveness from LTL, and conclude that LOC can express important constraints that cannot be expressed with LTL. We have proposed two feasible verification approaches, simulation trace analysis and model checking for LOC. We use a set of case studies on these approaches to demonstrate their usefulness and effectiveness.

A simulation verification and analysis methodology has been proposed based on formal specification of design constraints, i.e. assertions. We apply our methodology on the Metropolis design framework and the network processor architecture simulator NePSim. LTL is used to express and verify functional constraints such as non-starvation and execution ordering, and LOC is used to specify quantitative performance and functional constraints such as latency, throughput, and data consistency. All these constraints can be checked with automatically generated trace checkers on simulation traces using small amounts of CPU time and memory. The ability of LOC to carry out performance evaluation at the system level also opens up design exploration avenue uncharted before. We therefore utilize LOC in the design exploration of dynamic voltage scaling techniques in the network processor model. Our approach is shown to be an efficient tool to help a designer choose an optimal configuration in a large design space, specially when the number of considered parameters is large and manual analysis of simulation results becomes tedious.

In addition to the assertion-based simulation verification, we have also proposed a deadlock analysis approach with built-in simulation monitors. We study deadlock problems in system level designs that include complex synchronization constructs and function-architecture separation and mapping. We discuss our deadlock analysis approach including a data structure called the dynamic synchronization dependency graph and an associated deadlock detection algorithm. We use two examples, a complex function model for video processing and a model of function-architecture mapping, to demonstrate the effectiveness and efficiency of our approach in deadlock analysis for system level designs.

For small but important designs or library modules that will be instantiated many times across different designs, it is possible and useful to exhaustively prove the desired properties at a high level of abstraction using formal verification techniques. We have therefore proposed a formal verification methodology for system level designs with the approach of automatic generation of verification models from design specifications. This methodology is unique in that it is able to operate at different levels of abstraction and to allow verification to drive the design process. In addition, system functions, abstract architectures, and mappings can all be verified. Integral to the methodology is a semantically correct translator from a system level language, Metropolis Meta-Model, to a software verification language, Promela. Case studies have been performed to show the power of such an approach both in terms of constraint verification driving synthesis and formal verification of designs before and after synthesis steps. In addition, automatic abstraction and propagation algorithms have been proposed to further simplified generated verification models.

Bibliography

[1] http://www. omg.org, object constraint language specification, 1997.

[2] http://www.eda.org/dcwg, quick reference guide for the design constraints description language, 2000.

[3] http://developer.intel.com/design/network/ixa.html, Intel IXP1200 network processor family hardware reference manual, 2001.

[4] http://www.open-vera.com, OpenVera assertions white paper, Synopsys, Inc., 2002.

[5] http://netlib.bell-labs.com/netlib/spin /whatispin.html, Spin manual, 2003.

[6] http://www.eda.org/vfv, PSL homepage, 2003.

[7] http://www.systemc.org, SystemC homepage, 2003.

[8] http://developer.intel.com/design/intelxscale, Intel XScale microarchitecture, 2004.

[9] http://www.intel.com/design/network/products/npfamily/ixp2400.htm, Intel IXP2400 network processor, 2004.

[10] http://www.intel.com/design/network/products/npfamily/ixp2800.htm, Intel IXP2800 network processor, 2004.

[11] http://www.itrs.net/common/2004update/2004update.htm, International Technology Roadmap for Semiconductors, 2004.

[12] http://www.nlanr.net, NLANR measurement and network analysis, 2004.

[13] http://www.cs.ucr.edu/ cadgroup/pac, Performance Assertion Checker homepage, 2005.

[14] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs - automatic generation of simulation checkers from formal specifications. *Technical Report, IBM Haifa Research Laboratory, Israel*, 2003.

[15] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. Wiley-IEEE Press, 1994.

[16] P. Alexander, C. Kong, and D. Barton. Rosetta usage guide. http://www.sldl.org. 2001.

[17] B. Alpern and F. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages*, 11(1):147–167, Jan. 1989.

[18] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. *Technical Report 2001/01 Cadence Berkeley Laboratories*, Nov. 2001.

[19] F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of International Workshop on High Level Design Validation and Test*, Nov. 2001.

[20] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4):45– 52, Apr. 2003.

[21] J. Brunel, E. A. de Kock, W. M. Kruijtzer, H. J. H. N. Kenter, and W. J. M. Smits. Communication refinement in video systems on chip. In *Proceedings of the* 7^{th} *International Workshop on Hardware/Software Codesign*, pages 142–146, 1999.

[22] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Standford University Press, 1960.

[23] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 9–14, 2000.

[24] E. Cerny, B. Berkane, P. Girodias, and K. Khordoc. *Hierarchical Annotated Action Diagrams: An Interface-Oriented Specification and Verification Method*. Kluwer Academic Publishers, 1998.

[25] A. Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(3):350–366, 1987.

[26] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Workshop on Logics of Programs*, pages 52–71, 1981.

[27] E. M. Clarke, O. G. Jr., and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[28] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[29] C. Eisner and D. Fisman. Sugar 2.0 proposal presented to the accellera formal verification technical committee. Mar. 2002.

[30] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., 1972.

[31] F. Fallah, P. Ashar, and S. Devadas. Simulation vector generation from HDL descriptions for observability-enhanced statement coverage. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 666–671, 1999.

[32] O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In *Proceedings of International Symposium on System Synthesis*, Oct. 2001.

[33] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proceedings of IFIP/WG6.1 Symposium on Protocols Specification, Testing, and Verification*, June 1993.

[34] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer-Verlag, June 1993.

[35] A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–377, 1969.

[36] T. Hafer and W. Thomas. Computational tree logic and path quantifiers in the monadic theory of the binary tree. *Proceedings of International Colloquium on Automata, Languages, and Programming*, July 1987.

[37] Z. Har'El and R. P. Kurshan. Software for analysis of coordination. In *Proceedings of the International Conference on System Science*, pages 382–385, 1988.

[38] J. P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 1988.

[39] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[40] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 404–413, June 1995.

[41] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[42] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–258, May 1997.

[43] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, 13(3):289–307, Nov. 1998.

[44] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison Wesley, 1979.

[45] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, pages 890–904, 1986.

[46] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, pages 471–475. North Holland Publishing Company, 1974.

[47] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, Dec. 2000.

[48] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[49] E. d. Kock, G. Essink, W. Smits, P. v. d. Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: application modeling for signal processing systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[50] M. Krishnamurthi, A. Basavatia, and S. Thallikar. Deadlock detection and resolution in simulation models. In *Proceedings of the 26th Conference on Winter Simulation*, pages 708–715. Society for Computer Simulation International, 1994.

[51] T. Kropf. *Introduction to Formal Hardware Verification*. Spinger-Verlag, 1998.

[52] Y. Luo, J. Yang, L. Bhuyan, and L. Zhao. NePSim: A network processor simulator with power evaluation framework. *IEEE MICRO, special issue on network processors*, Sept. 2004.

[53] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems: Specification. *Springer-Verlag*, 1992.

[54] E. J. McCluskey. *Logic Design Principles*. Prentice Hall, 1986.

[55] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[56] A. Mok and G. Liu. Early detection of timing constraint violation at runtime. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 176–186, Dec. 1997.

[57] A. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *Proceedings of Real-Time Technology and Applications Symposium*, pages 252–262, June 1997.

[58] J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, 1983.

[59] A. Pnueli. The temporal logic of programs. In *Proceedings of the* 18^{th} *IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[60] M. Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine*, 15(2):8–23, 2001.

[61] M. Sfinghal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37–48, 1989.

[62] F. Vahid and T. Givargis. *Embedded System Design: A Unified Hardware/Software Approach*. John Wiley & Sons, 2002.

[63] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *Logics for Concurrency. Structure versus Automata, LNCS Vol 1043, Springer-Verlag*, pages 238–266, 1996.

[64] P. Wolper. Temporal logic can be more expressive. *Information and Control*, (56):72–99, 1983.

Appendix A

Formal LOC Syntax and Semantics

A.1 Representing System Behaviors

We use the term *behavior* to denote the sequence of inputs and outputs that a system exhibits when excited by the input sequence. In general, we want to consider both finite and infinite sequences, as well as hybrids where some inputs or outputs appear infinitely many times, and some appear only finitely many times. Formally, let E be a set of *event names*[1] and for each $e \in E$ let $V(e)$ be its *value domain*. Then, a *behavior* β is a partial function from $E \times \mathbb{Z}$ to $\bigcup_{e \in E} V(e)$ such that:

(1) $\beta(e, n) \in V(e)$ for each $e \in E$, and each positive integer n for which $\beta(e, n)$ is defined,

[1] In this work, we assume that E is finite. However, the approach presented here could easily be extended to arbitrary sets of event names. This extension would allow us to consider networks with dynamic process and interconnection creation.

(2) if $\beta(e, n)$ is not defined for some $e \in E$ and positive integer n, then $\beta(e, m)$ is not defined for any $m > n$.

(3) $\beta(e, n)$ is not defined for any $e \in E$ and any $n \leq 0$. [2]

If n is the largest integer for which $\beta(e, n)$ is defined, then we say that there are n *instances* of e in β. We also say for all positive integers $k \leq n$ that $\beta(e, k)$ is the value of the k-th instance of e in β.

A *system* is specified by a set of event names, their value domains and a *set of behaviors*. In a typical system, event names may represent interconnections, e.g. wires in a hardware system, or mailboxes in a software system. The behavior of the system is then characterized by sequences of values observed on the wires, or sequences of messages to mailboxes.

Behaviors, by themselves, are not sufficient to evaluate performance constraints that may involve quantities like timing or power of the system. For this, we need additional information regarding performance measures. We represent this information as annotations to behaviors. Formally, given an arbitrary set T, an *annotation* of behavior β of type T is a partial function f from $E \times \mathbb{Z}$ to T, such that $f(e, n)$ is defined if and only if $\beta(e, n)$ is. We refer to f as a T-valued annotation of β. Similarly to events, if f is a T-valued annotation, then we say that T is the value domain of f. An *annotated behavior* is a pair (β, A) where β is a behavior and A is a set of annotations of β.

[2] Clearly, we could have defined β as a partial function on positive integers, but this definition happens to be more convenient when we define the semantics.

Here we show a few uses of annotations, but make no proposal for their specification. We assume that they are part of the functional specification, and thus specified with the same language as the functional specification. In a way, they are an extension of an already common design practice, where comments and assertions are placed in the code to ease design understanding and debugging.

Annotated behaviors are structures for which we want to state constraints. In other words, annotated behaviors are models of LOC formulas.

A.2 LOC Syntax

LOC formulas are defined relative to a multi-sorted algebra $(\mathcal{A}, \mathcal{O}, \mathcal{R})$, where $\mathcal{A}$ is a set of sets (sorts), $\mathcal{O}$ is a set of operators, and $\mathcal{R}$ is a set of relations on sets in $\mathcal{A}$. More precisely, elements of $\mathcal{O}$ are functions of the form $T_1 \times \cdots \times T_n \mapsto T_{n+1}$, where n is a natural number, and $T_1, \ldots, T_{n+1}$ are (not necessarily distinct) elements of $\mathcal{A}$. If $o \in \mathcal{O}$ is such a function, then we say that o is n-ary and T_{n+1}-valued. Similarly, an n-ary relation in $\mathcal{R}$ is a function of the form $T_1 \times \cdots \times T_n \mapsto \{true, false\}$. We require that $\mathcal{A}$ contains at least the set of integers, and the value domains of all event names and annotations appearing in the formula. For example, if $\mathcal{A}$ contains integers and reals, $\mathcal{O}$ could contain standard addition and multiplication, and $\mathcal{R}$ could contain usual relational operators $(=, <, >, \ldots)$.

The basic building blocks of LOC formulas are *terms*. We distinguish terms by their value domains:

- i is an integer-valued term,
- for each value domain $T \in \mathcal{A}$, and each $c \in T$, c is a T-valued term,
- if τ is an integer-valued term, $e \in E$ is an event name, and f is a T-valued annotation, then val$(e[\tau])$ is a $V(e)$-valued term, and $f(e[\tau])$ is a T-valued term,
- if $o \in \mathcal{O}$ is a T-valued n-ary operator, and $\tau_1, \ldots, \tau_n$ are appropriately valued terms, then $o(\tau_1, \ldots, \tau_n)$ is a T-valued term.

We say that τ in a term of the form val$(e[\tau])$ or $f(e[\tau])$ is an *index expression*.

Terms are used to build LOC formulas in the standard way:

- if $r \in \mathcal{R}$ is an n-ary relation, and $\tau_1, \ldots, \tau_n$ are appropriately valued terms, then $r(\tau_1, \ldots, \tau_n)$ is an LOC formula,
- if ϕ and ψ are LOC formulas, so are $\overline{\phi}$, $\phi \wedge \psi$, and $\phi \vee \psi$.

For example, if a and b are names of integer-valued events, and f and g are integer-valued annotations, then the set of LOC formulas includes the following:

$$\text{val}(a[i]) = 5 \ \wedge \ \text{val}(a[i+1]) = 5$$

$$f(a[i+4]) + f(b[g(a[i])]) < 20$$

$$\overline{\text{val}(a[i]) = 0} \ \vee \ f(b[i]) = 0 \ .$$

When reading these formulas, it is helpful to think of i as being universally quantified, as clarified in the LOC semantics next.

A.3 LOC Semantics

We first define the *value* of formulas and terms with respect to an annotated behavior and a value of the variable i. We use a special symbol $undef$ to denote that the value of a term or a formula is not defined, and assume that $undef$ is distinct from any element of any sort in $\mathcal{A}$. We use $\mathcal{V}^n_{(\beta,A)}[\![\alpha]\!]$, where α is a term or a formula, to denote the value of α evaluated at the annotated behavior (β, A) and the value n of variable i. If α is a T-valued term, then $\mathcal{V}^n_{(\beta,A)}[\![\alpha]\!]$, is in $T \cup \{undef\}$, and if α is a formula, then $\mathcal{V}^n_{(\beta,A)}[\![\alpha]\!]$ is in $\{true, false, undef\}$. Note that this implies that for some k-ary T-valued operator o, the formula $o(\tau_1, \ldots, \tau_k)$ can take value $undef$, while o itself cannot, because it is T-valued. There is no contradiction here, only a slight abuse of notation, as we use the same symbol o to represent both the operator and its name appearing in LOC formulas. This ambiguity in the meaning of o, can always be easily resolved from the context in which o appears. Also note that we do not make a requirement that all annotations appearing in the formula must be defined in A. For such undefined annotations, we use value $undef$. The value of an LOC formula is defined recursively as follows:

- $\mathcal{V}^n_{(\beta,A)}[\![i]\!] = n$,

- $\mathcal{V}^n_{(\beta,A)}[\![c]\!] = c$ for each element c of each value domain T,

- for each event name e and each integer-valued term τ,

$$\mathcal{V}^n_{(\beta,A)}[\![\text{val}(e[\tau])]\!] = \begin{cases} \mathit{undef} & \text{if } \mathcal{V}^n_{(\beta,A)}[\![\tau]\!] = \mathit{undef}, \\ & \text{or } \beta(e, \mathcal{V}^n_{(\beta,A)}[\![\tau]\!]) \text{ is not defined} \\ \beta(e, \mathcal{V}^n_{(\beta,A)}[\![\tau]\!]) & \text{otherwise,} \end{cases}$$

- for each annotation f, each event name e, and each integer-valued term τ,

$$\mathcal{V}^n_{(\beta,A)}[\![f(e[\tau])]\!] = \begin{cases} \mathit{undef} & \text{if } f \notin A, \\ & \text{or } \mathcal{V}^n_{(\beta,A)}[\![\text{val}(e[\tau])]\!] = \mathit{undef}, \\ f(e, \mathcal{V}^n_{(\beta,A)}[\![\tau]\!]) & \text{otherwise,} \end{cases}$$

- for each k-ary operator o, using v_j to denote $\mathcal{V}^n_{(\beta,A)}[\![\tau_j]\!]$ for each $j = 1, \ldots, k$,

$$\mathcal{V}^n_{(\beta,A)}[\![o(\tau_1, \ldots, \tau_k)]\!] = \begin{cases} \mathit{undef} & \text{if } v_j = \mathit{undef} \text{ for some } j, \\ o(v_1, \ldots, v_k) & \text{otherwise,} \end{cases}$$

- for each k-ary relation r, using v_j to denote $\mathcal{V}^n_{(\beta,A)}[\![\tau_j]\!]$ for each $j = 1, \ldots, k$,

$$\mathcal{V}^n_{(\beta,A)}[\![r(\tau_1, \ldots, \tau_k)]\!] = \begin{cases} \mathit{undef} & \text{if } v_j = \mathit{undef} \text{ for some } j, \\ r(v_1, \ldots, v_k) & \text{otherwise,} \end{cases}$$

- $\mathcal{V}^n_{(\beta,A)}[\![\overline{\phi}]\!] = \begin{cases} \mathit{true} & \text{if } \mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathit{false}, \\ \mathit{false} & \text{if } \mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathit{true}, \\ \mathit{undef} & \text{otherwise,} \end{cases}$

- $\mathcal{V}^n_{(\beta,A)}[\![\phi \wedge \psi]\!] = \begin{cases} \mathit{true} & \text{if } \mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathit{true}, \\ & \text{and } \mathcal{V}^n_{(\beta,A)}[\![\psi]\!] = \mathit{true}, \\ \mathit{false} & \text{if } \mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathit{false}, \\ & \text{or } \mathcal{V}^n_{(\beta,A)}[\![\psi]\!] = \mathit{false}, \\ \mathit{undef} & \text{otherwise,} \end{cases}$

- $\mathcal{V}^n_{(\beta,A)}[\![\phi \vee \psi]\!] = \begin{cases} \mathit{true} & \text{if } \mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathit{true}, \\ & \text{or } \mathcal{V}^n_{(\beta,A)}[\![\psi]\!] = \mathit{true}, \\ \mathit{false} & \text{if } \mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathit{false}, \\ & \text{and } \mathcal{V}^n_{(\beta,A)}[\![\psi]\!] = \mathit{false}, \\ \mathit{undef} & \text{otherwise.} \end{cases}$

We say that an annotated behavior (β, A) satisfies a formula ϕ, if $\mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathit{false}$ does not hold for any integer n.

Appendix B

Proof of LOC Verification Complexity

Before we present the proofs of Theorem 1 and Lemma 1 (in Section **??**), we need to define systems that we are dealing with. Formally, a finitely-valued finite-state system is a sextuple $(S, S_0, T, E, G, \alpha)$ where:

- S is the set of *states* that must be finite,
- $S_0 \subseteq S$ is the set of *initial* states,
- $T \subseteq S \times S$ is the *transition relation*,
- E is the set of event names such that for each $e \in E$ the value domain $V(e)$ is finite,
- $G : \{(e, v) : e \in E, v \in V(e)\} \mapsto 2^T$ is the *generation function*,
- *annotation axiom* α is an LOC formula that may refer to values of event in E, but also to some annotations. Value domains of all the annotations appearing in α must be finite.

We use $G(e)$ as an abbreviation of $\bigcup_{v \in V(e)} G(e, v)$. Intuitively, $G(e, v)$ is the set of transitions on which e is generated with value v.

An annotated behavior (β, A) is in the set of behaviors of the system $(S, S_0, T, E, G, \alpha)$ if it satisfies α, and there exists a (possibly finite) sequence of states $s_0, s_1, \ldots$ such that:

- $s_0 \in S_0$,
- $(s_{i-1}, s_i) \in T$ for all $i > 0$ for which s_i exists,
- for all $e \in E$, all transitions (s_{i-1}, s_i), and all positive integers n: if e is generated on (s_{i-1}, s_i) for the n-th time, then it must be possible to generate the value $\beta(e, n)$ on that transition, i.e. if it holds that:

  $$(s_{i-1}, s_i) \in G(e) \enspace ,$$

  $$n = \left|\{j : 1 \leq j \leq k, (s_{j-1}, s_j) \in G(e)\}\right| \enspace ,$$

  then the following must also hold:

  $$(s_{i-1}, s_i) \in G(e, \beta(e, n)) \enspace .$$

B.1 Proof of Theorem 1

We will reduce the *Post Correspondence Problem (PCP)* [44] to checking whether a finitely-valued finite-state system with LOC annotation axioms satisfies an LOC formula. Recall that a PCP instance is given by two ordered lists of strings, $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. The question is whether there is a sequence of integers $i_1, \ldots, i_k$ (all form 1 to n) such that strings $a_{i_1}a_{i_2}\ldots a_{i_k}$ and $b_{i_1}b_{i_2}\ldots b_{i_k}$ are the same.

We now describe the system used in the reduction. The states of the system are 4-tuples (i_a, j_a, i_b, j_b) where i_a and i_b range from 1 to n, j_a ranges between 1 and the length of the string a_{i_a}, and j_b ranges between 1 and the length of the string b_{i_b}. Initial states are those where $j_a = j_b = 0$. In addition, there is a special state denoted by $DONE$. The system has two events a and b, both valued from 0 to n. Informally, the system moves into (i_a, j_a, i_b, j_b) after it sees the j_a-th letter of a_{i_a}, which must also be the j_b-th letter of b_{i_b}. Formally, the transitions in the system are the following:

- From $(i_a, j_a - 1, i_b, j_b - 1)$ to (i_a, j_a, i_b, j_b) if j_a-th letter in string a_{i_a} is the same as j_b-th letter in string b_{i_b}. If $j_a = j_b = 0$, then event a with value i_a and event b with value i_b are generated. Otherwise, no events are generated on this type of transitions.

- From $(i_a, j_a - 1, i_b, j_b)$ to $(i_a, j_a, i'_b, 1)$ if b_{i_b} has j_b letters, and j_a-th letter in string a_{i_a} is the same as the first letter in string $b_{i'_b}$. A b event with value i'_b is generated on this type of transitions.

- From $(i_a, j_a, i_b, j_b - 1)$ to $(i'_a, 1, i_b, j_b)$ if a_{i_a} has j_a letters, and j_b-th letter in string b_{i_b} is the same as the first letter in string $a_{i'_a}$. An a event with value i'_a is generated on this type of transitions.

- From (i_a, j_a, i_b, j_b) to $(i'_a, 1, i'_b, 1)$ if a_{i_a} has j_a letters, b_{i_b} has j_b letters, and the first letters in strings $a_{i'_a}$ and $b_{i'_b}$ are the same. An a event with value i'_a, and a b event with value i'_b are generated on this type of transitions.

- From (i_a, j_a, i_b, j_b) to $DONE$ if a_{i_a} has j_a letters, and b_{i_b} has j_b letters. Events a and b are generated on this type of transitions, both with value 0.

The system has a single binary annotation called $good$, and the annotation axiom is:

$$good(a[i]) \iff (\text{val}(a[i]) = \text{val}(b[i]) \wedge ((i = 1) \vee good(a[i-1]))) \quad .$$

PCP has a solution if and only if the system above does not satisfy the LOC formula:

$$\overline{\text{val}(a[i]) = 0} \quad .$$

Indeed the formula above is violated if and only if there is a path in the system from some initial state to $DONE$, such that along this path a and b are generated the same number of times (say $k+1$), and the first k values of of a and b are not only equal but also larger than 0. If $i_1, \ldots, i_k$ denotes those values, then it is not hard to check that strings $a_{i_1} a_{i_2} \ldots a_{i_k}$ and $b_{i_1} b_{i_2} \ldots b_{i_k}$ are the same.

We have just shown that PCP can be reduced to checking whether a finitely-valued finite-state system with LOC annotation axioms satisfies an LOC formula. Since the former is known to be undecidable, it follows that the latter is also undecidable.

B.2 Proof of Lemma 1

In this section we define the Presburger formula SYS_I whose existence was claimed by Lemma 1. We do so in several steps. First, we characterize the transition relation with formulas $TRAN_{sq}$ for each pair of states (s, q). These formulas have free variables t_{pr}, one for each transition $(p, r) \in T$. We construct $TRAN_{sq}$ such that an assignment $t_{pr} = n_{pr} \in \mathbb{Z}$ satisfies $TRAN_{sq}$ if an only if there is a path in T from s to q that crosses transition (p, r) exactly n_{pr} times. We set:

$$TRAN_{sq} = FLOW_{sq} \wedge CONN_s$$

Formula $FLOW_{sq}$ requires that the number of times a path enters the state must be equal to the number of times it leaves the state. The exceptions to this rule are states s, which must be exited one extra time, and q, which must be entered one extra time. Formally:

$$\begin{aligned} FLOW_{sq} = & \bigwedge_{(p,r)\in T} (t_{pr} \geq 0) \\ & \wedge \bigwedge_{r\in S} \Big(\sum_{(p,r)\in T} t_{pr} + Ind_{r=s} = \sum_{(r,w)\in T} t_{rw} + Ind_{r=q} \Big) \ , \end{aligned}$$

where Ind_P is 1 if proposition P holds, and it is 0 otherwise.

For example, for the system in Figure 2.2:

$$\begin{aligned}FLOW_{13} =&(t_{12} \geq 0) \wedge \cdots \wedge (t_{84} \geq 0)\\ &\wedge (t_{12} = t_{23} = t_{31} + 1 = t_{12} + t_{14})\\ &\wedge (t_{14} + t_{84} = t_{45} = t_{56} = t_{67} = t_{78} = t_{84}) \ .\end{aligned}$$

Unfortunately, $FLOW_{sq}$ is not sufficient to fully characterize paths from s to q. For example, the assignment $t_{12} = t_{23} = 1$, $t_{31} = t_{14} = 0$, $t_{45} = t_{56} = t_{67} = t_{78} = t_{84} = 2$ satisfies $FLOW_{13}$ but it does not describe a path from 1 to 3. Rather, it describes a path and a loop not connected to the path. To eliminate such loops, in addition to $FLOW_{sq}$ we must state that if $t_{pr} > 0$, then there must exist a simple path from s to p, i.e. there must exist a sequence $s_1, \ldots, s_{k-1}, s_k$ of no more than $|S|$ states, such that $s_1 = s$, $s_k = p$, and $t_{s_{i-1}s_i} > 0$ for all $i = 2, \ldots, k$. This is stated by formula $CONN_s$ which uses $|S|$ variables v_k to represent this path. Here, we assume that S is a subset of integers. This assumption can be made without loss of generality, as integer encodings can be easily defined for any finite set. If the path is of length $l < |S|$, we require that $v_k = p$ for all $k > l$. So, if the value of v_k is not p, we are still in the active portion of the path and we must require that $t_{xy} > 0$, where

x and y are values of v_k and v_{k+1} respectively. Formally, we define:

$$\begin{aligned}CONN_s = \bigwedge_{(p,r)\in T} (t_{pr} > 0) \Longrightarrow \exists v_1 \ldots \exists v_{|S|} : \Big(& (v_1 = s) \wedge (v_{|S|} = p) \\ & \wedge \bigwedge_{k=1}^{|S|-1} (v_k = p) \Longrightarrow (v_{k+1} = p) \\ & \wedge \bigwedge_{k=1}^{|S|-1} \overline{(v_k = p)} \Longrightarrow \big(\bigvee_{(x,y)\in T} (v_k = x) \wedge (v_{k+1} = y) \wedge (t_{xy} > 0) \big) \Big) \ .\end{aligned}$$

It may appear $TRAN_{sq}$ needs a term similar to $CONN_s$ stating that if $t_{pr} > 0$, there must exists a simple path from r to q, but in fact, this statement is already implied by the conjunction of $FLOW_{sq}$ and $CONN_s$.

For example, for the system in Figure 2.2:

$$CONN_1 = ((t_{45} > 0) \Longrightarrow (t_{14} > 0)) \wedge \ldots \ ,$$

implying that:

$$\begin{aligned}TRAN_{13} = & (t_{31} \geq 0) \\ & \wedge (t_{12} = t_{23} = t_{31} + 1) \\ & \wedge (t_{14} = t_{45} = t_{56} = t_{67} = t_{78} = t_{84} = 0) \ .\end{aligned}$$

In the next step, we use $TRAN_{sq}$ to characterize generation relation with formulas GEN_{sq} for each pair of states (s, q). These formulas have a free variables g_e for each event $e \in E$. We construct GEN_{sq} such that an assignment $g_e = n_e \in \mathbb{Z}$ satisfies $TRAN_{sq}$ if an only if there exists a path in T from s to q along which event e is generated exactly n_e times. It is not hard to see that :

$$GEN_{sq} = \underbrace{\exists \ldots \exists t_{pr} \ldots}_{\text{over all } t_{pr} \text{ s.t. } (p,r)\in T} : TRAN_{sq} \wedge \bigwedge_{e\in E} \Big(g_e = \sum_{(x,y)\in G(e)} t_{xy}\Big)$$

For example, for the system in Figure 2.2:

$$\begin{aligned} GEN_{13} = & \exists t_{12} \ldots \exists t_{84} : TRAN_{13} \\ & \wedge (g_{x_1} = t_{12} + t_{23} + t_{31}) \\ & \wedge (g_{x_2} = t_{45} + t_{56} + t_{67} + t_{78} + t_{84}) \ \ , \end{aligned}$$

which can be simplified to $(g_{x_2} = 0) \wedge (\exists j \geq 0 : g_{x_1} = 3j + 2)$.

So far, we have characterized a system independently of the LOC formula. Next, we will define SYS_I for a specific interpretation I of the set of event expressions $\mathcal{E}_\phi$. But first, we need to introduce some additional notation. In the rest of the section, we will use e_ϵ, a_ϵ, and b_ϵ to denote the event name and constants appearing in event expression ϵ, i.e. we will assume that every ϵ is of the form val$(e_\epsilon[a_\epsilon i + b_\epsilon])$ or $f(e_\epsilon[a_\epsilon i + b_\epsilon])$, where f is an annotation. We say that two event expressions ϵ and ϵ' are *similar*, and write $\epsilon \sim \epsilon'$, if they refer to the same event, i.e. $e_\epsilon = e_{\epsilon'}$ and they both refer to the value of e_ϵ, or they both refer to the same annotation of e_ϵ.

We say that an ordered tuple $(q_0, s_1, q_1, \ldots, s_N, q_N) \in S^{2N+1}$ is an *instance* of interpretation I of $\mathcal{E}_\phi$ if the following is satisfied:

(1) q_0 is an initial state, i.e $q_0 \in S_0$.

(2) (s_n, q_n) is a transition, i.e $\forall n = 1, \ldots, N : (s_n, q_n) \in T$

(3) There exists a partition $\mathcal{E}_1, \ldots, \mathcal{E}_N$ of $\mathcal{E}_\phi$ such that for all $n = 1, \ldots, N$ and all $\epsilon \in \mathcal{E}_n$ the following holds:

   (a) if ϵ is of the form val$(e_\epsilon[a_\epsilon i + b_\epsilon])$, then the event e_ϵ can be generated on transition (s_n, q_n) with the value required by I, i.e. the following holds:

$$(s_n, q_n) \in G(e_\epsilon, I(\epsilon)) \ ,$$

   (b) I assigns the same value to all similar event expressions in the same partition, i.e.:

$$\forall \epsilon' \in \mathcal{E}_n : (\epsilon' \sim \epsilon) \Longrightarrow \big(I(\epsilon') = I(\epsilon)\big) \ .$$

We call any such a partition an *instantiating partition* of instance $(q_0, s_1, \ldots, q_N)$.

Intuitively, by traversing a path visiting $(s_1, q_1) \ldots (s_N, q_N)$ we could generate all event values required by I. However, SYS_I must also ensure that these values are generated at correct values of index expression. To do so, SYS_I uses a variable y_{ej}, for each $e \in E$ and each $j = 1, \ldots, N$, to count how many times event e is generated on a path segment form

q_{j-1} to s_j. Formally:

$$SYS_I = \bigvee_{(q_0,s_1,\ldots,q_N)} \bigvee_{(\mathcal{E}_1,\ldots,\mathcal{E}_N)} \underbrace{\exists \ldots \exists y_{ej} \ldots}_{\text{over all } y_{ej} \text{ s.t. } e\in E, 1\leq j\leq N} : \bigwedge_{n=1}^{N} \Big(GEN_{q_{n-1}s_n}(\ldots, y_{en}, \ldots)$$
$$\wedge \bigwedge_{\epsilon\in\mathcal{E}_n} (\sum_{k=1}^{n} (y_{e_\epsilon k} + Ind_{(s_k,q_k)\in G(e_\epsilon)}) = a_\epsilon i + b_\epsilon) \Big) \ ,$$

where the first disjunction ranges over all instances of I, the second disjunction ranges over all instantiating partitions of the current instance, and $GEN_{q_{n-1}s_n}(\ldots, y_{en}, \ldots)$ denotes the formula obtained form $GEN_{q_{n-1}s_n}$ by substituting variables g_e with y_{en} for all $e \in E$. The equation requires for all $\epsilon \in \mathcal{E}_n$ that the total number of times that e_ϵ is generated on the path from the initial state to the transition (s_n, q_n) is exactly as required by the index expression $a_\epsilon i + b_\epsilon$.

For example, the interpretation I which assigns 1 both to val$(x_1[3i])$ and val$(x_2[i])$ in formula (2.17) has a single instance $(1, 3, 1, 8, 4)$ with the unique instantiating partition $\mathcal{E}_1 = \{$val$(x_1[3i])\}$, $\mathcal{E}_2 = \{$val$(x_2[i])\}$. Therefore:

$$SYS_I = \exists y_{x_1 1} \exists y_{x_1 2} \exists y_{x_2 1} \exists y_{x_2 2} : \big(GEN_{13}(y_{x_1 1}, y_{x_2 1}) \wedge GEN_{18}(y_{x_1 2}, y_{x_2 2})$$
$$\wedge (y_{x_1 1} + 1 = 3i)$$
$$\wedge (y_{x_2 1} + y_{x_2 2} + 1 = i) \big) \ .$$

One can check that:

$$GEN_{13}(y_{x_1 1}, y_{x_2 1}) = (y_{x_2 1} = 0) \wedge (\exists j \geq 0 : y_{x_1 1} = 3j + 2)$$

$$GEN_{18}(y_{x_1 2}, y_{x_2 2}) = (\exists j \geq 0 : y_{x_1 2} = 3j) \wedge (\exists j \geq 0 : y_{x_2 2} = 5j + 4) \ ,$$

so SYS_I can be simplified to $(\exists j > 0 : 5j = i)$, as we anticipated in Section 2.6.