Formal Verification Along with Design for Transactional Models

Jacob C. Chang

April 11, 2008

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

April 2008

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(David L. Dill) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mark A. Horowitz)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christos Kozyrakis)

Approved for the University Committee on Graduate Studies.

Abstract

Chip complexity has grown to a point where traditional verification techniques are growing difficult and expensive to use. Current research is looking into the use of formal methods along with changes in the design methodology to cope with the problem of implementing complex designs correctly. It is suggested that the formalization of abstract models from the start of the design process can find problem with the design early and reduce the cost of fixing bugs.

The key to successfully using formal verification during the early design stages is to recognize that different verification problems must be approached from different perspectives. Each different perspective answers the question of "Why is the design correct?" using a different reasoning technique. By using different models and tools for each different perspective, a set of perspective can capture the design intuition, thus the models can be made small enough so that they can be constructed, verified, and modified quickly. This methodology was applied to the verification of the Smart Memory Project during the design phase, and it has found corner case bugs early in the design process, reducing the cost of re-design compared to if the bugs were found later on.

One particular perspective that was used in verifying the Cache Controller of the Smart Memory is the transactional model perspective. The tool TDV (transactional diagram verifier) was developed to enable quick model construction and fast formal verification to be performed on models with transaction view. TDV is especially efficient in verifying designs which have problematic corner case situations caused by interacting parallel processes.

TDV verifies parallel transaction models by producing and verifying a set of verification obligations from the specification. The way the specifications are modeled and converted into verification obligations are well suited in verifying transaction model designs.

Acknowledgments

I would like to thank ...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Design Complexity

With the increasing complexity of design that is made available through better fabrication technologies, the problem of verification also becomes a bigger issue. Traditionally, verification is performed using simulation techniques and currently, it is still the primary way to verify design correctness. Techniques in simulation technology such as coverage models, guided test generation, and faster simulation speeds have helped by allowing simulation to zero in on bugs more efficiently. The use of formal methods also has various successes through advances in model checkers which allows for complete coverage on small modules or conceptual protocols. The various successes with formal methods have also allowed a hybrid formal and informal verification approach. Simulations now use formal methods to guide their verification or allow formal methods to replace simulation in certain parts of the verification process.

1.1.1 Current Research Directions

Despite these advances, there are still shortcomings to both verification methods as the design complexity increases. The complexity of the design, especially with the recent trend of parallel processing units, raises the possibility of missed corner cases that simulation technologies have trouble analyzing. Although formal verification is

well suited for finding corner case design bugs, current formal verification technologies are only powerful enough to verify small designs. It is still unknown how to apply formal verification consistently to larger designs or find corner cases that result from the interaction of multiple modules. Although there have been successes in using formal verification in verifying large high level designs, the verification methodology successes are time consuming and specific to a particular design.

The other usage of formal verification is for quick verification of protocol ideas before the design process. However, in these verification efforts, it is not clear how to tie the verified protocol models with the real implementation to ensure that the design actually implement the verified model.

There have been various proposals in the ESL community in finding ways to bridge the gap between the abstract level description and the implementation. One research area is to automatically generate RTL level system description [2]. Or, instead of automatically generating RTL code, formal verification can be used to verify that RTL corresponds to a higher level description such as SystemC [39, 20]. Others emphasize the importance of design methodology change so that helpful assertions may be inserted into the RTL code [33].

Bridging the gap between an abstract model and a concrete model is a problem recognized by the formal verification community even before ESL research has came onto the scene [23]. This is because formal verification requires abstract models in order to make verification possible, and it is suggested that the correspondence between the abstract models and implementation can be verified through refinement techniques. However, all these techniques only work well for layers that are close to each other in terms of abstraction level, and there is still no practical way to link formally solvable abstract model to the RTL implementation at this time.

Recent research has suggested that by changing the methodology of design, it is possible to better integrate the abstract models into the design [27]. By formalizing the behavioral models (as opposed to using patched up C/C++ simulators), Researchers hope that more of the abstract model design formalizations can be directly linked to the design. This would make the verification of abstract and implementation possible. Various tools and model techniques have been developed that allow

for the formalization of abstract design [3]. The next section will look at different design methodologies that have been suggested in literatures and examine how the current formal verification technology can be practically used with today's design environments.

1.2 Design Methodologies

1.2.1 Refinement Methodology

The idea of designing from the abstract model and successively add in the details has been the predominate model for programming until the rise of object oriented programming [40]. It is suggested that stepwise refinement of the program in design makes it more adaptable to changes that will occur as a result of an extension or correction to the program. Since these extensions or corrections may not involve changes to the abstract description, only the last design choices and decisions will have to be re-implemented. This idea is also known as the top-down design methodology.

At first glance, formal verification is well suited for this methodology because the difference between the refinement layers can be made small, and thus it is possible to verify the correspondence of these two layers through formal methods. Formal verification can also 100% coverage verification, thus any design layer can be made bug free before it is refined to the next level. This ensures that bugs will not be propagated down the refinement chain. One such project which uses refinement methodology to develop a router design while doing formal verification is demonstrated in a project by Creveuil and Roman [8]. And observation made by Creveuil and Roman in their project is that it is more helpful to formalize the design and come up with a plan for formal verification than to do the formal proof itself. Although I have made similar observation in my verification experience, my experience of dealing with an actual project demonstrates that formal verification through refinement layers does not work due to design changes that occurs in a real industrial design.

Another proposal for writing high level specification and verification by refinement is seen in the Pentium Pro arbitration logic verification project [37]. Although this

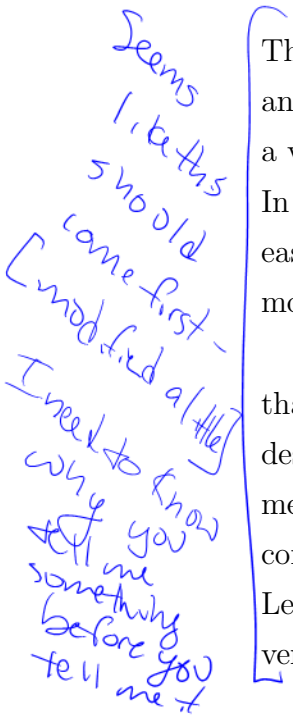
project was done on an industry design, this project is concerned about top-down verification after the design is complete rather than verification as part of the top-down design process.

So what is the point of this paragraph? Why is it here

The problem with using formal verification as part of the design process is that the designers of any project have many considerations besides the correctness of the design during the design process. The design often changes as the design progresses due to considerations of implementation efficiency and performance. This causes problems for doing formal verification using refinement techniques during the design process because it is a time intensive process to formalize an abstract design, formalize the properties of interest, and find techniques to reduce the design to a manageable level. Furthermore, formal verification efforts are sensitive to changes in the abstract design. Thus when the real design requires changes to the abstract model, the time intensive aspects of formal verification must be scrapped and reanalyzed [15]. Thus, using traditional formal verification techniques along side the design through refinement does not work because changes to the abstract level model during the design process make formal verification hard to keep up with the speed of the progression of designs.

Although it can be argued that by finding all the bugs on the abstract design, the abstract model will not need to be changed later, thus making refinement methodology possible. This concept of designing the abstract level design to completion before working on the more concrete model is ~~mentioned by~~ called the waterfall model in Royce's paper [31]. However, Royce argued that a design refined from an abstract model to a more concrete model without a mechanism for feedback is flawed since feedback from detail implementation is essential to produce a useful and implementable abstract design. Thus changes to the abstract model are going to be an essential part of the design process during the evolution of the design. Formal verification can prevent abstract model changes due to design bugs, but the abstract design can still change due to performance or implementability reasons which make formal verification on abstract model in the refinement style methodology impractical. The time consumption and the difficulty to incorporate new changes in formal verification is the reason why formal verification is applied generally toward designs that have been completed, and even in some cases, designs that is already in the fabs.

1.2.2 Design Methodology Motivation



The motivation for the refinement methodology is that it allows the design to be analyzed early in the design process, when the changes are less costly to make. From a verification perspective, it allows design problems to be found and corrected early. In addition, looking for bugs with an abstract design also means that design bugs are easier to trace to the source, and debugging is easier when it is done with the abstract model.

The advantage of early design verification is so important and advantageous such that that most design methodologies have explicit or implicit goal of achieving early design verification. The next two sections will describe how the current industry methodology uses simulators to achieve the goal of early verification and its short comings. And it will also describe the proposal put forth through Electronic System Level Design (ESL) to formalize the current ad-hoc design process to achieve better verification at the abstract level designs.

1.2.3 Current Industry Practice

In industry today, it is often that the designers create a C/C++ simulator of the design that captures some parts of the design for testing a concept and provides a platform for starting software development. This C/C++ simulator is implicitly an abstract model of the design that allows some design decisions to be verified before detail RTL is written. Just as there are different levels of abstraction that can be modeled in formal verification, simulators can also be written to model at different abstract levels in the design.

There are two issues with using C/C++ simulator to debug designs early in the design process. One is the issue of matching up the final implementation to the simulator. This is to verify that the final implementation is going to behave in the same way as the as the more verified (and more likely correct) simulator design. The problem is not only in verifying that they match, but also in maintaining the abstract model. In a typical design project, once the RTL is written, any bug fixes or changes tend to be reflected only in the RTL, but not the abstract simulator. This is because

it is at this point in the design process, the abstract version is no longer used since verification and simulation can be done on the RTL, thus the cost of maintaining a current version of the simulator seems to not be worth the effort. However, this prevents using the faster simulator to catch any bugs that may be introduced through the changes in the design.

The second issue is that simulator may still require too much detailed information to be able to be constructed early in the design process. This presents a problem for using formal verification on the abstract simulator model. C/C++ simulators usually contain programming details which unnecessarily blow up the state of the system under verification, making formal verification on C/C++ simulators impractical or not worth the effort compared with verifying it on the final implementation later in the design process.

The trade-off between these two issues can be made by writing the simulator at different abstraction levels. The designers can write a simulator early in the design process using an abstract model, but it offers fewer guarantees that the model will actually represent the final implementation. Or the simulator can be written so that it is cycle accurate and port accurate to the implementation, but usually this cannot be done until details of the design are finalized.

The goal of current research is to close this gap, that is, to link the abstract model with the implementation while giving the verification engineers the ability to start verification early.

1.2.4 Electronic System Level Design (ESL) Methodology

Historically, the ESL design methodology comes from the desire to create an executable design early in the design process so that software development may proceed at the same time. This means that abstract executable models must be developed to allow early creation of executable models. Being able to develop software at the same time as hardware design allows the requirements of the application to be better understood during the design process and allows enough time for the hardware to change according to its changing requirements.

Having an executable (and thus formalized) model is also appealing from a formal verification perspective. Because formal verification is incapable of dealing with large designs due to exponential scaling of the state space, creating a smaller abstract model representation of the design is often necessary for formal verification. It is for this reason that research are also being done of using formal verification within the methodology of ESL design [39].

Various languages and tools have been developed to allow for abstract executable models to be created. These ESL languages differ on abstraction levels as well as the target application. As an example, MATLAB is used to build an abstract representation that is often used in DSP and control applications. SystemC is meant to leverage off current industry practice of using C/C++ simulators by adding hardware centric concepts. SystemVerilog, Vera, Bluespec takes the opposite approach by adding more abstraction capability to RTL level languages, thus SystemVerilog operates at fairly low level design. SDL, UML, XML are also examples of high level abstraction descriptions languages.

The reason why these design languages and methodology have not been widely accepted in industry is because we see the same issues and trade-off as the C/C++ simulator models. Abstract models are good for quick modeling to give feedback for further direction in design, however, it is difficult to leverage the abstract model in producing RTL design, and once the RTL implementation is done, it is hard to verify that implementation corresponds to the abstract model and hard to maintain that abstract model because the gap between the two models is large. Even for lower level languages, the design it compiles from these languages are often not acceptable from a performance perspective.

The formal verification community have recognized the advantage of abstract design verification partly because there is no other way to perform formal verification on large designs [21]. However, the same shortcomings also plague the formal verification methodologies. We see that formal verification is very successful in verifying small, low level modules with respect to a local properties of interest, but it cannot handle large designs or designs with properties that are dependent on various modules. It is

also successful in verifying small high-level abstract description for prototyping purposes, but there is no assurance that the implementation will implement the verified abstract model correctly.

1.3 Formal Verification Methodologies

We explore the use of formal verification concurrently with the design by investigating the use of formal verification early in the design process in the Smart Memory project [18]. The following sections will discuss several observations on the issues involving doing concurrent design and verification based on our experience in this project. We will also suggest methodology concepts which allows for practical use of formal verification during the design process. The rest of the Thesis will describe the verification project using the suggested methodology for a class of verification problem. Specifically, a tool is developed to formally verify parallel transaction processes while allowing the verification effort to keep up with the design.

1.3.1 Methodology Observation: Formalization

Although it is argued that the waterfall model, which complete the abstract level design before working on the more concrete model, does not work practically. However, in the current industry practice and the proposed ESL methodologies, the refinement methodology in the waterfall model are still central to the design process. That is, abstract model is built (either formally in computer or informally in documentations) to verify and assess design decisions early in the design. Yet in all the methodologies, including simulator, ESL, and waterfall model, the problem of changing design specifications is still a major hurdle, which is what make the waterfall model impractical. This problem is especially pronounced in the use formal verification because the effort required for setting up formal verification is tremendous.

However, the waterfall model does reflect to the design process as it progresses from the abstract design to the implementation in the general sense. The changes that are made to the abstract design usually does not affect the design concepts, but

the changes to the *formalization* of the design concepts is what causes the need to re-design and re-analyze the abstract model. That is, the overall concept of how the design works and what is required for the correct operation usually does not change during the design process. But the formalization of the model involves taking the idea of the design and writing it down in a machine readable, unambiguous way. And this process usually constrains the design beyond the initial concept of the designers. The concept of formalization is different from the refinement process, where the details of how a design is implemented are specified.

In order to do any performance analysis or correctness analysis, an abstract design must be formalized by definition. The process of writing a C/C++ simulator, sketching the design out using MATLAB, UML or defining the interfaces are all part of the formalization process. Since this formalization is highly dependent on the implementability of the abstract design, the details of this formalization will invariably change when the more refined designs are constructed.

1.3.2 Abstract Formal Verification

There have been numerous projects which use formal methods to verify high level designs. Numerous papers have been published tackling different formal verification problems, and different methods of modeling have been proposed for these different problem under verying abstraction levels. For example, Oliveira and Hu stated that regular expressions and extensions of regular expressions are efficient ways to model communication protocol over bus [24]. The use of formal specification languages such as TLA are geared toward verifying abstract specifications [7, 1], although the amount of work in formalizing and setting up the verification of these abstraction specifications are very time intensive. Transaction Level descriptions are well suited for formal verification on lower level designs for detecting hazards [41, 17]. Our experience and the experiences of Chen, Yang, et. el. [5] show that that using assume/guarantee reasoning is a good way to verify protocols in a hierarchical manner.

This leads to the observation that different problems or different abstraction level descriptions require different ways of modeling in order to make formal verification

possible. That is, there is no one modeling technique or tool that can be used in order to solve the formal verification problem for the different types of design at different abstraction levels.

1.3.3 Proposal

To summarize the difficulties and observations in performing verification along with design, we propose that verification along with design methodology should have the following properties:

- Top-down design: The methodology should capture the advantage of the top-down design; namely the ability to catch design bugs early in the design process where the cost of fix is small.
- Design Intuitions: The implicit abstract models that are used by the designers (although it might not be formalized) should be captured by the methodology process to aid the verification efforts.
- Light-weight formal verification: Any formalization and verification of abstract design needs to be able to be done quickly so that any changes to the formalization can be re-verified without wasting much effort.
- Light-weight modeling: Formalize as little details as possible to minimize re-verification due to design changes. To accomplish this, the methodology should formalize only what is necessary for showing the correctness of design on conceptual level for the property of interest.
- Formal Verification Perspectives: Allow the use of different verification techniques and tools for different verification problems at different levels of abstraction.

The basis of our proposal is to perform formal verification along with design early in the design process. There are two points that make the proposed methodology different from the previous proposals. One is having the verification engineer work along

with the designers during the design process as opposed to bringing the verification engineer later in the design process or having the designers write their own abstract models. The second is to allow informal descriptions of the models to be constructed and verified formally.

The role of the verification engineer early in the design process is to understand the design and the motivation of the designers so that the model may be properly chosen to represent and reflect the designer's priorities. Performing formal verification on the model will also allow the verification engineer to give feedback to the designers to help them avoid pitfalls in their design that will cause unforeseen incorrect corner cases. Part of the interaction between the verification engineer and the designer is to understand the design so that the areas where formal verification will have the most impact can be identified. Separating out formal verification into smaller problem domains while letting traditional verification methodologies verify the rest of the design is the key to prevent monolithic formal verification problems that are too time intensive to be done during the design process.

However, in order for the verification engineer to be able to give feedback in a prompt manner, the verification must be performed quickly. Allowing for informal descriptions allows quick verification on the areas which formal verification has the most benefit. By informal description, I mean that the model can be built without consideration about matching the model to the real implementation. Our experiences have shown that such a model can be built and verified within a few days. Once the design progresses, if the match of abstraction model to the implementation becomes a worry to the designers, then further refinement or formalized correspondence can be established. By allowing quick informal descriptions to be verified, it allows corrections to be done early in the design process.

We apply this methodology to the cache system of the Smart Memory Project and demonstrate the use of formal verification during the design phase of the project. It is shown through this methodology application that formal verification can be practically used to find bugs early in the design process even though the verification may not be completely formalized. This methodology was applied to various design phases of the project at various abstraction levels including the verification of the deadlock

free property, the cache coherence protocol based on MESI, the implementation cache controller with respect to transaction rules, the SyncOp protocol, and the execution of the cancel command.

The verification model for these projects varied from informal, but systematic examination of the design, to formalized verification through model checker and theorem prover. One such verification effort, the verification of the cache controller architecture design, led to the development of a verification tool, TDV, which specializes in the verification of interrupt free properties in a parallel process environment.

1.4 TDV Tool Development

Building on the observation that different problems in verification require different perspective of formal verification model and tools, we developed the tool Transaction Diagram Verifier (TDV) for a few verification problems encountered in the Smart Memory Project. TDV is designed to verify design which can be easily modeled into transactions where each transaction is modeled as an abstract state diagram. It is well suited for problems with simple single transaction execution, but may have unforeseen corner cases when multiple transactions are occurring at the same time in the system.

Although this verification problem can possibly be checked by a model checker, the TDV tool contains several pre-verified logical reasonings that the model checker cannot normally reason about. This gives the advantage that our design methodology requires. The TDV model applies constraints to the input model so that an efficient verification scheme can be used to perform verification. The logical reasoning of why a design should work is built into the tool development process, thus if the verification problem fits within this tool's model framework, the validity of the verification approach will not have to be repeated.

1.5 High-level to Low-level Verification Gap

At the later stages where the formalization of the abstract protocol may not require much change, a formal abstract model with a more complete verification may be attempted. However, the problem with making sure that the abstract model is representative of the real design still exists. We propose a technique to verify the correspondence between the implementation and its abstract model, by linking and co-simulating the abstract model along with the implementation. At each simulation step, a set of abstraction functions translate the concrete state to the abstract state, and transition between the previous and current state in the abstract model is verified.

This methodology is used to verify the coherence and synchronization protocols in the Smart Memories' multi-processor system. During this process, implementation problems and abstract model inadequacies were revealed. Although this technique was done before on the Alpha 21364 EV7 processor [38], we apply the methodology during the design process as opposed to waiting until after the design has been completed. This gives the advantage that bugs are found early in the design process and allows for correcting the design when the cost is low. In addition, we are able to give feedback to the designers on how to correct their design to avoid corner case bugs in the future.

1.6 Smart Memory Project

All the verification examples using TDV in this case studies are from a real world design of the Smart Memory Project [18]. In particular, TDV verification contributed to the verification to the the different sections and aspects of the memory system of the project. Since the CPU core of the project is an tensilicia core, the verification of the CPU is not a concern to the people in this team.

Smart Memory is a modular computer with array of processor tiles and memory mats. The tiles are connected with high-speed links and the memory can be configured to support various memory models depending on the application. Each tile contains

two CPU cores and 8 configurable memory mats.

A *quad* is a collection of 4 tiles. Each quad have a cache controller which can interact directly with the memory mats in each tile as it receives commands from each processor or the memory controller. The memory controller controls the interface between different quads and gives access to the main memory.

1.6.1 Cache Coherence Mode

When the memory mats are configured in cache coherence mode, the MESI protocol is used to maintain communication between different processors [28]. When a memory request is initiated, the cache controller is responsible to bring in the the cache line while maintaining the MESI invariants. In order to do so, the cache controller performs a series of operations on the memory mats and requests to the memory controller. These operations may include probing the state of a cache line in the memory mat, changing the status of the cache line, moving or copying the data content of the cache line, and/or request to memory controller to initiate operations on the other cache controllers.

The memory mats are configured such that each cache line have a tag bits associated with the cache line. Tags of the cache line identify whether the cache line is in the invalid(I), shared(S), exclusive(E), or modified(M) state corresponding to the meaning in the MESI protocol. In addition, the tag may also be in the reserved(R) state to indicate that the cache location have invalid data, but it is being reserved by the cache controller so that the cache line location will be available after it completes fetching the data from another location.

1.6.2 Streaming Mode

When the memory mats are configured in the streaming configuration, the cache controller's DMA engine is responsible for copying the data from one memory mat to another memory mat. The control for data copying is controller by the software, thus the hardware support for streaming mode is relatively simple.

1.6.3 TCC mode

In the TCC configuration, the memory mats are configured to support transaction memory coherence and consistency model [13]. The TCC model provides the execution of transactions in an atomic manner in a parallel environment. It does so through speculative execution of a transaction, when the transaction is completed it will commit the execution changes to the system. If, however, the speculative execution is in violation with another process, the transaction will roll back to the beginning of the transaction execution and try again.

Chapter 2

Perspective Based Verification

This chapter will describe the idea of perspectives and how using a perspective approach will allow formal verification to be used early in the design process.

2.1 Perspectives

A perspective is a lightweight, partial formalization of a design that captures the abstract models from the designers with respect to one property class. The idea of perspective based verification is that instead of finding and developing one formal verification technology to apply to all formal verification projects, the designers and the verification engineers should divide the verification problems into property classes and choose different formal modeling techniques and tools for verifying the properties in those property classes. The following sections will describe the characteristics of a perspective in detail as well as further develop the notion of what is a perspective.

2.1.1 Abstract models from designers

During the early design process, the designers typically have an unstated high level model that they consider and analyze. This could be in the form of whiteboard sketches, informal design descriptions, or simply ideas that are still in the minds of the designers. The idea of perspective based verification is to capture enough details

about these high level informal models to analyze the correctness of these ideas while maintaining the brevity of the high level model. Building models at this point in the design stages allows for formalization of the designer's abstract view of the design.

In addition to the informal design model, the designers also have an idea of why the design is correct at the concept design stage. By choosing the correct perspective, the designer's reasoning of the correctness of the design can be used as part of the reasoning process in doing formal verification on the model.

2.1.2 With respect to one property class

Each perspective captures the high level model and correctness of design intuition using different theories. Since the different theories are not necessarily compatible, perspective based verification is useful only when one property class is considered at a time. A property class is a set of properties that

1. Describes the property on the same level of abstraction
2. Can be concisely expressed by the same modeling technique
3. Have the same reasoning of why the property is correct

By restricting the verification scope of each perspective, the correct tools can be identified and used efficiently for each verification problem.

2.1.3 Lightweight, partial formalization

By dividing the verification problems into perspectives, where the problem are efficiently captured and proved by the proper models and tools, the complexity of the problem description are greatly reduced. The models under perspective based verification are therefore small and easy to follow by the designers and the verifiers. This is what is meant by lightweight models.

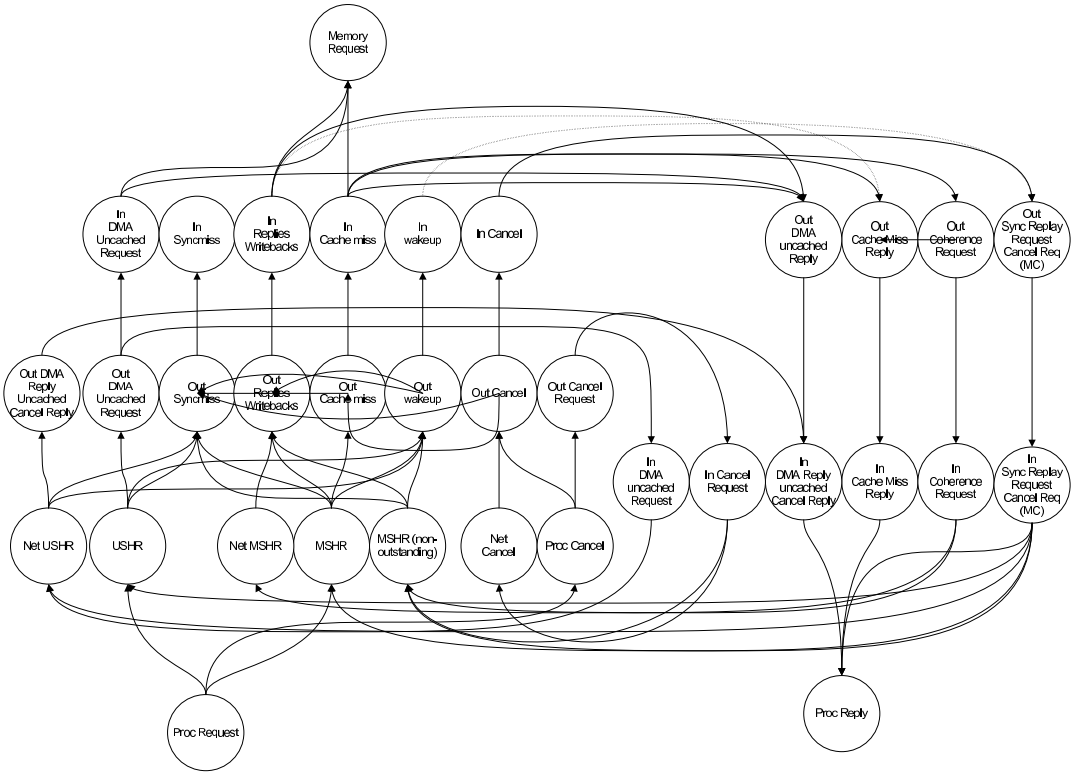


Figure 2.1: Memory System Dependence Graph

2.2 Example Perspectives

This section two example perspectives that were used to help validate the implementation of a shared memory multiprocessor [need to say what is implemented in these networks] and deb

2.2.1 Dependency Perspective

The dependency perspective is used to model the wait conditions that can occur in the system. It is often the case that in the design of a protocol or a network contains such dependencies. Protocol often require one command waiting for another command to finish before execution. Network also may have such dependencies due to traffic on the network and limited buffer space. The dependency perspective focuses on capturing these protocol and resource dependencies, and it verifies that there are no cycle in these dependencies, thus causing a deadlock situation.

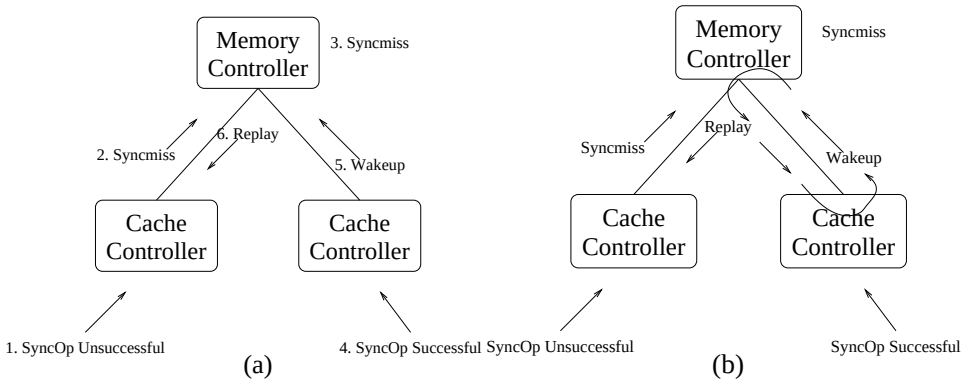


Figure 2.2: SyncOp deadlock bug

2.2.1.1 Example

Figure 2.1 is the dependency graph of the memory system in the Smart Memory Project. Because the dependence perspective only models dependencies, the graph can be made simple, easy to analyze, and easy to change as the design changes. Yet at the same time, this graph is able to capture both the network dependency, the protocol dependency, and the cache controller implementation dependencies. The combination of these dependencies are usually overlooked by the designers as a place for corner case bugs, and by using the dependency perspective, several bugs was caught using this model early in the design process.

2.2.1.2 Bugs found

Figure 2.2 shows an example corner case bug found in the synchronization operation using the dependency perspective. The synchronization operation is one of the mode of Smart Memory that make sure that provide synchronization between different threads on different CPUs through the use of shared memory space. When a synchronization operation occurs, it can either be successful, or unsuccessful. In the case that the synchronization operation is unsuccessful, it the memory system will put the thread on hold until the right condition occurs for the synchronization operation to try again. Figure 2.2a list some of the basic operations that the synchronization operation protocol uses.

1. SyncOp Unsuccessful occurs when the synchronization operation is attempted and unsuccessful.

2. In such case, a syncmiss command is sent to the memory controller to keep track which thread have a pending synchronization operation.

3. The Syncmiss command is stored in the memory controller for future replay

4. A SyncOp command can be successful

5. When a synchronization operation is successful, and if there are waiting operations pending, a wakeup will be sent to the memory controller to replay the waiting operation.

6. When the memory controller receives a wakeup, it will re-issue the syncOp command to replay at the cache controller.

The protocol itself is simple and seems to have no problems. However, when the dependencies of the protocol is combined with the dependency in the implementation of the cache controller, unexpected deadlock scenario can occur. Figure 2.2b shows such a deadlock discovered when the dependency graph of figure 2.1 is analyzed for cycles. To see the circular dependency, first we note that all of the messages described in figure 2.2a can be sent and received by any of the cache controller. In particular 6. Replay message can be sent to any cache controller. The internals of the cache controller is such that if the virtual channel for wakeup is congested, then it have no buffer space to receive replay. And on the memory controller, the original design is such that if the replay message channel is congested, then it have no buffer space to receive a wakeup, causing a loop in the dependency graph, thus causing a potential deadlock situation.

In this example, it is demonstrated that the use of perspective allow different layers of design to be modeled when there is a potential bug caused by the interaction of the two layers of design. In the synchronization operation case, it is the interaction between the synchronization operation protocol and the architecture design of the cache controller and the memory controller that cause the deadlock. But more

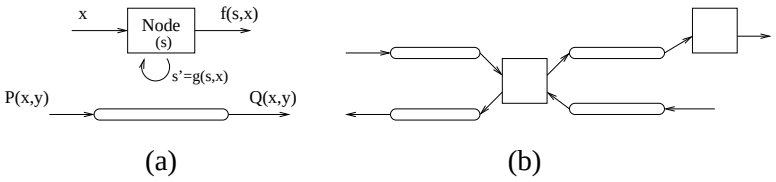


Figure 2.3: Messaging System Model

importantly, this is easily modeled because there are many detailed information such as the fact that there could be multiple cache controllers and multiple memory controllers that will complicate the model, but do not add much in terms of finding the correctness of the design with respect to the deadlock free property of the system.

2.2.2 Message System Protocol Perspective

The correctness of the synchronization protocol itself was also verified through the perspective based verification. The correctness of the protocol depended on the action of the memory controller and the cache controller when it receives a message in conjunction with the ordering requirements of the messages themselves. The Message System Protocol Perspective is recognized as a useful perspective for verifying these types of designs.

2.2.2.1 Model

The Protocol Messaging System model consists of nodes and message conduits, which is illustrated in figure 2.3a. Nodes are simple mealy state machines with the messages being the input to the machine. Conduits passes the message as it is, but places restrictions on the ordering of the messages. Figure 2.3a shows that if the ordering between message class x and message class y at the input can be captured by predicate $P(x, y)$, then the output ordering for that conduit must satisfy $Q(x, y)$ between message class x and message class y. Message class is a set of all messages that satisfy a particular property definable by a predicate.

These elements of nodes and conduits are combined together into a network as shown in figure 2.3b, and safety properties of the states in the nodes or end-to-end

orderings can be proved using this model. The Message System Protocol Perspective simplifies the verification of messaging ordering properties or protocols which depends on the message ordering specifications of the network.

2.2.2.2 Bugs found

The Synchronization protocol maps easily into the Message System Protocol Perspective, and the correctness requirement of making sure every synchronization operation is serviced was verified. Using this perspective, a previously unknown requirement on the synchronization operation protocol was found. Figure 2.2a shows that either a wakeup or a syncmiss message can be sent from the cache controller depending on the success of the synchronization operation. By modeling the synchronization operation protocol in the message system protocol perspective, the requirement that wakeup message cannot get ahead of the syncmiss message (but the syncmiss message can and must be allowed to get ahead of wakeup message to avoid deadlock) is found early in the design process. The ability to construct a model early allowed the designers to change the processor-cache controller communication protocol to fix this bug before much details about the processor-cache controllers have is finalized.

2.2.3 State transitions with safety properties

Model checkers have been very successful in verifying safety properties and some temporal properties of low level designs. Yet applying to larger designs with different protocols have pretty much been hit-or-miss. Model checkers have been applied to some projects with ease, while it was difficult for verification of some other projects. The idea of perspectives gives a framework for explaining why some verification projects have worked well and why some have not.

Section 2.1 described that the characteristic of a good perspective is that it captures the model used by the designers, it verify properties that is of the same property class, and it capture the verification problem efficiently. Thus, the reason why model checkers do well on low level control designs is because low level design descriptions are most efficiently represented by a state machine representation. The property class

being verified by model checkers are safety and some temporal properties. And as long as the design remain relatively small, model checkers is a good tool to verify low level control designs.

It is also well known that data-path designs are hard to verify using model checker verification techniques. This is because data-path designs are not represented efficiently by state machines, thus model checker is the wrong perspective to use to verify data-path designs. By understanding the concept of perspective based verification, it will help the verifiers to know when to apply traditional verification techniques, and when to apply techniques that are developed for other types of perspective verification.

2.2.4 Parallel Transaction Perspective

One of the verification issue that still plague designers is the complex interaction that occur when multiple transactions are executing in parallel, and thus might interfere with each other. Although each of these transactions, when taken into consideration by itself is easy to design correctly, it is unclear that correct behavior is maintained when multiple occurrences of the process are executed at the same time due to the wide range of execution order and resource access patterns that are possible. TDV is designed to verify designs with this type of concerns, that is, designs with simple single process execution, but complex multiple process execution on the architecture design level.

Generally speaking TDV, is designed to verify the design as described by a Transaction Diagram. The Transaction Diagram describes the behavior of a single process execution. If TDV is to be used as part of the verification along with design methodology, the TDV models needs to be easy and quick to build. This means that the Transaction Diagram for one transaction must be simple so that the model can be constructed quickly. Thus TDV is practical to designs which have simple single process descriptions, but TDV is able to verify designs that are difficult to implement correctly due to the interacting behavior of multiple transactions.

There is no set definition of what constitutes a “simple description” that will work

with this methodology. A more complex description of the process means that more time must be spent in building the model, and this may or may not be acceptable depending on the circumstances of the project. Chapter 5 describes some real world examples where the transaction description is simple enough to be useful.

2.3 Uses of Perspectives

2.3.1 Why do perspectives work?

The goal of perspectives is to find corner case bugs while maintaining a simple model so that it may be understood and modified by the designers and the verification engineers as the design changes. Perspectives are built so that it is lightweight, achieving this objective. The two reasons why the idea of perspectives is able to accomplish this is because it applies constructive modeling to the formation of the formal model, and it follows the designer's intuition on the workings of design. The rest of this section will describe characteristics of perspective based verification that allow models constructed in this framework to be lightweight.

2.3.1.1 Constructive modeling of small models

Instead of taking a design and abstracting away information until it reaches a manageable level, perspective based verification build a model until it contains enough information to prove the property of interest. This approach motivate and allows the verification engineer to keep the model simple not only for the purpose of having the ability to run formal tools on it, but also for the purpose of achieving maintainability as the design changes. This same approach is also recognized by Suhaib, Mathaikuty, Berner, and Shukla [36] to achieve an agile model which can change with the design.

However, our experience have shown that without the correct specification framework for the problem under verification, achieving small model formal verification is not possible. This is because without the correct modeling framework, much of the model specification and model construction time is spent on properly converting one

model framework that is natural to the problem at hand into another model framework that is available by the formal tools, thus making the formal model large and hard to change. By recognizing a set of commonly encountered verification problems in today's design, the correct tool which targets toward a specific level of abstraction, uses a specific model, and captures the reasoning of the correctness of the design can be chosen, making small model construction possible. This means that no single one formal tool should be used in a large design across multiple abstraction levels because it contains verification problems that belongs to different perspectives. Instead, the verification problem should be broken up into multiple perspectives and different tolls should be used order to verify the various properties of interest in a design.

2.3.1.2 Designer's intuition

In constructing the formal model for finding corner case bugs, perspectives allow the capture of the designer's concept of the design and the designer's correctness argument efficiently with a formal model. This is because many unnecessary details that are not relevant to the correctness of the property in a perspective can be thrown out and not included in the verification model. Since a designer's model typically represent an efficient encoding of the design, thus a formal model which parallel the designer's model are also small and lightweight.

By allowing the perspective to capture designer's intuition, it further insulate the model from changes to the design during the design process because any changes to the design will not usually change the intuition of the correctness of the design. However, if major changes are required, changes that follow the designer's intuition will also require major change to the design. Thus the cost of changing the perspective parallels with the cost of a design change. As the result, the designers will less likely to make changes that will require major change to the verification models since it will be equally costly to them to make those changes in the design.

2.3.2 How to choose the right perspective?

Different perspecties are required to capture the different ways the designers think about their high level model and they way they think about why their design is correct. The idea of perspective based verification is to identify a set of perspectives commonly encountered in design, and develop the right tools and modeling techniques so that verification problems encountered can use the right perspective model and tool in an efficient manner. Thus, a important aspect of making perspective based verification work is finding the correct perspective, and therefore the correct model and verification tool for a given property class that the designers wish to verify.

A useful way of choosing the correct perspective is to ask the question "Why is the design correct?" The following examples shows how each perspective is suited to how the correctness question is answered.

1. Dependence perspective: A design is deadlock free if there are no dependency cycles can occur in the system. For designs with static dependency, this property can be verifyed by drawing out the dependency graph and check for cycles. This correctness verification is what the dependence perspective captures in its model.

2. Message System Protocol Perspective: The correctness of a protocol such as the syncronization operation protocol depends on state machine changes and ordering of the messages. The message system protocol perspective provide a mechanism for modeling these two properties as well as provide a framework for integrating them.

3. State Transition with Safety Properties: In this perspective, the design is argued to be correct because the complex description of the state behavior should not bring it to an error state. The specification used by many model checkers provide a mechanism for describing state transitions as well as valid or invalid states.

4. Parallel Transaction Perspective: To verify that parallel transactions do not interfere with each other, the intuitive correctness argument is usually based on

a combination of how one transaction's actions will not affect another or another mechanism ensures that interfering transactions does not take place. The parallel transaction perspectives focus in on allowing the user to encode those arguments simply in the model, thus the parallel transaction perspective have been a good model to use for verification of the cache controller architecture.

2.3.3 Abstraction Layers

Perspectives target finding corner cases for a particular property class and at a particular abstraction level. The model built to represent the design will be an abstraction of the actual design, but with enough details to capture the corner case bugs that are introduced at that level of abstraction. As the design progresses, more details about the implementation of the abstract model will be made known to the designers. The abstract model constructed for verification purposes can now help with the verification of the detailed design by having a formal specification for the requirements of the implementation. That is, the model and the assumption used by the abstract model is now the specification requirements for the next level of implementation.

Requiring formal proofs for all the proof obligations generated by the abstract levels is practically unfeasible on a large design project because each abstract level usually increase the number of proof obligations exponentially. But instead, one should focus on the proof obligations that may produce unexpected behaviors in the next level of abstraction. This methodology works well in practice because most of the proof obligations created at an abstract level can be easily verified using traditional simulation techniques, but it allows for verifying the unexpected interactions between abstract levels which typically result in many corner case bugs in design projects.

2.3.4 Perspective trade-offs

The idea of using perspective works by reducing the model size by targeting the model for a particular property class and abstraction level. However, by dividing up the problem into perspectives, it requires two trade-offs compared with complete formal verification methodologies.

The first trade-off is that since the models are abstracted version of the actual design, and no formalized process exists to verify that the design implements the abstract model, one cannot say that the entire design is completely verified. However, the perspective verification idea makes protocol and architecture level verification of corner case bugs possible early in the design stages.

The second trade-off of using different tools and models is that the different descriptions of the design are not easily linked with each other because they are using different theories that may not be compatible with each other. Under the assume-guarantee reasoning, assumption of the input of the model is the guarantee of it's neighboring modules, or the assumption of the abstraction level is the guarantee of the lower level implementation. With perspective based verification, they are composed of different models, and the assumption and guarantees of one perspectives is not directly translatable into the model of the neighbor or different abstraction level. Thus a formal link that the assumption of one model is ported to be the guarantee of another model is hard to do with perspective based verification.

2.3.5 Model Checkers and Perspectives

At a fundamental level, model checkers are designed to verify that an execution does not reach an error state in a complex state machine execution. Since most verilog level designs are modeled as state machines, and all designs are computationally equivalent to state machines, it have been widely used as the base of formal verification models.

As the result, model checkers have been used as the tool to verify designs with different perspectives. In addition, model checkers have been successful at it because model checkers have developed techniques to reduce the problem size to deal with verification problems with different perspectives. However, two problems remain with using just model checkers to solve problem with different perspectives. One is that model checker do not have the capability to solve all perspectives that might be needed. The difficulty of using model checker to verify data path design is an example of a perspective that model checker is not well suited for. The second is that the input representation to a model checker is not concise and efficient way to represent the

abstract design and the designer's intuition. This will cause the model to be too large and not easy to change early in the design phase, where lots of changes are occurring.

However, many of the tools developed for each perspective may be built on top of a model checker, leveraging the research that have gone into them which makes them efficient and fast state machine verifiers. The advantage of perspective is that each tool can, first of all, incorporate proof ideas that are developed during the tool development time. This way, the model checker does not have to deal with the complexity about the perspective that it is not suited to solve. Secondly, the input specification for each perspective tool are geared toward efficient representation of the designer's abstract design and intuition. The tool may translate them into equivalent model checking problems to solve them, but the perspective tool allows the model to be kept simple and easy to verify along with the design.

Chapter 3

Transaction Diagram Verifier

The Transaction Diagram Verifier (TDV) is the tool that was used to correctness of design with Parallel Transaction Perspective. This chapter will describe the model of the Parallel Transaction Perspective and the use of TDV in verifying the correctness of this model.

3.1 TDV modeling

3.1.1 Transaction Model

TDV models a transaction based system. A transaction, as shown in figure 3.1, represent the behavior of the system when a command or request is received. TDV analyzes a system where arbitrary many transaction are occurring at the same time. A transaction consists of *transitions as* represented by the blocks in the figure, and *links* as represented by the arrows in the figure. The links represent the possible ordering of execution that can occur for a transaction.

Each block (representing a transition) is composed of *guards* and *actions*, which describe the atomic execution step of the system when the transition represented by that block is executed. The guards specifies that the design which the transaction diagram is representing will not execute the transition associated with that block unless the guard is true. The action describe the state change behavior when the

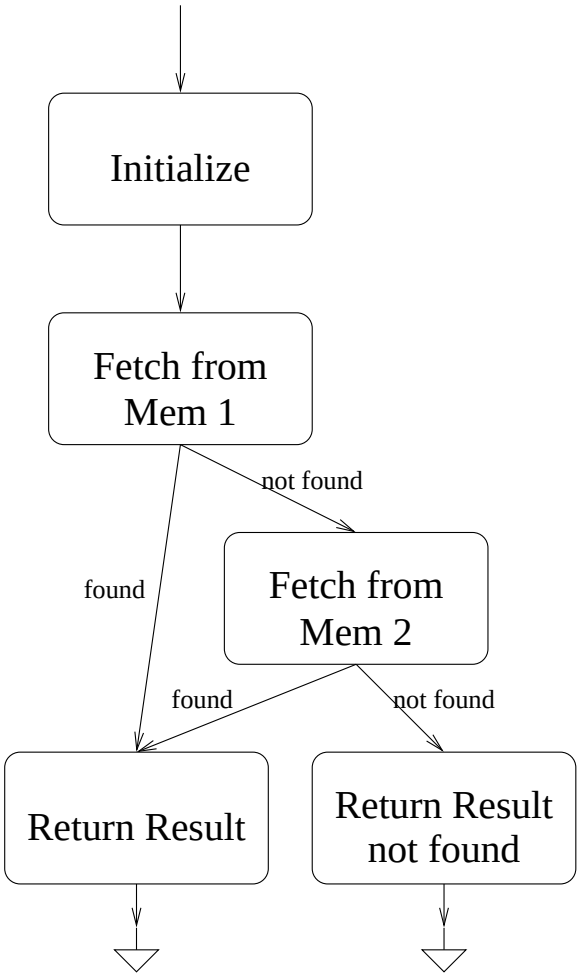


Figure 3.1: Example Transaction

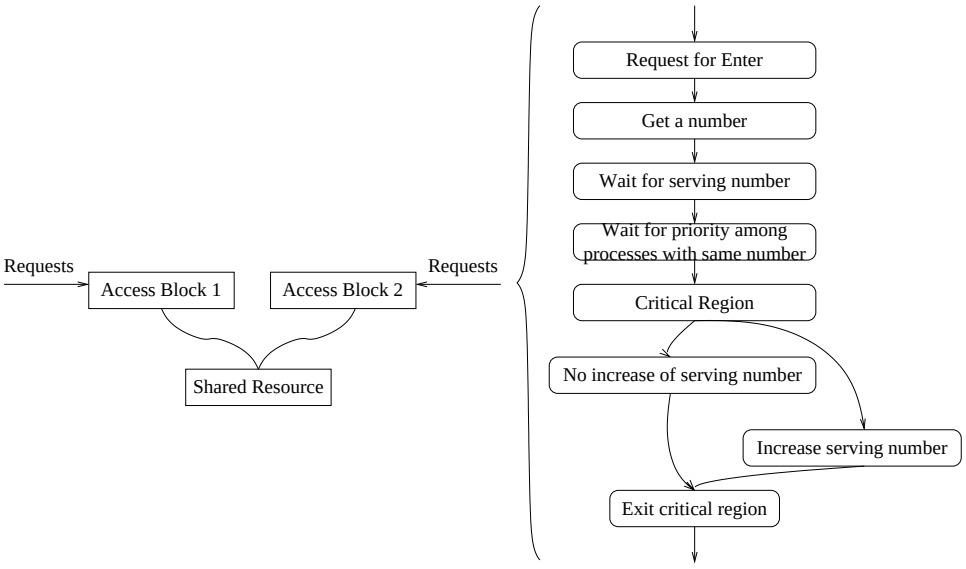


Figure 3.2: Example Design

transition associated with that block is executed.

3.1.1.1 Example

This section will give a example description of a TDV model for a simple mutual exclusion algorithm as an illustration of using TDV to verify a design. Figure 3.2 illustrate the example where two blocks of a system is trying to access the shared resource. The desired property to verify is that the two blocks in the system does not access the shared resource at the same time.

The transition diagram on the right side of figure 3.2 implements the Bakery Algorithm for mutual exclusion for multiple processes. The mutual exclusion algorithm works by sequencing the processes trying to enter the critical region. Each process requesting access to the shared resource will obtain a *sequence number* indicating the order which the competing processes will access the critical region. It is possible, due to parallel transactions occurring at the same time, to have multiple processes obtaining the same sequence number, in such a case, the tie is broken by looking at the process id, which is unique among the different processes. A common variable, the *serving number*, will control which processes is allowed to enter the critical

region. That is, if the processes's sequence number is equal (or smaller) than the serving number, then it may enter the critical region (if it have the lowest process id among all other processes that are allowed to enter). Once the process finishes with the critical region, it will increment the sequence number if there no more processes are allowed into the critical region with the current sequence number.

3.1.2 Variable Access

The guards and actions of a transition block are specified using boolean expressions on the state of the system. This section will describe how the state variables can be referenced by these boolean expressions that represent the behavior of the system.

The state of the system is divided into two categories–the *private state variables* and the *shared state variables*. In the system where multiple transactions are occurring at the same time, the private state variables are variables that are unique to each transaction, and the shared state variables are variables that are have shared access among all the executing transactions. Examples of private state variables include the flow control information, read and write results, or branching information. Shared variables contain state information that are not specific to any process. Example of shared variable includes the shared memory space.

In TDV, each transaction can reference its own private variables, the shared variables, and variables of another arbitrary process. References to its own private variables will be given the prefix *pri.*, references to another arbitrary process's private variables will be given the prefix *po.*, and reference to shared variable will be given the prefix *shr.*.

Consider the example started in figure 3.2 which implements the Bakery mutual exclusion algorithm. Each process will have a state information of whether it is attempting to enter the critical region (*pri.entering*), the obtained sequence number (*pri.seq_num*), and a helping variable to flag whether it is in the critical region (*pri.critical*)). Each process also have a unique process id (*pri.id*). Note that these private variables does not reference to the state of a specific process, but it represents the state of any generic process.

When a process tries to obtain a sequence number, it will get the value that is greater than all of other processes sequence number which can be expressed by $po.seq\_num$. The service number is the shared variable ($shr.serv\_num$).

To determine if a process will be next in entering the critical region, that process must have a sequence number lower or equal to the serving number

$$pri.entering \wedge pri.seq\_num \leq shr.serv\_num$$

, and it must have the lowest process id among all other processes whom also have a sequence number lower or equal to the serving number

$$po.entering \wedge po.seq\_num \leq shr.serv\_num \rightarrow pri.id < po.id$$

Note that the three appearances of the $po.$ prefixed variables in the above expression, refers to the private variables of the same other process even though the process is arbitrary. Also note that although the above expression only makes comparison with one other process (process po), because process po represents an arbitrary other process, the above expression represent that the current process will enter earlier than all other processes, that is, it is going the next in line to enter the critical region.

3.1.3 Guards and Actions

The guards and actions are represented by boolean expression predicates over variables. Each transition contains a predicate expression representing the condition for execution of that transition (guard), and a predicate expression representing the state change due to the execution of that transition (action).

For example, the transition "Wait for service number" is represented by the the guard for that transition by the predicate $pri.seq\_num \leq shr.serv\_num$, that is, the transition will not execute until the serving number is greater or equal to the sequence number obtained by the current transaction.

The action expression is expressed as a predicate on the current state variables and next state variables. The next state variables will be given the prefix pri', po',

and shr'.. The set of variables that the action may change as the result of executing the transition is the *action variable set*.

For example, the transition "Request for Enter" sets the variable $pri.entering$ to true, and thus it is expressed as $pri'.entering = true$ with the action variable set $\{pri.entering\}$. This present that the execution of this transition will change the variable $pri.entering$, and it will change the variable in such a way that $pri'.entering$ will equal to *true* (in another words, next state of $pri'.entering$ will be *true*).

3.1.3.1 Non-deterministic assignments

Using this format, non-deterministic assignments or partially deterministic assignment may be expressed. For example, the transition "Get a number", expressed as

$$pri'.seq\_num \geq po.seq\_num$$

with action variable set $\{pri.seq\_num\}$ states that $pri.seq_num$will be set to any value as long as it is greater than the sequence number of all other processes.

To express a non-deterministic assignment, the action predicate will simply be the boolean value *true*. For example, an action with the action variable set of $\{shr.critical\_loc\}$ with the action predicate of *true* represent a non-deterministic assignment to the variable $shr.critical\_loc$ when the transition is executed.

It is possible to write a description of an assignment such that it not a model of any system. That is, as action predicate which reduces to always false will result in no possible assignment possible by any model. Since any scenario which this happens means that there is a problem with the action description, TDV tool will verify that all action predicate is satisfiable during the verification stage.[1] This places a sanity check on the TDV model construction, although it does not guarantee that the TDV model accurately represent the design under verification.

[1]In reality, it verifies that the antecedent of an implication statement is satisfiable, which will include the check for the satisfiable check for action predicates.. This will be further explained in Chapter 4.

3.1.3.2 Atomic Execution

Transitions represented by blocks in the transaction diagram are modeled as an atomic execution. That is, when there are more than one variable in the action variable set, the assignments to the next state variable for all the variables in the set are done all at the same time, according to the behavior described in the action predicate.

Not only the actions are atomic, the guards and the action are also atomic. That is, the check that the guard is true for a transition to execute and the actual execution of changing of state of the system is done at the same time.

For example, the transition of "Increase serving number" will increase the serving number if all transactions attempting to enter the critical region have higher sequence number than the current serving number. This condition is represented by the guard

$$po.entering = true \wedge po.seq\_num > shr.serv\_num$$

If this condition is true, then the transition block will increase the serving number:

$$po.entering = true \rightarrow shr'.serv\_num > po.seq\_num$$

with the action variable set of $\{shr.serv\_num\}$. (This will actually increase the serving number to include all current waiting processes.) Because this transition is atomic, the check and the assignment of $shr.serv\_num$ occurs without interruption from any other process. If the user of TDV wishes to model these two actions as interruptible events, he may easily do so by splitting the "Increase serving number" transition block into two blocks in the TDV model.

3.1.4 Links

The order of execution for any transaction in the system is modeled using links. A link is a directed edge between transition blocks representing possible next execution blocks for a process after the current block executes. It is possible to have more than one link coming out of a block. In this case, the next execution may be any one of the blocks pointed to by the links as long as the guard is satisfied. The choice of which

block the transaction will actually take is non-deterministic, but it the choice can be modeled by specifying the guard of the the links are pointing to.

For example, after the critical region, the execution can either incrementing the serving number or not, depending on whether the serving number needs incrementing. In figure 3.2, there is a link from critical region to both of those transition blocks. As seen in the previous section, the guard for "Increase serving number" is

$$po.entering = true \wedge po.seq\_num > shr.serv\_num$$

, which if true, will increment the serving number. The guard for "No increase of serving number" then, is the opposite expression

$$\neg[po.entering = true \wedge po.seq\_num > shr.serv\_num]$$

. Thus the execution either take one path or the other depending on the state of the system.

3.1.5 Invariant Modeling

The properties that TDV verifies are given in the form of pre- and post-conditions of a transition block. Pre-conditions of a block are boolean expression assertions that are expected to hold when any process is poised to execute that transition block. This is in contrast to guard of the block, where the expression could be false when a process is poised to execute, but will not execute unless expression is true. Likewise, post-conditions of a block are boolean expressions assertions that are expected to hold when the system have just executed that transition block.

Because TDV does not analyze the model based on previous execution history, any information about the state of the system that is required to execute transition correctly are specified using pre- and post-condition boolean expressions.

For example, the execution "Wait for serving number" transition ensures that the process's serving number is less or equal to the current system serving number before it is allowed to enter the critical region. In another words, the post-condition

of "Wait for serving number" is $pri.seq\_num \leq shr.serv\_num$. This condition must also be true when the process enters the critical region. Thus the pre-condition of "Critical Region" is also $pri.seq\_num \leq shr.serv\_num$. In order to model this condition is maintained from the execution of block "Wait for serving number" to "Critical Region", this assertion is also the pre-condition and post-condition of the block in between these two events "Wait for priority...".

Note that "Wait for priority..." block itself does not actively ensure that the condition is satisfied, but this specification states that this property will be maintained through the transition block, and thus TDV will perform the verification for this property under the multi-process executing environment.

3.1.6 Exclusive Property

Recall that the prefix *pri.* refers to the private state variable of its own process, *po.* refers to the private state variable of another arbitrary process, and *shr.* refers to the state variable shared by all processes. In addition to these prefixes, the exclusive prefix can also be used in TDV to reference to a process with an particular definable property.

Consider a system where a token is being used to indicate which process can have access to the resource. Also consider that we wish to express the property, the process who have the token have flag variable set to 0. This property is expressible under TDV using the exclusive property prefix.

The exclusive property prefix can be used in TDV to refer to the process with a particular property, but this process must be exclusive in the sense that there is at most one process in the system with such a property at any time. Considering the token example, in order to reference the process with the token, the system must ensure that no two processes can have the token at the same time. The user of TDV defines this exclusive property A, $pri.token = true$, which tells the TDV system that only one process should have property A at any time. The user then associates a prefix, for example $pA.$ with this exclusive property. Then whenever the TDV model uses the state variable with prefix $pA.$, then it is referring to the process with the

exclusive property.

TDV will automatically insert the invariant that property A is indeed exclusive as the pre-condition and post-condition for every transition block. The invariant is constructed by taking the exclusive property, replacing the references to *pri.* with *po.*, negate it, then ANDing it to the original exclusive property. In our example, $pri.token = true \wedge \neg(po.token = true)$ would be the invariant that will be added to the pre- and post-condition of every transition block.

Now we may express the property that the process with the token have flag variable set to 0 by

$$pA.flag = 0$$

Although in this particular case, the expression $po.token = true \rightarrow po.flag = 0$ can capture the same property, in practice, it is helpful to be able to model the exclusive process with a different reference prefix than *po.* so that the TDV model may still reference an arbitrary other process in its expressions in addition to the exclusive process.

3.1.7 TDV Verification

TDV verifies that the pre-conditions of all blocks are true for any process that is about to execute that block. It also verifies that the post-conditions are true whenever a process have completed the execution of the block. It does so by verifying that these conditions are maintained for any process execution that may occur in the system, given that these process executions follow the the links, actions, and guards of the blocks specified in the TDV model.

TDV verification is conceptually divided into two verification tasks. One task is to verify that pre-conditions and post-conditions hold under single process execution. The other task is to verify that pre-conditions and post-conditions hold when multiple process can execute at the same time. In single process execution verification, TDV verifies that when the pre-condition of the block is satisfied, if the transition is allowed to execute because the guard is true, and when the transition state behavior is described by the action of that block, then then the post-condition of that block will

become true. TDV also checks that if there is a link from block A to block B, denoting transition block B may execute after transition block A, then the pre-condition of block B can be derived from the post condition of block A.

To verify that pre-condition of a block holds in a multi processor environment, TDV assumes that the pre-condition of a block is true. Then TDV execute any other transition block for any other process and verifies that the pre-condition is maintained after this execution. The formalization of the correctness that these checks will prove that pre- and post-conditions is maintained for all possible executions is presented in chapter 6.

3.2 Properties of a TDV model

There are several properties to the way TDV models the design that make it better suited for multi process architecture verification compared to existing tools. These properties are selected during the design of TDV in order to balance expressiveness of the model and the speed of verification for the target application.

Since the target application is architecture level verification, an abstract description of the circuit is necessary to describe the design in TDV. This is accomplished by allowing course grain execution description for the transition blocks. Each of the blocks in the TDV model can represent an abstract execution, which allows for more than one state variable to change in an execution. So concepts such as "write data into cache," which is composed of updating the address field and the data field can be described as one operation under the TDV model. Allowing abstracted block description is necessary in order to keep the model simple by not including every implementation detail in the TDV model. In TDV, allowing multiple state variable assignment is accomplished by including multiple variables in the action variable set of the block.

Another property of the TDV model which allows for abstract description is non-deterministic modeling of the properties in the block, especially for assignment. As detailed in section 3.1.3.1, the assignments can be fully non-deterministic, partially non-deterministic, or deterministic.

Although the whole TDV model may have a large state space for each process, most of the time, a specific block in the model will only probe and assign a small number of states in the system. By capturing all the required information needed to prove the correctness of a transition in the pre- and post-conditions, TDV does not need to enumerate and keep track of the entire state of the process. The term *frame condition* will refer to the fact that the irrelevant state variables are not changed during an execution of a block. Since for most transitions, most states remain unchanged, the behavior of the system are efficiently represented through the use of the action variable set, where only variable that are changing by the transition are explicitly expressed.

It is interesting to note that not all assignments will have to be explicitly stated in the TDV model. For example, if a transition clears the lock, from a TDV correctness point of view, it may non-deterministically clear or not clear the lock. While this may cause deadlock in the real system, the safety properties may still be completely verified using this simplified model of the design.

Part of why this form of specification works is because TDV does not carry state information from the execution of one block to the next cycle, instead it only depends on pre- and post-conditions for assumption about the state information. Thus, the lack of post-condition concerning the state of the automatically implies that a transition may non-deterministically change the lock variable. This is another way for the user to avoid putting in non-relevant details about the model which speed up the TDV model building process.

TDV is able to verify multi-process model quickly because the input property are restricted. Yet this model is powerful enough to model most of the abstract architecture verification designs. By allowing the expression to access the state space of two arbitrary processes and the process with exclusive property, TDV restricts the input property so that efficient verification algorithm may be used, but it also allows enough expressiveness to write the properties of interest that usually require explicit quantifiers in other tools.

3.3 TDV Contributions

TDV is a practical application of verifying interference free execution of programs that first presented in a paper by Owicki and Gries [25]. This technique have fallen out of favor because the previous application of this technique was applied to designs that are too large and too complicated for this technique to be practical.

Several properties about perspective based verification that makes this technique well suited for TDV. One is the perspective based verification uses small models. Two is that the TDV perspective captures the designer's reasoning of why parallel transaction should not interfere with each other–namely that the executions should be interference free. This reasoning parallel with the technique used by the TDV tool, thus using TDV to verify problems in the parallel transaction perspective result in efficient verification in terms of tool runtime and modeling time.

The use of the TDV tool can be viewed as a theorem prover with automated verification for interference free properties. The user guide the verification process by specifying the invariants that will complete the proof. TDV then automatically analyzes the verification conditions with respect to the invariants given by the user. Thus, although TDV is not as automated as a model checker, it have several advantages over model checker tool. One is that the user is able to better control the verification process, and thus able to capture the designer's intuition of why the design is correct and use it to guide the proof. Two is that TDV is able to verify interference free properties of arbitrary number of processes.

One of the idea that make TDV possible is restricting access to private variables to only a "self" process and an "other" process. This reduces the reasoning required by the tool, thus allow verification to be performed fast, yet it gave enough expressiveness to apply to most of the verification problems that we have encountered.

3.4 Limitations

3.4.1 Variable Access Limitation

As described in section 3.1.2 the TDV model allows access to private variables to one process and an another arbitrary process in addition to the shared variables. This means that properties that are expressed in the TDV model are restricted to expressions that are expressible with these variables. Properties that depend on the state of three processes are not expressible. For example, expression such as "If a process have property A, and if another process have property B, then the transition is allowed." is not easily expressible in TDV.

3.4.2 Invariant Specification

The TDV model presents an obligation for the user to provide invariants of the proof. Thus practically, TDV is geared only toward applications where invariants can be specified easily.

From the experiences of using the TDV tool, we have discovered properties of some verification problems have invariants that are hard to express. One example is when there are complex communication between the different threads such that the status of the communication cannot be captured by simple invariants. For example, consider a design where a message is passed from one process to another, making incremental changes to the message for each process, and eventually pass back to the originating process. The originating process must have an invariant that keeps track the possible changes that the other process is making to the message and the different combinations of events that can happen. This invariant is not a simple expression, and thus greatly complicates the invariant specification process.

Another case where the invariants may be hard to write is when the invariant for one process is dependent on the state of another process. The reason this might lead to complicated invariants is for similar reasons as before. Invariants such as "If another process have property A, then invariant a holds, otherwise, invariant b holds." Although the example given is manageable by the TDV model, when too many such

case is needed, the invariants will become hard to manage. Verifying a pipeline architecture is an example of needing complex invariants. The invariants required to verify a pipeline will require many different cases, depending on the bypass states and conditions of the pipeline. This means that for each bypass case, the invariant will have to account for that case to grab the data from the correct bypass location.

However, as with all engineering problems, this isn't a firm rule about the usability of TDV on designs. There are cases when the invariant is dependent on the state of another process, but it does not produce complex invariants either because the other process does not change their related states much, or the other process is prevent from doing so using some other mechanism. There is a range of trade-offs between the complexity of the invariants (and thus verification time) verses the range of problems that TDV can be used practically. In addition, there might be the possibility of finding another way to model the design such that invariants will not be hard to write. Whether this can actually be done and still maintain the accuracy to the real design will depend on the design in question.

A helpful design technique that may help in producing simple models is design for verification. That is, instead of matching a model to the design, we allow the models to be constructed in the simple way, and check that the design can match the model. Chapter 5 described the verification effort done using this methodology on the Cache Controller architecture design. By using this technique or modeling the design in a different manner, it may be possible to avoid complex invariants that hamper the verification speed when using TDV.

3.4.3 Non-interactive Concerns

The advantage of TDV was to reduce the verification problem of interactive free property and reducing it to simple to verification rules that can be analyzed by the tool. For properties that do not fall into this category, TDV does not offer any advantage over existing formal verification tools. One example of this is the verification of the correct execution of a single process,

The Cache Coherence protocol of the Smart Memory processor is an example

where the primary concern for correctness does not come from interacting processes. At this level of description, the designers are concerned with the rules and the action of each process will maintain the description given by the MESI protocol. That is, the transformations used by each process will keep a global invariant of the MESI protocol. Although TDV may be used to verify this property, it does not offer any advantage compared with other tools which are also invariant based. This is because the verification task does not involve detecting conflicts with the interacting processes, and the mathematics behind the TDV tool have only improved upon the method for detecting conflict with multiple processes, thus any other invariant based tool will work equally well as TDV.

3.4.4 Quick Model Construction

To use TDV in a verification along with design setting, the model must be able to be constructed and verified quickly. Thus it is suitable for verifying designer's ideas with simple single process state machine, but it is not suitable for designs with complex state machines. A model checker is better suited for those verification problems. However, it is possible to simplify a single process state machine to a simpler model using other means of verification or abstraction, then use TDV to verify the interaction between these processes. Whether this is possible will be dependent on the design and the ability to abstract the model.

Chapter 4

TDV Internals

This chapter will describe how TDV represent the model as described in chapter 3 and how it verify the invariants of that model.

4.1 TDV Input Organization

Currently, an API C++ interface is used to construct the TDV model that will be described in the following section. The TDV model represents a transaction diagram that models the execution of a process. Each execution step is represented by a block, and the possible execution path is given by the links. Each block consists of the guard and the action property, which represent the design under verification. Each block also consists of pre- and post-condition properties, which represent the invariants to be verified by TDV.

4.1.1 Boolean Expression

Boolean expressions are the basic building block of the TDV model, for they are the predicates that are used to express pre-conditions, post-conditions, and guards of the blocks. The TDV tool offers an API interface which is used to construct Boolean expression in the TDV system. TDV offers the standard Boolean connectors of NOT, AND, OR, Implies, If-then-else over terms. A term consists of equality or inequality

comparisons between two variables.

A special class of Boolean expressions represent assignments in the TDV model, which are the actions of the transition block. In addition to the Boolean formula, the assignment also consists a list of variables which is the action variable set (see chapter 3).

TDV uses these Boolean expressions, which represent pre-conditions, post-conditions, guards, and actions, to construct verification obligations, which are then proved using STP. That is, TDV takes the Boolean expression and construct equivalent STP expressions for verification.

4.1.2 Variables

The terms in TDV are composed of equality or inequality comparisons of variables. Since TDV uses the STP as the backend of its logic, it supports terms that are supported by STP. Which include bit-vector and arrays of bit-vectors. Boolean variables are represented in TDV as bit-vector of length 1. Multi-dimensional array expression are also supported by TDV.

When the variable is being used in the context of a property under verification, it represent the possible values of that variable at particular point in time in an execution. Thus separate variable is used in the property expression that represent the value of the variable in the current clock cycle and the next clock cycle. In TDV, this distinction is represented by adding the tick symbol (') to the prefix of the variable representing the next clock cycle. For example, if $pri.A$ represent the current value of the private variable A, then $pri'.A$ represent the value of the private variable A at the next clock cycle. An expression of $pri.A = true \wedge pri'.A = false$ represent a transition with the behavior that that A goes from high to low during the transition.

TDV also support the use of constants, which allow the naming of a specific bit-vector pattern. For example, a very useful constant that is built in the TDV tool are the constant *true* and *false* which is a short hand for bit-vectors of length 1 and have the value 1 and 0. Other uses for constants in the TDV model are to represent the state of the system using descriptive names in the Boolean expression instead of the

number encodings.

4.1.3 Frame Condition

TDV contains subroutines which automatically compute the frame condition for a given state transition. Recall that under the TDV model, any variable not explicitly listed in the action variable set will remain unchanged. This frame condition is used when TDV computes the Boolean expression that represent the next cycle property.

Consider the Boolean expression $P(s_t)$ that represent the property of the system at time t, and it execute a block which the action is described by $A(s_t, s_{t+1})$ with the action variable set S. This represent that the execution will set the variable in set S such that $A(s_t, s_{t+1})$ is true when $P(s_t)$ is true. The basis for TDV verification comes in the form $P(s_t) \wedge A(s_t, s_{t+1}) \rightarrow Q(s_{t+1})$. Where $Q(s_{t+1})$ is the property the model should maintain after the execution.

The variables in P represent the state of the current cycle (variables in its Boolean expression doesn't have tick mark in its prefix), and variables in A represent the state in the current and next cycle in the expression (Boolean expression contains variables with and without the tick mark). The variables in Q needs to be renamed by TDV so that it represent the variable of the next state (change the variables in Q from no tick mark to tick marked prefix).

The frame condition states that all variables not explicitly expressed in the action variable set will not change value. In another words, $pri.B = pri'.B$ if $pri.B$ is not in the action variable set. Instead of putting this expression explicitly in the TDV, TDV simply avoids renaming $pri.B$ into $pri'.B$ in the Boolean expression Q. Or in another words, TDV will rename the variables in Q into its next state variable expression only if it appears in the action variable set.

For variables in the action variable set which are array variables, the renaming expression gets more complicated because the index, thus the variables that will change, might depend on the state of another variable. For example, suppose the array variable $pri.A[pri.c]$ appears in the Boolean expression $Q(s_{t+1})$ and action variable set for a particular transaction is $\{pri.A[pri.b]\}$. This represent that the $pri.b$'s array

entry to the array $pri.A$ will contain a new value, while the rest of the array entry will stay the same. Thus, to rename the expression $pri.A[pri.c]$ that appears in Q, it must be replaced with the expression

$$\begin{aligned} if \quad & (pri.b = pri.c) \\ then \quad & pri'.A[pri.c] \\ else \quad & pri.A[pri.c] \end{aligned}$$

which capture the frame condition appropriately.

As an interesting note, if the action variable set is $\{pri.A[pri.b], pri.b\}$ then the expression $pri.A[pri.c]$ in Boolean expression Q will be replaced by

$$\begin{aligned} if \quad & (pri'.b = pri.c) \\ then \quad & pri'.A[pri.c] \\ else \quad & pri.A[pri.c] \end{aligned}$$

because $pri.b$ in Q will also be renamed to $pri'.b$ for it is part of the action variable set.

4.2 TDV Model Constructions

The fundamentals of the TDV models are composed of transaction blocks, made up of pre-conditions, post-conditions, guards, and actions, expressed in boolean expressions. The constructs that will be described in the following section are ways to organize these information and provide routines for entering common patterns easily into the TDV model.

4.2.1 Sub-Blocks

Sometimes, there are situations where the behavior and the assertions of the transition blocks are the same except for a few differences. To help with the input of the model

in this case, the concept of sub-blocks is used in TDV. A sub-block automatically inherits the pre-conditions, post-conditions, guards and actions of the block it is derived from. In addition, the user may specify more properties of the sub-block. In this manner, the user may create multiple sub-blocks that is derived from the same main block, thus avoiding having to repeat the same behavior pattern for all of the blocks. But at the same time, the different sub-blocks derived from the same block may contain different properties that distinguish one sub-block from another. In TDV, the sub-blocks are created by naming a block "mainblock.subblockname", which denotes that the sub-block is derived from "mainblock."

Internally, TDV handles the sub-blocks as completely separate blocks, so all of the logic of correctness associated with it are considered as separate blocks. All of the properties, including the links are automatically copied into the sub-block.

As an example, suppose one of the transition block reads the status from the system, but does not act on the result of the status until later. All the transition blocks between these two events behave the same way except that it is pre-destined to act on the result of the status at a future point in time. Sub-block can be used in this case to distinguish the status results, yet these sub-blocks can inherit from the main block which contains all of the behavior independent from the status result. This way, the user does not have to manually create multiple almost identical blocks, instead by using sub-blocks, TDV will automate this process.

4.2.2 PropertySet

A propertySet is a set of Boolean expressions that can be manipulated and reference as a group. It is equivalent to a a Boolean expression composed of the conjunction of all the Boolean expressions that are in the set.

4.2.3 PropertyClassSets

A common characteristic of TDV models is that a property P becomes true at one point in the execution, and that property must remain valid until another point in the execution of a process. The block execution which the property P should become true

is named the *begin block*. The block execution which property must hold property P is named the *maintain block*. The block which property P no longer needs to hold after the execution of the block is named the *end block*.

PropertyClassSets is the API construct that allow the automatic repeat of insertion of property P among the pre- and post-conditions of the transition blocks to achieve this behavior. And this is done by making P the post-condition of the begin blocks, making P the pre-condition of the end blocks, and making P the pre and post-condition of the maintain blocks.

The user specify the set of properties using PropertySet. And the user specified the region which this property holds begin blocks, end blocks, and maintain blocks through the PropertyClassSet API. Then TDV tool will automatically generate the appropriate pre and post condition and verify that they are maintained with respect to the design.

At the begin block of the TDV model, the user usually have to specify actions or guards (underlying design model) such that property P become true. This can be done in TDV by specifying an action variable set in addition to the property P. Then TDV will insert the assignment into the begin block in addition to adding it to the post-condition of the begin block. A common example of its use is to make sure that any value written at the begin block will not be affected by other processes until the end block. Then the boolean property of the assignment will become the pre- and post-conditions of the maintain blocks.

4.2.4 Construction of maintain blocks of PropertyClassSet

The "maintain blocks" in the PropertyClassSet specifies which block should inherit a particular property from the previous execution block, and maintain this property when it executes. Commonly, the designer's view is that a particular property is true from this point until another point. Thus, the maintain blocks are simply all the blocks which can be part of the execution path from the begin blocks until the end blocks. So instead of having the user specify the maintain blocks manually, a function is provided by TDV that automatically adds the blocks which are part of the path

from begin block until end block to the maintain block set.

Using this function reduces the number of changes required when the TDV model changes. Because the maintain blocks are automatically constructed, any changes to the TDV model with respect to links and blocks will automatically updated.

4.2.5 Inverse PropertyClassSet

A PropertyClassSet, say A, describes which blocks will cause a property P to become true, which blocks will maintain the property P, and which blocks will drop this property. It is common to also specify the inverse of this property, $\neg P$, for all of the other blocks. That is, when the property P becomes true at begin blocks, $\neg P$ becomes false. This is equivalent to having an inverse PropertyClassSet, say $\overline{A}$, where the begin blocks of PropertyClassSet A is the end blocks of the inverse PropertyClassSet $\overline{A}$.

In the same manner, the end block of PropertyClassSet A is the begin blocks of PropertyClassSet $\overline{A}$. The maintain blocks of PropertyClassSet A does not provide any requirements on the equivalent blocks in PropertyClassSet $\overline{A}$, but the blocks which was not mentioned in A becomes the maintain block of $\overline{A}$. Because of the availability of sub-blocks, the construction of the inverse PropertyClassSet is not as straight forward. But a function is provided in TDV to automatically construct the inverse PropertyClassSet so that property $\neg P$ can be added to the TDV model for all the blocks which does not hold property P.

4.2.6 Unchanged Value Property

One of the patterns that exists in proving interference free property of parallel processes is that a value, once read or written by a process, will remain the same until a specified operation is completed. To verify this property, a copy of the variable is made when this variable is read or written by the process. Since this is a fresh variable, no other process should have access to this copied variable. To check that the variable does not change, TDV simply verifies that the content of the variable is the same as the copy of the variable.

The TDV tool have a function that automatically put in an action that creates a

copy of the variable at the begin block (block which the requirement that variable is unchanged starts). And it adds the verification condition that the variable have the same content value as the copy for all the blocks which this variable needs to stay unchanged.

4.2.7 Global Invariant

This provides a way to insert invariants into all of the blocks in the model. It's an invariant that must be held true at all point of the execution by a process.

Since TDV does not have construct for functions, global invariant can be used as a way to mimic the function behavior. By defining an array variable, the array index acts as the parameter to the function, and the array value acts as the function return value. Since the global invariant means that the pre- and post- condition contains this property, and since there are no action that changes this variable array, the content of the array stays the same at all time globally. The array index can then be used as a easier way to reference expressions in other Boolean expression properties.

4.2.8 Block Regions

Block regions are a group of transition blocks that are mostly different, but they have a few properties in common. These blocks can be grouped together through the use of a Boolean flag, where the flag is on for all the blocks within the same region group.

Using the PropertyClassSet, the user can specify a group of blocks which is of the same group. TDV can take this group, and create a model which will modify the flags at the appropriate times such that whenever the execution of a process reaches those blocks, the flag will be on, and when the execution of a process is not at those blocks, the flag will be off.

There are two ways to specify properties which belongs to the group of blocks as identified by a flag. To specify that all the block in the group have property P, the user can use the global invariant to add to every block the property $(pri.flag = true) \rightarrow P$.

The other way to use the group block is that TDV can add the property $pri.flag \wedge po.flag \rightarrow P$ for all the blocks in the region. This construct states that no two

transitions which have the property P can be in the region at the same time, and it is used for specifying mutual exclusion properties.

4.3 Verification

The previous section describe the input the user can use to construct the TDV model and the verification task into the TDV system. This section will describe how TDV verifies that the pre and post conditions from model, as well as some of the control that the user have over the verification process.

4.3.1 TDV Verification Basics

Conceptually, TDV validates the following three things

1. if the pre-condition is satisfied when a process executes a block, then after the execution, the post-condition is satisfied.

2. If the execution of a process passes from block A to block B (there exists a link between block A and block B), then the pre-condition of block B is implied by the post-condition of block A.

3. For a process a about to execute block A for which the pre-condition holds, if another process b interrupts and executes another block B, the pre-condition of block A will still hold even after the execution of process b.

Verification of these three conditions is sufficient to show that the invariants will hold in the model environment where parallel processes are running. Similar reasoning is used by Owicki and Gries [25] for their work on verifying concurrent programs by proving non-interference between processes.

Section **??** will give a more formal description of the three verification obligations as well as the proof that these three verification conditions will verify the pre- and post-condition invariants.

4.3.2 Verification layers

During the process of creating a proof strategy, one might encounter a situation where a non-trivial sub-proof is required. Instead of requiring the user to complete the whole proof, TDV allows the user to assume properties in order to prove the rest of the invariants. This allow the user to concentrate on one aspect of the proof at a time, and return later to construct the proof for the assumed properties.

This technique is also useful for properties about the design that are easy to prove using traditional simulation techniques. These proprieties can be assumed by TDV which can be verified using an external proof tool.

4.3.3 Vacuity checker

TDV perform some sanity check to make sure that the proof it is doing is not true trivially because the model description is a model of an impossible design. TDV verifies statements that are representative of the verification statements presented in section 4.3.1. As seen in that section, each of the verification statement verified by TDV is in the form of an implication statement, where the antecedent is a description of the current state of the system. The trivial case is when the antecedent is provability false, which implies that the antecedent contains a contradiction. This will allow TDV To prove its proof obligation, but it represent a model of the design which is unimplementable. Thus, it is important to catch these cases where the proof is completed trivially to make sure that the design is correctly modeled in TDV. TDV does so by verifying that for all the statements that TDV uses to prove the correctness of the design, the antecedent of the statement is satisfiable. If TDV detects that an antecedent is not satisfiable, TDV will mark it as a bug in the description of the design.

The other sanity check verifies that the description of the model cannot allow TDV to conclude that there is more than one transaction with the exclusive property. Because the TDV must prove that the exclusive property is exclusive to one transaction, if TDV is also able to prove that more than one transaction can have the exclusive property, it suggest that there is an error in the description of the model.

Chapter 5

TDV Case Studies

In this chapter, several verification project using TDV, both successful and unsuccessful, will be presented.

5.1 Cache Controller Architecture Verification

The cache controller of the smart memory project manages the memory mats in the system. This section describes the verification of cache controller design when it is operating in the cache coherence mode to ensure that the cache controller service of the requests will maintain the requirement of the MESI protocol.

5.1.1 Architecture Description

The cache controller service requests from the processor or the memory controller by routing the request through each of the internal blocks. Each internal block have access to a specific part of the system. For example, the S-Unit deals with all the operations that require access to the tag portion of the memory mat. The cache controllers is programmed such that each request will initiate a sequence of commands executing in the internal blocks to complete the request. The internal blocks of the cache controller is shown in figure 5.1.

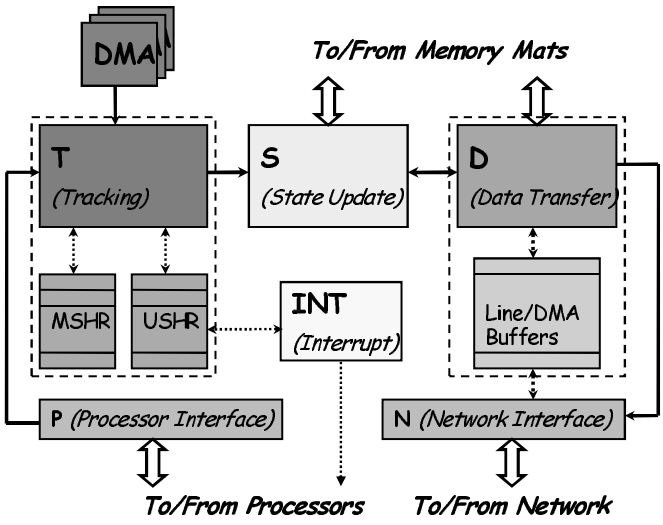


Figure 5.1: Cache Controller Architecture

5.1.1.1 Cache Line Access

In order to keep cache line request from interfering with each other, yet maintain the ability to service non-interfering requests in parallel, the cache controller design enforces a number of access restriction rules to the cache line. The rest of this section will discuss these restriction rules that allows for the non-interfering execution of multiple cache requests.

The first restriction limits the execution of processes with the same cache line request. Most of the time, the cache controller will not accept a request if it is currently processing another request with the same cache line. The one exception to this is when a process in the cache controller is waiting for a reply from the memory controller. This process is said to be in the *waiting* state. An important property about the design is that only requests originating from the processor can be in the waiting state. If a process in in the waiting state, then the cache controller is allowed to accept another request from the memory controller even if it have the same cache line address. This behavior is to prevent deadlock in the memory system. The verification problem is to verify that this exception will not cause consequences not intended by the designers. Indeed, the application of TDV found a bug due to

the complexity introduced by this exception. This bug will be described in section 5.1.6.1.

The second restriction limits processes with the same index from executing certain commands at the same time. Because any process can potentially access the line of another cache line of the same address through eviction, the *index lock* governs when the cache controller cannot accept another request with the same index because of potential conflicts. Recall that each request is broken down into a sequence of commands executed by the cache controller; *index locked* commands refers to the set of commands that should have the index locked during execution. By ensuring that the cache controller do not allow two process with the same index to be index locked at the same time, it should prevent processes from interfering with each other due to eviction. The specification for verification includes how the cache controller achieve this and which commands are classified as index locked commands, and section 5.1.6.1 is a corner case bug found by TDV caused by the incorrect classification of index locked commands.

The third restriction prevents requests with the same index and from the same processor from being processed by the cache controller at the same time. This is because the access patterns among requests from the same processor are different than access patterns among requests from different sources. This places additional restriction than what index locked can provide. Having the cache controller not accepting any requests if it is already processing another request from the same processor with the same index make sure that these processes do not interfere with each other.

5.1.1.2 Acceptance Conditions

The following summarizes the accepting conditions based on the cache line access restriction given in the previous section. The cache controller will accept a new request if the following are NOT true:

1. If there is currently a process with the same address that is not in the waiting state.

2. If there is currently a process with the same index that is still in the index

locked region

3. If there is currently a process with the same index that have the same requester.

The verification task is divided into two sections. One is to make sure that the cache controller follows these accepting conditions. Two is to make sure that these accepting conditions will result in a non-interference execution for arbitrarily number of processes processed by the cache controller.

5.1.2 TDV model

A TDV model was constructed to match the architecture design of the cache controller. Figure 5.2 shows the TDV model for the cache controller when operating under cache coherence mode. Each block in the figure represent the execution of a command of the sub-unit drawn in figure 5.1. The sub-unit of the cache controller receives a command from the previous unit, process the request, then sends the command to its neighbor unit. The links between the blocks in the TDV model (figure 5.2) corresponds to possible command sequence of a requests to the cache controller.

The blocks in the index locked region as labeled in figure 5.2 represent the index locked commands. The commands in the outstanding region represent that the process is in the waiting state.

Each block in the TDV model represents a transition action of the cache controller. Each of these transition action is represented by a corresponding state change in the TDV state. The acceptance condition are implemented as guard at the request acceptance block. The possible paths that a transaction can take are dependent on the result of the transitions that occurs in each transaction. Some of the path choices are modeled by the guards of the transition blocks, but most path decisions are modeled as a non-deterministic choice since the correctness of the properties of interest does not depend on the path of the transaction.

In addition to the actions that are related to the operation of the cache line, there are also actions that relates to the status of the process. The status variables is what the cache controller uses to determine whether the request should be accepted

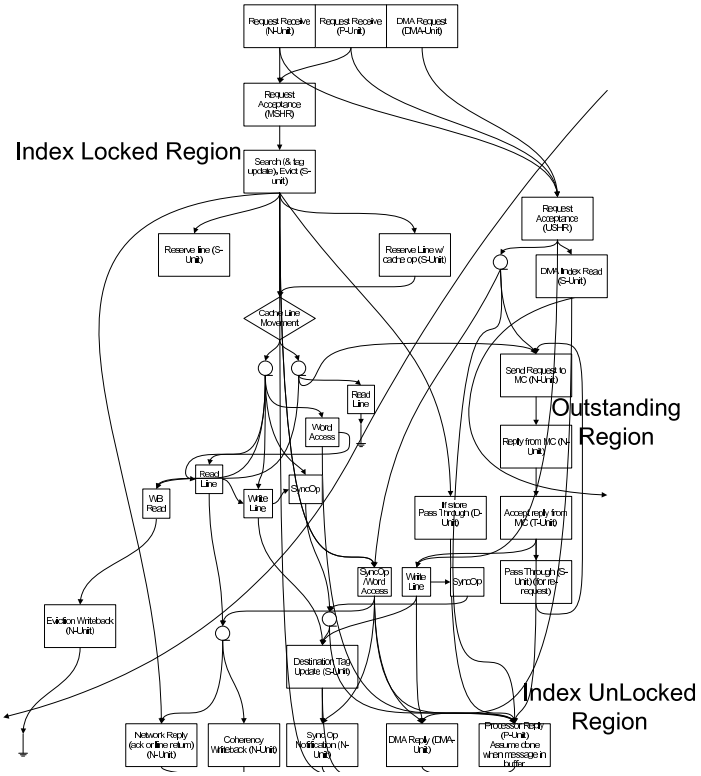


Figure 5.2: Cache Controller Transaction Diagram

to avoid conflict or not, and it is what TDV uses to verify that processes do not interfere with each other in unexpected ways. The status state include information such as if the process is finished, if it is currently index locked, or if it is currently waiting for memory request response.

The correspondence and verification that the TDV model accurately represent the cache controller architecture is done through manual inspection. Although this process is not formal, various design flows was found through process because the designs must still adhere to a precise definition of the TDV model.

5.1.3 Verification Task

Most of the time, the correct execution behavior of each process will depend on the result of the previous commands. In a single process environment, there is usually no problem with this behavior. However, in a multi-process environment, another process might change the state of the system, thus any decision by a process that was based on previous information may no longer be valid. If this occurs, the behavior of the process will no longer be correct. The verification task is to make sure that the information each block uses to make its decision is still valid when the decision is being made.

The decision that a process will make based on previous block executions are called *assumption information*. An example of this assumption information is the status of the cache line, which is obtained by the "Search & Tag Update, Evict" block. As an example of a property to be verified based on assumption information, if the cache line is found in a particular tile in a shared state, the process expects that the line will stay valid until it reads the data from that tile. The assumption information is that the line is in a valid state when it executes the transition blocks up until when the line is read. If another process interrupts this and invalidate the line, it will cause an error and the assumption information is no longer valid.

In addition to keeping track of the status of the line that will be read in the future, there also exist assumption information about the destination cache entry (where the line will be written to in the future). Namely, the cache controller is going to find

an memory location to fulfill the cache request, and that memory location will be reserved and untouchable by other processes until the current process is finished with the fulfillment of the request.

Part of the design that keeps one process from interfering with another process is through the use of locks and conditions for execution of the hardware. This part of the TDV model is necessary to prove that assumption information remains unchanged. In addition, the locking and mutual exclusivity algorithm implemented by the design will also need to be verified. TDV is also used to verify both of these properties.

5.1.4 Manual Reasoning of Correctness

This section will describe how the accepting rules in section 5.1.1.2 will keep one process from interfering with each other. One process interfere with another when one process changes the state variable (either shared or private) such that the assumption information are also changed.

In the cache controller, the most problematic case is trying to maintain the assumption information about the status of the cache tag. Many of the execution decision for a process is dependent on the status of the tag, thus the status is the assumption information for many of the blocks. There are also numerous blocks which can modify the cache tag, either in a cache which have the same address as the process request, or in a cache with the same index as the process request. The verification task is to make sure that one process does not change the tag of a cache line that is currently being accessed by another process.

As an example, consider the verification task that the tag status from another tile with the same address will remain the same until data have been read from it (neighbor tag property). A parallel verification task is that the tag status of a reserved line will remain the same until the current process finishes updating the data (destination tag property). The blocks which can violate this assumption information are blocks which writes to the cache tags: “Search & tag update, Evict”, “Reserve line”, “Reserve Line w/ cache op”, and “Destination Tag Update.” The “Destination Tag Update” can modify cache tags if it is accessing the cache with the same address,

while the blocks "Search & tag update, Evict", "Reserve line", and "Reserve Line w/ cache op" can modify cache tags with the same index. The process which the assumption information should hold will be called the *interrupted process*, and the process which will be executing one of the above block which may change the cache tag will be called the *interruptee process*.

Assume that no two processes with the same index can be in the index locked region at any time (index exclusivity assumption), no two processes with the same address can be executing in the non-outstanding region at the same time (address exclusivity assumption), and no two request from the same tile can have the same index active at the same time (index same tile exclusivity assumption). The regions are as marked in figure 5.2. The proof of maintaining the cache tag after read can be divided up into the following cases:

1. Consider the interrupted process that is about to execute a block that is in the index locked region. Since no other processes is can also be in the index locked region with the same index (index exclusivity assumption), the only other process that might cause trouble is if the interruptee process executes "Destination Tag Update" because "Destination Tag Update" is the only block which can write cache tags that is outside the index locked region. However, the address exclusivity assumption ensures that the interrupted and interruptee process does not have the same address and thus cannot change the tag status from another tile. Thus the neighbor tag property is satisfied. For destination tag property, note that the "Destination tag update" will only modify the destination cache line's tag. And index same tile exclusivity assumption ensures that two processes requesting from the same tile cannot have the same index. Since the "Destination Tag Update" can only update cache lines with the same index, the destination tag property is maintained.

2. Consider a process that is about to execute a block that is outside the index locked region. The neighbor tag property is no longer need to be valid because according to the TDV model the neighbor data is read before process exits the index locked region. Address exclusivity assumption implies that the interruptee

process have a different address than the interrupted process. "Reserve Line", "Reserve Line w/ cache op", and "Search & tag update, Evict" can all change the tag information of the cache tag for different address but the same cache index. However, the only tag information it may change is the cache line from the requesting tile. Index same tile exclusivity assumption states that if two processes request are from the same tile, then their index must be different. Thus it is not possible for the interruptee process to change the tag of another process with the same index.

3. The above condition have an exception: A process in the outstanding state may be interrupted by another process originated from the memory controller. And these two processes may have the same address. However, if the process in is the outstanding state, it means that the tag of the cache line of interest is in reserved or shared state (TDV model will change it to reserved or shared state before it puts the process into outstanding state). No request from the memory controller can turn a reserved state tag line into something else, and if the request from the memory controller can turn the shared tag into invalid, it will also change the assumption information of any other process with the same address. In another words, the second process will be notified of such a change and when it exits the outstanding state, it will perform the appropriate action based on the new information.

This conclude the reasoning why one process is not going to interfere with the assumption information of another process inside the cache controller. Note that there are many details that still requires to be expanded when it is represented by the TDV model. These things include making sure that the locking regions corresponds correctly to the locks being turned on and off by the cache controller for a process, and appropriate state modification and information gathering by the blocks. Some of these details have a clear correspondence to the design hardware, and things such as locking regions require further proof by the TDV to reduce the description that corresponds to the underlying hardware.

5.1.5 Using TDV

The manual reasoning above gives the basis for forming the pre-conditions and post-conditions in the TDV model. The assumption information in the manual proof above are the pre- and post-conditions of selected blocks in the TDV model. For example, the neighbor tag property is valid from the point where the neighbor tag information is read until when the neighbor data have been read by the cache controller. Thus, all the blocks in the TDV model between "Search & Tag update, Evict" and "Read Cache Line" will have the neighbor tag property as its pre-condition and post-condition. Similarly, the destination tag property will be the pre- and post-condition of all blocks between "Reserve Line" or "Reserve Line w/ cache op" and "Destination Tag Update." Various other pre- and post-conditions mostly have to do with the status of the index lock is also added to the TDV model.

The actions that are required and designed in the cache controller are easy to form and is also incorporated into the TDV model. They correspond mostly to the function of the block in the cache controller design.

5.1.6 Bugs found and corrected

There are two categories of design flows that was found using this technique. One class of bugs is when the underlying hardware and thus transaction diagram representing it contains a bug. This is the type of bugs that TDV will find when it is verifying the transaction diagram. The other kind of bugs comes from the hardware not following the action blocks of the transaction diagram or does not have atomic behavior. Both of these bugs are catchable with the TDV modeling methodology.

5.1.6.1 Transaction Model Bugs

Eviction Writeback Lock after completion One of the design bug that was found using this technique is the potential for incorrect fetch of data due to improper release of the index lock. The situation occurs when the index lock is released before the eviction writeback task completes, which can occur in the original design. If the index lock is released early, another request with the same address can execute, find

that the line is not in the cache (because it is in the process of being evicted), and return that the line does not exist inside the quad. But this is incorrect since the line does exist, although not in the caches, the line exist in the cache controller during the process of writing back to the memory. This incorrect reporting will cause the requesting process to fetch the incorrect data from main memory instead of waiting until the writeback is complete before fetching the data.

Degrade notification TDV modeling also found that the original design violated the invariant that forbids state changed by another process. This invariant is violated when the interrupted process is in the outstanding state, thus an interrupting process request from the memory controller with the same address is allowed to execute and change the state. As mentioned in the third case of the manual reasoning in section 5.1.4, the interrupting process is responsible to notify the interrupted process if it have made a change in the tag. The original design did not notify the interrupted process of this change, thus the interrupted process will end up with the wrong state information, and thus the correct data will not be fetched.

5.1.6.2 Matching Transaction Model with Design

Unlock at the beginning of D-unit rather than at the end of S-Unit Another situation where one process is able to change the content of another process in the original design where the index lock is being unlocked too early. In the original design, unlock occurs after the Search & tag update operation is completed by the S-Unit.

D-unit operation are represented in TDV with the blocks “WB Read”, “Read Line”, “Write Line”, “SyncOp”, and “Word Access”. Unlocking the index after the S-Unit completion effective place the D-unit operation blocks in the index unlock state.

Although the need to place TDV D-unit blocks inside the index locked state is obvious in the TDV model, it is hard to understand why this is necessary without constructing the TDV model. This is due to the fact that the D-unit operates in a FIFO manner, so the designers did not expect an interruption by another process is

possible. This assumption is actually true if the unlock occurs at the beginning of the D-unit rather than the end of the S-unit, which is the fix for this bug.

This problem was caught when we attempt to correspond the TDV model to the actual design. Without moving the unlock command to the beginning of the D-unit, the TDV model with the D-unit blocks being in the index locked state does not correspond to the actual design. Even with the correction in place, the natural interpretation of the TDV model requires that the index unlocks at the end of the D-unit blocks. However, using the FIFO property of the D-unit, it can be shown that unlocking at the beginning or at the end gives the same consequent behavior with respect with other interrupting properties. Thus one can conclude that the TDV model can match the implementation with the additional FIFO property assumption.

Reserve Line w/ cache op requires previous result An incorrect reading of the tag information within a process was found during the matching process which can occur when reserving a cache line requires re-reading of the tag information from the requesting cache (occurs in the "Reserve Line w/ cache op" stage). The reason the destination cache need to be re-read is that there is a possibility of tag information being changed between the previous stage and the current stage.

However, in the design, the designers thought of this operation as the same as redoing the previous step (the "Search & tag update" step, with the additional operation of evict. This turn out to be a problem as the previous step is a destructive step. That is, the "Search & tag update" step may change the tag information in addition to the read. Thus, any repeat of the previous operation will be incorrect, as the tag information have been changed.

The reason why TDV model was uniquely suited to catch this design flaw is that by design, the need to perform a re-read does not occur often in a real testing environment. And even if a re-read occurs, most of the time, the erroneous data will only cause a re-fetch of the cache line from memory, thus the overall data operation is still correct, although performance is impacted. Yet, there are few cases where the incorrect data will be fetched, but this corner case is not guaranteed to be exercised by simulation test. This design flaw occurred as the designers are fixing another

corner case, thus the correctness of the change with respect to the whole design is not considered fully. The construction of the TDV model and the need to match it provides a systemic way to verify that all design changes are considered for correctness with respect to the overall protocol.

The solution to this bug to perform a re-read on the reserving cache line at the "Reserve Line w/ cache op" stage, but the tag from other caches are read only during the "Search & tag update" stage. The result of this read must be forwarded to the "Reserve Line w/ cache op" stage, so that a re-read is not performed.

Clear Valid flag only after all sub-processes have completed The valid flag's function is to prevent processes which will act on the same address from executing in the cache controller at the same time. A design flaw was found using the TDV model which allows the valid flag to be turned off before all the execution of a process have been completed.

This problem occurred in the design because it is possible for each process to fork into different parallel threads that are going to different processing units. The original design turns off the valid flag when the main thread finishes. However, the main thread finishing does not guarantee that other forked thread have finished. When the valid flag is turned off, it allows other processes with the same address to start executing in the cache controller. In some instances, this can cause problems and conflict with the existing process.

The fix is to make sure that all the different forked threads have a way to notify the main thread that it is finished. The valid flag cannot be turned off until all the threads have been completed.

5.2 Cancel Operation

5.2.1 Protocol Description

The cancel operation is part of the memory synchronization operations (SyncOps) that are used as semaphores or enables the programming of the TCC protocol. The

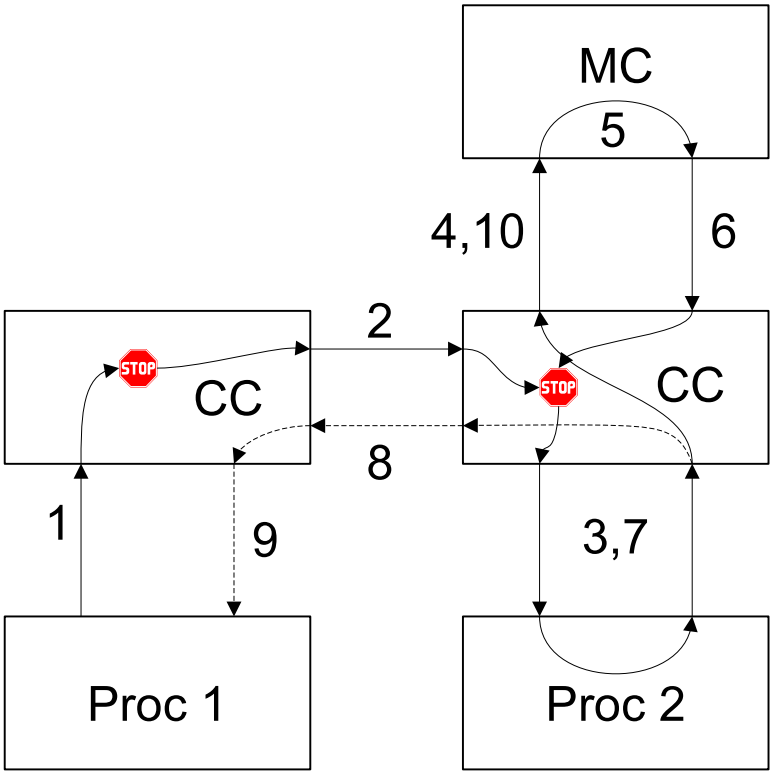


Figure 5.3: Synchronization Command Paths

syncop commands can either be executed right away or must be held until the execution condition are correct. The commands that are being held for execution are called *waiting process*. The system must keep track the waiting processes so that it can enable them (wake up) when the conditions are correct. The cancel command clears the memory system of all the syncop requests from a particular processor. To clear the memory system, it not only needs to clear the database of waiting processes, but it also need to clear the syncops that are currently active in the system. A syncop command is called active if it is currently in the system which more processing can be done without waiting for an outside condition. Assume that any processing of a request can be done in one cycle (take it off the input queue, process it, put it on the next queue in one cycle), then an active process could located in the queue linking from one block to another.

The term *path segment* will be used to refer to the following queues:

1. SyncOp request/syncmiss message from processor to CC
2. SyncOp request message from CC to CC
3. In CC waiting for line to be fetched to local cache
4. Syncmiss message sending from CC to MC
5. Waiting in SyncOp queue in MC
6. SyncReplay message sending from MC to CC
7. In CC performing the SyncOp operation
8. SyncSuccessful message from CC to CC
9. SyncSuccessful message from CC to Processor
10. SyncWakeup message sending from CC to MC

The cancel operation, when completed, will ensure that there are no syncop commands in the system from a particular processor. This operation is complicated by the fact that the processor are free to generate new syncop requests, and old requests may always be active and looping in the system. The cancel protocol also will need to make sure that requests which have completed their operation is not dropped, but is able to return the result to the requesting process.

There are three modes of operation that will affect the pathways of syncop command and requests. The first mode is to cancel requests for memory addresses on the cache coherent region. Figure 5.3 shows the pathway that a syncop command of this mode may follow. The figure also demonstrates that a syncop request may loop in the system. This loop is caused by a consistent failed replay and consistent wake up by another processor's syncop.

Because each queue is FIFO, if no syncop command enters the queue after the cancel enters the queue, then when cancel exits the queue, the queue is free of syncop commands. Thus, if the cancel process can do this with all the queues, then it can guarantee that the system is free of syncop commands. To do so, the cancel command

will follow the pathway of the syncop command while maintaining that no new syncop commands can be injected into the system. The location which is marked "STOP" in the diagram are the places where the hardware can ensure that any syncop requests will be dropped and will not enter the next queue. The cancel command will activate or cancel the dropping mechanism, depending on the state of the command.

5.2.2 TDV Model

Although the description and the design of the cancel protocol doesn't derive from a transaction diagram, the transaction diagram can be used to model the interesting properties concerning the successful completion of the cancel command. This TDV can be used to find bugs in the cancel protocol design by building the appropriate cancel protocol TDV model.

The cancel protocol can be modeled as a series of states which represent the clearing status of the memory system, which are blocks in TDV model. The clearing status indicates which path segments are cleared of the syncop commands. Thus, each block's action and post-condition is that a certain path segment is cleared of syncop commands. Each block will also represent which path segments have already been cleared in order to reach the current execution block. This information is encoded in the TDV model as pre-conditions of the block. Upon the end of the cancel execution, the ending block should have the pre-conditions that all path segments are cleared, which is the property of interest.

The TDV model also contain blocks which represent the insertion and the movement of syncop commands. Each of these syncop command blocks represent part of the circuit taking a syncop command from a queue, performing an action, and putting it on another queue. Each of these blocks also contains guards so that the dropping behavior is modeled when the cancel protocol turns on the drop flag.

TDV checks the interaction of the cancel protocol blocks with the syncop command blocks to make sure that the property all path segments are cleared when cancel protocol finishes.

5.2.3 Design Constraints Found

5.2.3.1 Allow syncop completion

One potential bug that was found during the construction of the TDV model was the discovery that syncop operations that have been completed needs to be acknowledged by the processor instead of being canceled through the cancel protocol. This bug actually cannot be detected with the TDV model because it is not just a protocol issue, but arises from the an interaction problem with the user programming level.

In order for the system to be deadlock free, the number of sync loads commands issued need to equal to the number of sync store commands. This constraint is to be satisfied at the application level, that is, the user must make sure that the application is written so that the number of sync load equal to the number of sync miss. However, this requirement at the programming level itself is not enough. The hardware need to ensure that all sync loads and all sync stores are executed to completion if it is executed at all. The original design of the cancel protocol does not guarantee this condition.

This design flaw is hard to catch because on the application level, the syncop commands are re-issued after a cancel because cancel is supposed to drop ongoing sync op commands. However, if the syncop have already been completed, this information needs to be convey back to the user so that the command will not be re-issued upon the restart of the application.

Finding this bug demonstrates that by constructing a simplified model of the design, this simple model can be used to analyze the interaction between different programming models. Just like parallel processes can cause unexpected bugs when put together, different usage levels of the design can cause unexpected bugs when used together. The simplified design allow the verification engineer to analyze the nature of the interaction between different programming levels and thus make finding the correct constraints to verify and finding bugs easier.

5.2.3.2 Pitfall avoided

Although no bugs on the design itself was found using the TDV tool, this is due to the fact that the design is not yet completed during the time the TDV tool was used on the design. By having a verified TDV model, it is able to produce constraints so that when the designers refine their design, they can have guidelines that will prevent them from introducing bugs, which the designers did fine helpful.

One such constraint is derived from the TDV block where cancel is accepted from the processor (path segment 1) by the cache controller, and the cache controller will send this cancel on the replay path (path segment 7). This is also represented by the "STOP" in the diagram where the cancel command will flip a flag that instruct the dropping of incoming syncop commands. Since TDV model's block are atomic, the receiving of the cancel command, turning on the drop flag, and sending out of the cancel command on another path must have atomic behavior. Being able to represent the cache controller such that atomic behavior is observed requires certain parts of the cache controller having FIFO behavior. This FIFO behavior is not immediate obvious as a requirement as it is part of the computational data path. But the need for the FIFO constraint on part of the data-path was made clear through the construction of the TDV Model. So having a clearly defined constraint through the TDV modeling process helps the designers to avoid introducing bugs into the design.

5.3 Unsuccessful TDV efforts

5.3.1 Cache Coherency Protocol

The cache coherency protocol verification refers to verifying that the interaction between the processor, cache controller, and the memory controller will maintain the MESI protocol with the caches on the system. This verification have been previously done using theorem prover PVS to argue about the structure of the state space in conjunction with SMV to prove that the structure is maintained. Even though this exercise found bugs in the design, this verification effort was a painstaking long process.

It was hoped that TDV is can be used to verify the cache coherency protocol more quickly. However, it was discovered that during the modeling phase of TDV, the same problem that was encountered of dealing with memory hierarchy in the PVS proof was also present in the TDV model. Further analysis of the problem revealed that using TDV to verify the cache coherency protocol will be as complex as using PVS and SMV to verify the protocol as it was done previously. This is because the complexity of the cache coherency protocol at this level of description does not come from complex interactions between different processes. Rather the correctness derives from the fact that each local node have a special relationship between it and the neighboring nodes, and that local changes will maintain the global invariant. Thus TDV does not offer any special advantage over existing tools in solving the cache coherence correctness problem.

5.3.2 TCC Protocol

The TCC Protocol verification made sure that the design follows the transaction programming model. This includes properties such as, "when a process commits, the resulting change will effectively be seen by all the processes at the same time." In the TDV model, each process keeps a copy of the "ideal memory" in addition to the state of the real memory representing the cache of the system. The behavior of the ideal memory reflects the actions of the abstract transaction programming model. That is, any changes to the local process's memory is not reflected in the ideal memory until it commits. When it commits, the changes are instantaneously reflected in all processes' memory space (their ideal memories). It also keeps track of any memory that was read by the process, and ensures that process cannot commit if the memory content previously read is different from the current ideal memory. Thus, the the process must be restarted if the read memory is different from the current ideal memory content. Making sure that the memory read content is the same as the current ideal memory if committing is one of the invariant to verify in the TCC TDV model.

There was difficulty using TDV to verify this above invariant. This is due to various model extensions which reflect the real design. These extensions make the

invariants required to prove the correctness complicated. An example of a model extension occurs in the modeling of commit in the TDV model. During a commit, the process in the design will update the value to the other caches for each address independently. Thus, at a given point in time, the cache may contain updated commit data or old data that is yet to be updated during the time when another process is performing an update. However, in order to verify the invariant, the location of the "ideal data" must be tracked to verify that this ideal data is not corrupted. Having the ideal data located at different points for different addresses complicates the tracking of ideal data in the system. This makes the formulation of the invariant in order for verification to succeed hard.

Another complication is that the invariant "read memory is the same as the ideal memory." have exceptions. The exception occurs when the read occurs after a write for the same process on the same address. In this case, the read value is not the ideal memory value as seen by all the processes, but the read value should be the value of the last write on the same process. These complications makes the invariant for the TCC protocol hard to model in TDV. TDV is suited for handling parallel processes that are complex if the single process execution is simple. In this case, the model for single process execution is not simple.

There are several modification to the current TCC model that will make the verification more manageable in TDV. One is to break up the TCC model into several verification steps, TDV can be used to verify the TCC implementation with simple single process description, while other formal techniques can be used to verify that the single process description is being implemented correctly.

A second possible way to simplify TDV model is to extract and verify the assumptions of the TCC protocol instead of also verify the TCC protocol at the same time. The TCC protocol have been designed and proved to give the correct memory model given a certain abstracted behavior of the system. Instead of re-verifying this proof using TDV as it is done with the current effort, a simpler TDV model can be used that verify the implementation adheres to the protocol.

Third is that given enough time, TDV is able to verify the TCC model, however, the trade-off between the advantage of TDV verification verses time spent on it must

be balanced. In this case, the TCC verification project was not started after the design phase, the interest in verifying the TCC implementation was not great enough to warrant spending additional time on the verification effort.

Chapter 6

TDV Correctness

This Chapter will show that the verification conditions for TDV is sound in proving the correctness of the invariant properties in the TDV model. The proof will be done by specifying the TDV model in first order logic in a transition system and showing that the invariants can be derived from the proof obligations proved by the TDV tool.

6.1 User level TDV modeling

6.1.1 Model Expression Representation

The state of a TDV system is divided into private variables, which are states that are unique to each transaction, and shared variables, which are states that are shared among different transactions. Thus, the state of the TDV system is said to be composed of $pri[i] : PRIV\_STATES$, which is an array of private state variables where i is the transaction identifier, and $shr : SHR\_STATES$, which is the shared state variables.

The expressions in TDV model can reference to the variables of one process, and one other arbitrary process. Since the choice of the process are arbitrary, when TDV model write expressions such as $pri.lock \wedge \neg po.lock$, it is really expressing the first order formula $\forall i, j.(i \neq j \rightarrow pri[i].lock \wedge pri[j].lock)$, where i refer to one process, and j refers to one other arbitrary process that is different from i.

In general, TDV expressions are in a predicate of the form $P(shr, pri, po)$. And this is modeled in first order logic as

$$\forall i, j.(i \neq j \rightarrow P(shr, pri[i], pri[j])) \tag{6.1}$$

As an example, TDV can express the idea that only one process have the lock by with the first order logic equation of:

$$\forall i, j(i \neq j \rightarrow \{pri[i].lock \rightarrow pri[j].lock\})$$

which can be expressed in the TDV model by defining P to be

$$pri.lock \rightarrow po.lock$$

As a side note, all private variables used in the TDV model can also be modeled as shared variables. For example, consider expressing the private variable *pri.lock* as a shared variable. This can be accomplished by constructing an shared variable array $shr.lock[idx]$ where each process can access "its" private variable *lock* using the process id as the index. (e.g. $shr.lock[pri.id]$). However, even though that from a theoretical point of view, TDV can handle expressions not restricted by the above form by making variables as shared variables, doing so will result in a more complex model and harder to verify, it is not practical to do so in a real verification usage environment unless the resulting expansion is kept simple.

6.1.2 Model Expression Representation with Exclusive Property

With the addition of exclusive property process access, TDV can express certain properties not possible with equation 6.1. Namely, TDV can use expressions that refers to an transaction with the exclusive property in addition to the two processes already named.

In this section, adding the reference to the transaction with one exclusive property

will be explained, then the expansion to a more generalized form of allowing multiple exclusive property will be explained.

The exclusive property can be expressed as

$$\forall i, j.[i \neq j \wedge excl(pri[i]) \rightarrow \neg excl(pri[j])] \tag{6.2}$$

that is, no two transactions can both have the exclusive property at the same time. This exclusive property is defined by the user, and the user also associates a prefix along with the particular defined exclusive property. Now TDV may express equations of the form

$$\begin{aligned} & \forall i, j.[i \neq j \wedge excl(pri[i]) \rightarrow \neg excl(pri[j])] \\ \wedge \quad & \forall i, j, x[i \neq j \wedge excl(pri[x]) \rightarrow P(shr, pri[i], pri[j], pri[x])] \end{aligned} \tag{6.3}$$

the first conjunction clause states that only one transaction have the exclusive property at a time, and the second clause states that transaction represented by x is the exclusive transaction.

Suppose that there is no transaction with the exclusive property in the system. In this case, an arbitrary transaction can be created as a placeholder with the exclusive property. Since state of this transaction not related to the exclusive property is non-deterministic, this is a safe generalization of the system under test.

To see that equation 6.3 is a more general form of equation 6.1, let the exclusive property refer to the placeholder transaction only. That is, the exclusive property will be *true* if it's the placeholder transaction, *false* otherwise. The first term $\forall i, j.[i \neq j \wedge excl(pri[i]) \rightarrow \neg excl(pri[j])]$ reduces to *true* by the definition of the placeholder transaction. To eliminate the $\forall x.$ term, consider when x does not refer to the placeholder transaction. In this case, $excl(pri[x])$ is *false*, thus the implication expression reduces to *true*. Thus the only case under consideration is when x refers to the placeholder transaction. Since the placeholder transaction is non-deterministic, that is, the states of $pri[x]$ is non-deterministic, thus the predicate $P(shr, pri[i], pri[j], pri[x])$ should be independent of $pri[x]$, or that

$P(shr, pri[i], pri[j], pri[x]) = P(shr, pri[i], pri[j])$. This reduces equation 6.3 to 6.1 when exclusive property extension is not used.

In general, more than one exclusive property may be used in TDV. The generalize form of a property in TDV is of the form

$$\begin{aligned} & \forall i, j.[i \neq j \wedge excl1(pri[i]) \rightarrow \neg excl1(pri[j])] \\ \wedge \quad & \forall i, j.[i \neq j \wedge excl2(pri[i]) \rightarrow \neg excl2(pri[j])] \\ \wedge \quad & \cdots \\ \wedge \quad & \forall i, j, x_1, x_2, \cdots [i \neq j \wedge excl1(pri[x_1]) \wedge excl2(pri[x_2] \wedge \cdots) \rightarrow \\ & \qquad P(shr, pri[i], pri[j], pri[x_1], pri[x_2], \cdots)] \end{aligned} \tag{6.4}$$

where $excl1$, $excl2$, etc are independent exclusive properties, and P is defined by the user through TDV.

6.1.3 Guards, Pre- and Post-conditions

The transition blocks of the TDV model consists of pre- and post-conditions, guard, and action. The pre- and post-condition and guards of the transition blocks are predicates on the shared variables states and two private variable states (self and one arbitrary other process's private variable state). These predicate defines the predicate P used in equation 6.4.

Suppose the transition block is named B, then the guard of that block is denoted by

$$B.guard(s_{shr}, s_{pri}, s_{po}, s_{excl1}, \ldots)$$

where s_{shr} represent shared variables, and s_{pri}, s_{po}, s_{excl1},... represent private variable assignments. Suppose that $A.guard$ is instantiated with $A.guard(shr, pri[i], pri[j], pri[x])$, where $pri[i]$ is the private state variable of process i, $pri[j]$ is the private state variable of process j, and $pri[x]$ is the private state variable of the process x. The interpretation is that process i is poised to execute block A and process j is any other process that's not i, and process x is the process which have the exclusive property. The system will be allowed to execute process i only when $A.guard(shr, pri[i], pri[j], pri[x])$

is true.

Similar expressions are given for pre- and post-conditions of a transition block

$$B.pre(s_{shr}, s_{pri}, s_{po}, s_{exl1}, \ldots)$$

and

$$B.post(s_{shr}, s_{pri}, s_{po}, s_{exl1}, \ldots)$$

In TDV, s_{shr} variables are prefixed by *shr.*, s_{pri} variables are prefixed by *pri.*, s_{po} variables are prefixed by *po.*, and $s_{exl?}$ variables are given the prefix assigned by the user when the user defines the exclusive predicate.

6.1.4 Assignments

The assignment of a transition block describes the behavior of the block when it executes. It is composed of a boolean expression and a set of variables, named *action boolean expression* and *action variable set*. The execution of the transition block will assign the variables in the action variable set such that action boolean expression will be true. We will denote the action boolean expression as $A.assign(s_{shr}, s_{pri}, s_{po}, t_{shr}, t_{pri}, t_{po})$, where t_{shr}, t_{pri},t_{po}represent the state after the block have been executed.

The prefix representing s_{shr}, s_{pri}, s_{po} are *shr. pri.*, and *po.*, the prefix for t_{shr}, t_{pri}, t_{po} in the action boolean expression expression are given the prefix *shr′.*, *pri′.*, and *po′.*.

For example to write that the execution of a block will increase count by 1, the action boolean expression will be

$$pri'.count = pri.count + 1$$

and the action variable set is $\{pri.count\}$. Substituting this expression into equation **??** gives the transaction equation governing the behavior of the system:

$$\exists i.\forall j.(i \neq j \rightarrow (pri'.count = pri.count + 1))$$

6.1.5 Exclusive Condition

The actual representation of model expression of TDV is actually in the form of

$$\forall i, j, x.[i \neq j \rightarrow (excl(pri[x]) \rightarrow P(shr, pri[i], pri[j], pri[x]))] \tag{6.5}$$

which is similar to the form without the exclusive property. However, when the exclusive property is defined, TDV will automatically generate the equation 6.2 as pre- and post- conditions for all transition blocks in the system. That is, the property

$$excl(pri[i]) \rightarrow \neg excl(pri[j])$$

will be added to the pre- and post-conditions. Or that

$$\begin{aligned} P(shr, pri[i], pri[j], pri[x]) \quad \equiv \quad & excl(pri[i]) \rightarrow \neg excl(pri[j]) \\ & \wedge Q(shr, pri[i], pri[j], pri[x]) \end{aligned}$$

. If the second term Q is dropped, equation 6.5 reduces to

$$\forall i, j, x.[i \neq j \rightarrow [excl(pri[x]) \rightarrow (excl(pri[i]) \rightarrow \neg excl(pri[j]))]]$$

which is logically equivalent to

$$\forall i, j.[i \neq j \wedge excl(pri[i]) \rightarrow \neg excl(pri[j])]$$

if there exist a process with the exclusive property (see section 6.1.2 to see how TDV model can always guarantee this). This is the first part of equation 6.3.

To see how the second part of equation of 6.3 can be derived from equation 6.5, one simply notes that there always exist a process with the exclusive property, and keep the second term Q. Thus equation 6.3 is represented by the TDV model when TDV automatically generate the exclusive condition as part of the pre- and post-conditions for all transition blocks in the system.

6.2 Internal TDV modeling

6.2.1 Property Blocks

The TDV model is represented by defining a predicate P as it appears in equation 6.3. However, in the TDV model, P is expressed on a per block basis. That is, the user specify a property for a particular execution block, and the specified property only applies if process i is about to execute that block. Or more formally

$$\begin{aligned} P(shr, pri[i], pri[j], pri[x]) \quad \equiv \quad & pri[i].pc = BlockA \rightarrow BlockA.P(shr, pri[i], pri[j], pri[x]) \wedge \\ & pri[i].pc = BlockB \rightarrow BlockB.P(shr, pri[i], pri[j], pri[x]) \wedge \\ & \ldots \end{aligned} \tag{6.6}$$

where $pri[i].pc$ is the "program counter" that points to the block that process i will execute next, and $BlockA.P$, $BlockB.P$, ... are the properties defined for each block This expansion of P represent that if a process is about to execute $BlockX$, then $BlockX.P$ will describe the property of this process. This is why in equation 6.3, i is given the interpretation as the process that is about to execute.

6.2.2 Frame Condition

The frame condition $Frame(s, t, i, j, x)$ represent that all variables not in the variable action set from the assign statement will not change its value, where i is the executing process. In particular, the pc will not change if j is not the executing process, that is, $i \neq j \rightarrow s.pri[j].pc = t.pri[j].pc$. The same can be true for the arbitrary process x, thus this equation follows from the frame condition.

$$\begin{aligned} Frame(s, t, i, j, x) \qquad & i \neq j \rightarrow s.pri[j].pc = t.pri[j].pc \wedge \\ & i \neq x \rightarrow s.pri[x].pc = t.pri[x].pc \end{aligned}$$

6.3 Proof

The goal of the proof is to prove that the pre- and post-conditions holds for all executions. The pre- and post-conditions will be represented by the predicate $INV(s_t)$, where s_t is the state of the system at time t. Then the goal is to show that $\forall t.INV(s_t)$ when s_t follows the transition behavior as described by the TDV model. The following sections will expand the representation of INV and the transition behavior of s_t to the equivalent representation of the TDV model.

6.3.1 Transition System

s_t represent the state of the system at time t. The transition system description consists of an initial state s_0 and a transition relation $TRANS(s_t, s_{t+1})$ which is a predicate of the allowable transition from time t to time $t+1$.

To show that the invariant holds for all execution at all times ($\forall t.INV(s_t)$), it is sufficient to show that

1. $INV(s_0)$
2. $\forall s_t, s_{t+1}.(INV(s_t) \wedge TRANS(s_t, s_{t+1}) \rightarrow INV(s_{t+1}))$

$\forall t.INV(s_t)$ is then true by induction using (1) and (2).

6.3.2 Expansion of INV and $TRANS$

In terms of TDV expression representation $INV(s_t)$ is represented as

$$\begin{aligned} INV(s_t) \equiv \quad & \forall i, j.[i \neq j \wedge excl(s.pri[i]) \rightarrow \neg excl(s.pri[j])] \\ \wedge \quad & \forall i, j, x[i \neq j \wedge excl(s.pri[x]) \rightarrow \\ & \forall B.[s.pri[i].pc = B \rightarrow \\ & B.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x] \\ &]] \end{aligned} \tag{6.7}$$

The first conjunction term is the exclusive property–that only one process can have the exclusive property at a time. i represent the process which will be executed next, and j represent any other arbitrary process. Thus the term $i \neq j$ distinguish the executing process from the arbitrary process. Process x represent the process with the exclusive property, which is represented by the $excl(s.pri[x])$ term.

Similarly, $TRANS(s,t)$ is expressed as

$$\begin{aligned} TRANS(s,t) \equiv \quad & \exists i.\forall j,x.i \neq j \rightarrow \forall B. \quad [s.pri[i].pc = B \rightarrow \\ & B.Guard(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\ & B.Assign(s.shr, s.pri[i], s.pri[j], s.pri[x] \\ & \qquad t.shr, t.pri[i], t.pri[j], t.pri[x]) \wedge \\ & Frame(s,t,i,j,x) \end{aligned}$$

6.3.3 Base Case

The base case is automatic when the first starting transition block have no pre-conditions. To see this, consider the expression

$$\begin{aligned} \forall B.[\quad & s.pri[i].pc = B \rightarrow \\ ] \quad & B.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \end{aligned}$$

which appeared in equation 6.7. $s_0.pri[i].pc$ should refer to B_0, the starting transition block, thus the expression reduces to

$$B_0.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x])$$

and since the first transition block have no pre-conditions, this expression reduces to $true$.

The user does need to verify that the initial state of the system will only have one exclusive property. This is usually trivial and can be passed on as the requirement for the next abstraction level of verification.

6.3.4 TDV verification equations

TDV constructs the equation representing the verification ideas presented in section 4.3.1. PVS is used to prove the equations expressed below can be used as assumptions to prove

$$\forall s_t, s_{t+1}.[INV(s_t) \wedge TRANS(s_t, s_{t+1}) \rightarrow INV(s_{t+1})]$$

The first verification idea, "if the pre-condition is satisfied when a process executes a block, then after the execution, the post-condition is satisfied." is represented by

$$\begin{aligned}
\forall s, t.\forall i, j, x.\forall B.[\quad & i \neq j \wedge \\
& exl(s.pri[x]) \wedge \\
& exl(t.pri[x]) \wedge \\
& B.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Guard(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Assign(s.shr, s.pri[i], s.pri[j], s.pri[x], \\
& \qquad t.shr, t.pri[i], t.pri[j], t.pri[x]) \wedge \\
& Frame(s.pri[i], s.pri[j], s.pri[x], t.pri[i], t.pri[j], t.pri[x] \wedge \\
& B.Post(t.shr, t.pri[pri], t.pri[po], t.pri[pA']) \\
]
\end{aligned} \tag{6.8}$$

pri represents the process under verification and the executing process, po represents any other process, and pA and pA' represent the process which have the exclusive property. To denote that pA' is the process which have the exclusive property after the transition, the term $Exl(t_{pA'})$ is added to the antecedent of the implication equation. Note that pA and pA' are not the same process.

The second verification condition, "If the execution of a process passes from block A to block B (there exists a link between block A and block B), then the pre-condition of block B can be implied from the post-condition of block A." is represented by

$$\forall B1, B2.\forall s.\forall i, j, x \qquad Link(B1, B2)$$

$$
\begin{aligned}
& Exl(s.pri[x]) \wedge \\
& B1.Post(s.shr, s.pri[i], s.pri[j], s.pri[x]) \\
& B2.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x])
\end{aligned} \tag{6.9}
$$

where $Link(B1, B2)$ indicate that block $B2$ may execute after $B1$.

And the third verification condition, "For a process a about to execute block A for which the pre-condition holds, if another process b interrupts and executes another block B, the pre-condition of block A will still hold even after the execution of process b." is represented by

$$
\begin{aligned}
\forall s, t. \forall i, j, k, x. \forall B1, B2. \quad & i \neq j \wedge i \neq k \wedge \\
& exl(s.pri[x]) \wedge \\
& exl(t.pri[x]) \wedge \\
& Frame(s.pri[i], s.pri[j], s.pri[k], s.pri[x] \\
& \qquad t.pri[i], t.pri[j], t.pri[k], t.pri[x]) \\
& B1.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B2.Pre(s.shr, s.pri[k], s.pri[i], s.pri[x]) \wedge \\
& B2.Guard(s.shr, s.pri[k], s.pri[i], s.pri[x]) \wedge \\
& B2.Assign(s.shr, s.pri[k], s.pri[i], s.pri[x], \\
& \qquad t.shr, t.pri[k], t.pri[i], t.pri[x]) \wedge \\
& B1.Pre(t.shr, t.pri[i], t.pri[j], t.pri[x])
\end{aligned} \tag{6.10}
$$

$Exl(s_{pA})$ denotes that process pA is the process with the exclusive property before executing $B2$ and $Exl(t_{pA'})$ denotes that process pA' is the process with the exclusive property after executing $B2$.

6.3.5 TDV verification

TDV takes the specification given in the transition blocks and construct the boolean expression that will be described in this chapter and verifies that it is valid using the

STP decision procedure. [11] This section will also described the steps used to convert the boolean expression verified by STP into the assumption equations described in the previous section.

The block execution correctness verification boolean expression as verified by STP is

$$
\begin{aligned}
& exl(pA) \wedge \\
& exl(pA') \wedge \\
& B.Pre(shr, pri, po, pA) \wedge \\
& B.Guard(shr, pri, po, pA) \wedge \\
& B.Assign(shr, pri, po, pA, shr', pri', po', pA') \wedge \\
& Frame(pri, po, pA, pri', po', pA') \\
& B.Post(shr', pri', po', pA')
\end{aligned}
$$

where pri, po, pA, pri', po', pA' are uninterpreted variables for the private states and shr and shr' are uninterpreted variables for the public states. Since these variables are uninterpreted, the above equation can be universally quantified over these state representations:

$$
\begin{aligned}
\forall shr, pri, po, pA, shr', pri', po', pA' \quad & exl(pA) \wedge \\
& exl(pA') \wedge \\
& B.Pre(shr, pri, po, pA) \wedge \\
& B.Guard(shr, pri, po, pA) \wedge \\
& B.Assign(shr, pri, po, pA, shr', pri', po', pA') \wedge \\
& Frame(pri, po, pA, pri', po', pA') \\
& B.Post(shr', pri', po', pA')
\end{aligned}
$$

Instantiating $shr \equiv s.shr$, $pri \equiv s.pri[i]$, $po \equiv s.pri[j]$, $pA \equiv s.pri[x]$, $shr' \equiv t.shr$,

$pri' \equiv t.pri[i]$, $po' \equiv t.pri[j]$, $pA' \equiv t.pri[x]$:

$$
\begin{aligned}
& exl(s.pri[x]) \wedge \\
& exl(t.pri[x]) \wedge \\
& B.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Guard(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Assign(s.shr, s.pri[i], s.pri[j], s.pri[x], \\
& \qquad t.shr, t.pri[i], t.pri[j], t.pri[x]) \wedge \\
& Frame(s.pri[i], s.pri[j], s.pri[x], t.pri[i], t.pri[j], t.pri[x] \wedge \\
& B.Post(t.shr, t.pri[pri], t.pri[po], t.pri[pA'])
\end{aligned}
$$

The frame condition is that all variables not in the variable action set will remain the same. The above equation have the implicit assumption that pri, po, and pA all refers to different processes. If the refer to the same process, the frame condition should be different. To cover those cases, TDV calls STP several times, with different assumptions on if pri, po, and pA refers to the same process or not. However, is assumed that the process represented my pri is a different process represented my po, that is, $i \neq j$, thus

$$
\begin{aligned}
& i \neq j \wedge \\
& exl(s.pri[x]) \wedge \\
& exl(t.pri[x]) \wedge \\
& B.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Guard(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Assign(s.shr, s.pri[i], s.pri[j], s.pri[x], \\
& \qquad t.shr, t.pri[i], t.pri[j], t.pri[x]) \wedge \\
& Frame(s.pri[i], s.pri[j], s.pri[x], t.pri[i], t.pri[j], t.pri[x] \wedge \\
& B.Post(t.shr, t.pri[pri], t.pri[po], t.pri[pA'])
\end{aligned}
$$

TDV also verify the above equation for all the blocks in the system, thus it is universally quantified over B

$$
\begin{aligned}
\forall B.[\quad & i \neq j \wedge \\
& exl(s.pri[x]) \wedge \\
& exl(t.pri[x]) \wedge \\
& B.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Guard(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Assign(s.shr, s.pri[i], s.pri[j], s.pri[x], \\
& \qquad t.shr, t.pri[i], t.pri[j], t.pri[x]) \wedge \\
& Frame(s.pri[i], s.pri[j], s.pri[x], t.pri[i], t.pri[j], t.pri[x] \wedge \\
& B.Post(t.shr, t.pri[pri], t.pri[po], t.pri[pA'])
\end{aligned}
$$

$$]$$

since s, t, i, j, and x are all uninterpreted, we can universally quantify over those state representations

$$
\begin{aligned}
\forall s, t. \forall i, j, x. \forall B.[\quad & i \neq j \wedge \\
& exl(s.pri[x]) \wedge \\
& exl(t.pri[x]) \wedge \\
& B.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Guard(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B.Assign(s.shr, s.pri[i], s.pri[j], s.pri[x], \\
& \qquad t.shr, t.pri[i], t.pri[j], t.pri[x]) \wedge \\
& Frame(s.pri[i], s.pri[j], s.pri[x], t.pri[i], t.pri[j], t.pri[x] \wedge \\
& B.Post(t.shr, t.pri[pri], t.pri[po], t.pri[pA'])
\end{aligned}
$$

$$]$$

which is what we need as equation 6.8.

The Link property is verified by STP as

$$\begin{aligned} & exl(pA) \wedge \\ & B1.Post(shr, pri, po, pA) \\ & B2.Pre(shr, pri, po, pA) \end{aligned}$$

which after universally quantify over uninterpreted variables, and making the same instantiations, it becomes

$$\begin{aligned} \forall s.\forall i, j, x \quad & Exl(s.pri[x]) \wedge \\ & B1.Post(s.shr, s.pri[i], s.pri[j], s.pri[x]) \\ & B2.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \end{aligned}$$

TDV verifies this equation for every pair of blocks which have a link, thus it verifies

$$\begin{aligned} \forall B1, B2.\forall s.\forall i, j, x \quad & Link(B1, B2) \\ & Exl(s.pri[x]) \wedge \\ & B1.Post(s.shr, s.pri[i], s.pri[j], s.pri[x]) \\ & B2.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \end{aligned}$$

which is equation 6.9.

The interrupt correctness verification boolean expression as verified by STP is

$$\begin{aligned} & exl(pA) \wedge \\ & exl(pA') \wedge \\ & Frame(pri, po, p2, pA, pri', po', p2', pA') \\ & B1.Pre(shr, pri, po, pA) \wedge \\ & B2.Pre(shr, p2, pri, pA) \wedge \\ & B2.Guard(shr, p2, pri, pA) \wedge \\ & B2.Assign(shr, p2, pri, pA, \end{aligned}$$

$$shr', p2', pri', pA') \wedge$$
$$B1.Pre(shr', pri', po', pA')$$

after universally quantify over uninterpreted variables, making instantiations, noting that TDV assumes $i \neq j$ and $i \neq k$ when it is constructing the frame condition, and having TDV verify the equation for all pairs of blocks, the verified equation is

$$\begin{aligned}
\forall s, t. \forall i, j, k, x. \forall B1, B2. \quad & i \neq j \wedge i \neq k \wedge \\
& excl(s.pri[x]) \wedge \\
& excl(t.pri[x]) \wedge \\
& Frame(s.pri[i], s.pri[j], s.pri[k], s.pri[x] \\
& \qquad t.pri[i], t.pri[j], t.pri[k], t.pri[x]) \\
& B1.Pre(s.shr, s.pri[i], s.pri[j], s.pri[x]) \wedge \\
& B2.Pre(s.shr, s.pri[k], s.pri[i], s.pri[x]) \wedge \\
& B2.Guard(s.shr, s.pri[k], s.pri[i], s.pri[x]) \wedge \\
& B2.Assign(s.shr, s.pri[k], s.pri[i], s.pri[x], \\
& \qquad t.shr, t.pri[k], t.pri[i], t.pri[x]) \wedge \\
& B1.Pre(t.shr, t.pri[i], t.pri[j], t.pri[x])
\end{aligned}$$

which is equation 6.10

Chapter 7

Abstract Model Simulation

7.1 Introduction

Formal verification has proved to be very useful technique for checking the correctness of hardware designs. In the recent years, it has been applied to many real-world projects, from small, isolated blocks to large data path implementations [15]. Yet, due to the state explosion problem that occurs in checking the detailed implementation of large designs, simulation is still the main tool for verification in such projects. However, in such projects, formal techniques are regularly used to verify the high-level specification in order to expose protocol-level errors [30, 10, 29, 4, 6, 32, 14]. The problem is that, most of the time, the implementation does not adequately correspond to the high-level specification because such correspondence is ignored, or the specifications fall obsolete due to other constraints that appear during the design process. Often times these constraints alter the high-level behavior of the system and introduce errors which could have been revealed if the consistency between the implementation and its high-level specification was maintained.

To address this issue, we propose a method to check the consistency between the implementation and its abstract model by linking and co-simulating the abstract model with the implementation itself. The correspondence between the concrete and abstract models is maintained by abstraction functions, which map the states and transitions in the implementation to the states and transitions in the abstract

model. At every simulation step, the concrete state is translated to the appropriate abstract state and transition from previous to current abstract state is checked to be legal according to the abstract model. Since the implementation is constantly checked against abstract model, the model is forced to be continually updated to reflect the current version of the hardware. In addition, this methodology constrains the implementation to follow the high-level guidelines provided by the abstract model to guarantee correctness.

It also provides better observability of errors during simulation, since an error is reported as soon as an illegal abstract state transition is made; Errors can be detected without requiring that they be propagated to the outputs, even if they are masked by other interactions in the system. This also simplifies the task of debugging the system by pinpointing the exact state and/or state transition that caused the error.

Section 2 describes the methodology and elaborates about how abstraction functions can be realized. Section 3 discusses about the application of this methodology to verify the coherence protocol in the Smart Memories multi-processor system and the problems revealed by applying it. Section 4 explains a second case study, verification of the Synchronization protocol, and Section 5 concludes this Chapter.

7.2 Methodology

As mentioned, the proposed methodology attempts to close the gap between the implementation and its high level model, and co-simulates the formally-generated assertions along with the implementation. The methodology consists of the following steps:

1. Construct and formally verify the abstract model of the high-level protocol
2. Construct abstraction functions that translate the concrete states in the implementation into the abstract states
3. Simulate the design with the abstract model linked with it. At each step of simulation, the concrete states and transitions will be translated into abstract states and transitions and will be verified against the proved abstract model

This methodology has a number of advantages: First of all, by using the formally verified assertions, it extends the capability of simulation in finding corner case problems in the higher level protocol. Second, the linking of abstract model with simulation facilitates the use of formal constraints in verification of large designs. Finally, constructing the abstract model in parallel with the implementation and linking the two together can potentially reveal high-level problems early in the design process.

While this approach is similar to correct-by-construction methodology in requiring an abstract model early in the design and using it to restrict the kinds of errors possible in the resulting implementation, it is more easily integrated with today's design methodology, which is driven by the implementation issues and not the high-level design properties. By allowing the designers flexibility in constructing the implementation, even to the point where the abstract model must change, the methodology acknowledges that, in real designs, implementation issues often constrain the abstract machines. Yet, it maintains the benefits of improved checking that comes with generating and formally validating the abstract model.

7.2.1 Formalization

Abstraction functions have been widely used in formal verification since 1980s [12, 35, 16]. The following formalization is a standard application of it; our contribution is applying this technique in simulation. The proposed methodology expresses the properties of interest in the abstract domain, which implies that the abstraction function is part of the specification. The methodology can be formalized as follows: implementation is represented with the initial state s_0, and next state function $s_{i+1} = \chi(s_i, I_i)$, where I_i is the input at cycle i. The abstract model is represented by the Kripke structure $M_a = (S, R, S_0, L)$, where S is the set of abstract states, R is the set of valid transition, S_0 is the set of initial states. Suppose that ϕ is a property that we are interested in verifying, then to say that the abstract model has property ϕ, we say that for all paths $\pi_a = t_0, t_1, t_2, \ldots$, where $t_0 \in S_0$ and $\forall i \geq 0. R(t_i, t_{i+1})$, that

$$M_a, \pi_a \models \phi \tag{7.1}$$

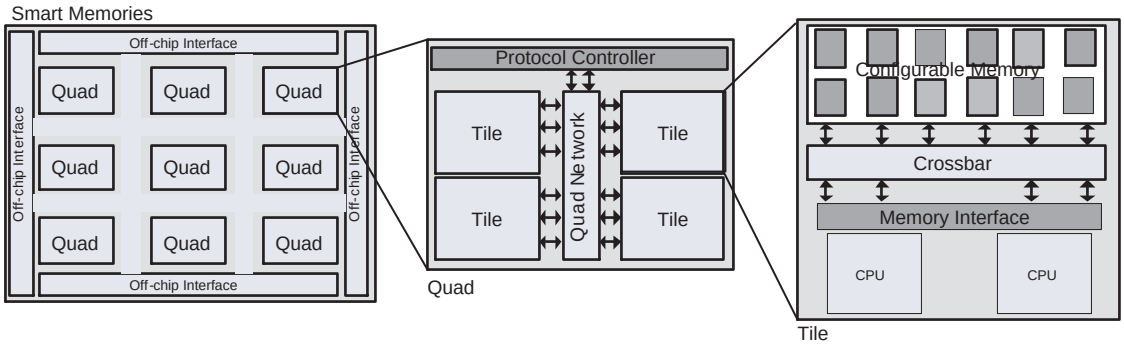


Figure 7.1: Smart Memory Architecture

Let α be the abstraction function such that $t = \alpha(s)$, where $t \in S$, and s is a concrete state. The verification we are interested on the real design is that for all paths $\pi_{\alpha(c)} = \alpha(s_0), \alpha(s_1), \alpha(s_2), \ldots$

$$M_a, \pi_{\alpha(c)} \models \phi \tag{7.2}$$

Conceptually in the process of verification, a safeguard placed on the real design can be broken up into two components: an abstract invariant that needs to be held, and how the concrete state is related to that invariant. The abstract invariant corresponds to ϕ, and the interpretation of the concrete states is α.

Verification of Eq (7.2) can be accomplished by checking that

1. Equation (7.1) holds.

2. $\alpha(s_0) \in S$

3. $\forall I. \forall s$ reachable in concrete model $R(\alpha(s), \alpha(\chi(s, I)))$

Conditions 2 and 3 state that $\pi_{\alpha(c)}$ is a valid path of M_a.

We can prove condition 1, which are invariants of the abstract model, through formal techniques. However, it is difficult to prove 3 formally because it would involve applying formal methods to a large design (which is not feasible). Instead, we show that 2 and 3 are correct for all the test vectors used in simulation.

Req Valid	Idx Lock	CWB Lock	Line Address	Op-Code (5)	Type (4)	Action (6)	State	Mrg	$1 Reply			$2 Reply			$3 Reply			$4 Reply		
									V	E	M	V	E	M	V	E	M	V	E	M

Figure 7.2: Concrete State Information

7.2.2 Abstraction Function Construction

In order to make the methodology applicable to real design projects, the abstraction functions should be kept simple enough so that they are not affected by the design modifications as the design evolves during the process. An important step to facilitate this purpose is to make the abstract model as close to the implementation as possible by employing various formal techniques such as refinement, theorem proving, model checking, etc.

On the other hand, the design team can make the implementation better correspond to its abstract model, given that such abstract model is constructed in the early stages of the design process. If the information required by the abstraction functions is exposed to the designers, they can attempt to make this information accessible in the implementation itself, thus helping the development of abstraction functions. Together these techniques can tremendously simplify the task of developing abstraction functions, as explained in the case studies.

This methodology was applied to the verification of Smart Memories system, which is a scalable, modular and reconfigurable shared memory multi-processor [19]. As figure 7.1 shows, the system consists of Tiles each having two processing cores, a number of reconfigurable memory blocks and a crossbar. The Tiles are grouped in Quads, where each Quad consists of four Tiles and a shared protocol controller/interface. Quads communicate over a mesh-like network and interface to external memories through a number of memory controllers. Since this is a coherent shared-memory environment, the methodology is applied first to verify the cache coherence protocol, and then to the fine-grain synchronization protocol.

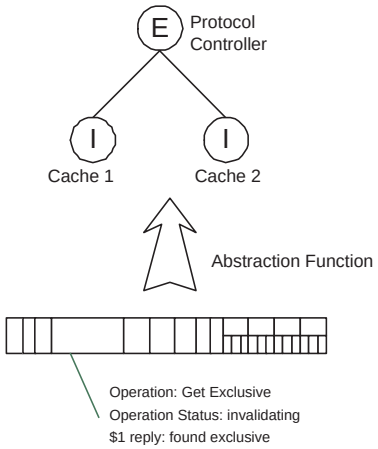


Figure 7.3: Example of Abstraction

7.3 Cache Coherence Protocol

The coherence protocol implemented in Smart Memories system is MESI, where there can be multiple readers of a cache line, but there can be only a single writer at a time. MESI protocol and its variants have been verified several times for bus-based and directory-based systems [34, 9]. In Smart Memories, the coherence protocol is implemented in a hierarchical manner: the Protocol Controllers keep the caches inside the Quad coherent with respect to each other. They lookup the caches upon each cache miss received from a processor and enforce the coherence invariant. They also do cache-to-cache transfers between caches inside the Quad if possible. Memory Controllers on the other hand, enforce coherence among the Quads: they receive cache miss requests from Quads and send coherence requests to other Quads to have them change the cache line states according to the protocol, or return the cache line in case it is modified.

7.3.1 Applying the Methodology

As the first step, an abstract model of the system and the coherence protocol is created. For this purpose, the memory system is considered as a hierarchy of memory elements, where a memory element can represent a cache, a protocol controller, a

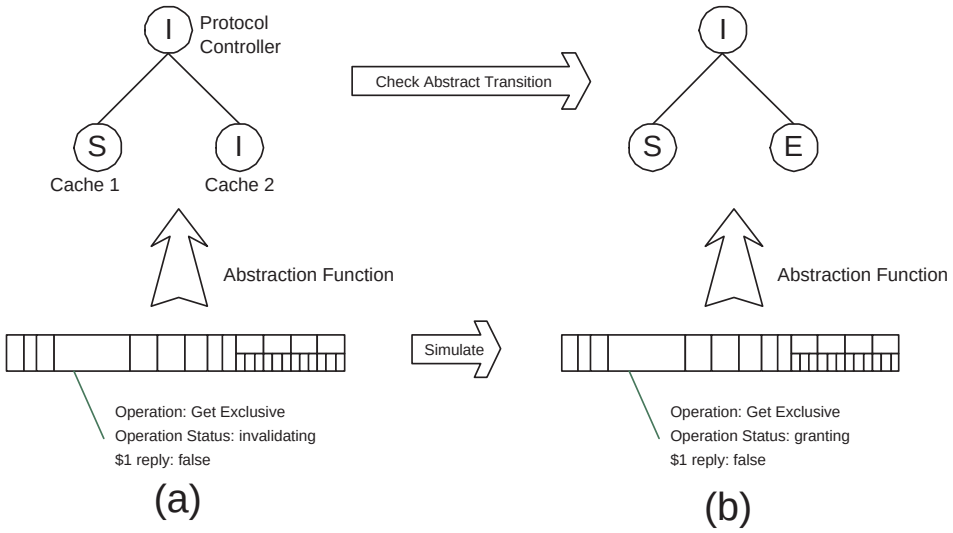


Figure 7.4: Abstract Model Checker

memory controller or a memory bank in the real system. In the abstract model, each memory element can contain a cache line and will associate one of the MESI states with it, in addition to some information about the state of that cache line in the other memory elements.

Given this model, rules governing the state transitions in each memory element preserve the coherence invariants. For example, a protocol controller is allowed to give a cache line to a cache in Exclusive state only if the protocol controller ensures that all other memory elements have that cache line in Invalid state. Since the abstract model consists mainly of states and transition rules, it is essentially a finite state machine. Verifying the fact that this state machine enforces the coherence between all the caches in the system was performed using various formal verification tools and techniques. The state machine was described and formally verified using Cadence SMV model checker [22]. Higher level properties were proved using PVS theorem prover [26]. A Perl script was used to translate these abstract state machine descriptions into an executable code in the Smart Memories functional simulator.

The next step is to extract the abstract states from the concrete states within the functional simulator, using abstraction functions. For each memory element in the system, a function is developed which returns the state of a given cache line in that

memory element. As an example of the abstraction function, we show how such a function can be developed for the protocol controller. Figure 7.2 shows the (concrete) state information which the protocol controller keeps for each cache miss. Note that there is no specific MESI state associated with the missing cache line. Instead, the protocol controller keeps information concerning the type of cache miss, the lookup results from Quad caches, and replies from memory controller. This information is enough for the abstraction function to determine the abstract state of that cache line within the protocol controller, as described by the following example.

Consider a write miss for cache line X received by the protocol controller from one of the caches. The protocol controller will send lookup/invalidate requests to the other caches in the Quad. Each cache will perform a lookup and invalidate its local copy, and then it will send the previous state of the line (state before invalidation) back to the protocol controller. Now if a cache sends back an Exclusive state, based on the replies from all the caches and the type of the cache miss, the abstraction function can deduce that line X is now transferred from that cache into the protocol controller and it is in the Exclusive state. Figure 7.3 illustrates the information decoded by the abstraction function to produce the equivalent abstract state. The abstraction function also checks to ensure that all other caches return Invalid state, since otherwise the concrete state is not consistent and therefore cannot be translated into the abstract state.

To see how this might be useful, suppose that there is an error in the implementation, where sometimes protocol controller refills a cache in the Exclusive state without invalidating all other copies of the cache line. This problem is very difficult to catch in the simulation since revealing it depends on many factors such as the test case being simulated, contents of cache line, the sequence of accesses to cache line from different processors, and specific timing of the events. However, such a problem will be revealed immediately using our methodology, since this cache refill in the Exclusive state corresponds to an illegal state transition in the high-level abstract model.

Figure 7.4 shows the details of this detection process. In the first cycle, cycle (a), we see that the protocol controller have not yet invalidated cache 1. The abstraction function will specify that protocol controller is in the Invalid (I) state for this cache

line. In the next cycle, cycle (b), the protocol controller erroneously refills the cache line in cache 2. The abstraction function will say that cache 2 is now in the Exclusive (E) state. However in the abstract model the transition of cache 2 from I to E is not allowed if the protocol controller is in the I state, thus the error is detected by simulating the abstract model.

In order to perform the comprehensive consistency check between concrete and abstract states, the abstract state is advanced whenever there is a change in the concrete state of the functional simulator. Note that not all the state changes in the functional simulator correspond to a state change in the abstract model since they might be irrelevant to the high-level model, and hence to the property of interest.

7.3.2 Results

Applying the methodology to the implementation of cache coherence protocol in the Smart Memories functional simulator discovered various design and model errors, ranging from coding bugs in the simulator itself to incorrect state transitions for the cache lines in the system.

A very interesting problem was that the system requires distinguishing between two different types of write back: eviction write backs and coherence write backs. Write backs might be initiated by either evicting a modified cache line from a cache (eviction write back), or by degrading a modified cache line to shared state (coherence write back). Eviction write backs implicitly express the fact that none of the caches in the system contain the line, but this is not true for the coherence write back, since copies of the line still exist in some caches in the Shared state. The necessity for such distinction was realized when the designers and verification experts were trying to construct the abstract model of the implementation and correspond the cache line state transitions in the functional simulator to its abstracted model. Since this correspondence was made at the early design stages, it helped the designers to correct the problem and implement two different types of write back requests in their system.

Another problem (a potential pitfall) discovered when the system is trying to do an optimization via bypassing the data carried by a write back request back to a

Table 7.1: Semantics of the synchronization operations

Instruction	F/E==1	
	W==1	W==0
Sync Load	F/E=0,W=0 Read data, Send Wakeup	F/E=0 Read data
Sync Store	Stall Send Sync Miss	W=1 Stall, Send Sync Miss
Future Load	Read data	Read data
Reset Load	F/E=0, W=0 Read data, Send Wake up	F/E=0 Read data
Set Store	Write data	Write data
Instruction	F/E==0	
	W==1	W==0
Sync Load	Stall Send Sync Miss	W=1 Stall, Send Sync Miss
Sync Store	F/E=1, W=0 Write data, Send Wakeup	F/E=1 Write data
Future Load	Stall Send Sync Miss	W=1 Stall, Send Sync Miss
Reset Load	Read data	Read data
Set Store	F/E=1, W=0 Write data, Send Wake up	F/E=1 Write data

cache miss request. As a result, multiple caches could potentially obtain an exclusive version of the cache line, which obviously is incorrect. Both of these examples are corner case problems that require a specific sequence of events to occur in order for traditional simulation to detect them.

Last but not least, the methodology was able to reveal some coding problems in the functional simulator itself. Examples are incorrect identification of cache line states, incorrect initializations and incorrect state transitions within controller's finite state machine. The abstract model was able to capture such errors by detecting the violations of invariants when running concurrently with the functional simulator.

7.4 Synchronization Protocol

Memory words in the Smart Memories system are augmented with extra meta-data bits that can be used for different purposes. An interesting application of these bits is in implementation of fine-grain locks. For each word, a single meta-data bit is considered as "Full/Empty" (F/E) indicator: when F/E bit is one, the location is considered as full and processor is allowed to consume the data contents of the word. When the bit is zero, the word is considered as empty and processor is stalled until another value is written into that address. This situation is usually referred to as a ´Synchronization Miss¡. In such a case, processor sends a synchronization miss request to a memory controller before stalling, to enlist itself as a waiter on that specific address. The processor is awakened later on to retry its access, when another processor writes the data.

In addition to the Full/Empty bit, a second bit is also used to indicate the fact that a synchronization miss has already occurred for the word and thus there is at least one waiting processor on this address. This second bit is called the "Waiting" (W) bit. Upon a successful access to a word, if the W bit is set, a wake up request is generated and sent to the memory controller. The memory controller then looks up all the waiting processors stalled on the specified address, and wakes up the first one in order to retry its access. The entire interaction between the processors, memory blocks and memory controller which causes stalling and un-stalling the processors

upon synchronization operations is referred to as the *"Synchronization Protocol."*

The instruction set of the processors is extended by adding special memory access instructions that can leverage these extra bits and the manipulation mechanism. Table 7.1 describes the semantics of these instructions and the effect they have on the F/E and W bits. Columns two to four describe the result of these memory access operations and read-modify-write action on the bits, based on the initial value of F/E and W bits.

Obviously, the important property about this protocol is the fact that it is deadlock free. The protocol has to guarantee that all the waiting processors will eventually wake up and successfully retry their accesses so that the application can make forward progress, given that synchronization accesses are used correctly by the application itself.

7.4.1 Results

Once again, applying the proposed methodology discovered some faults, which otherwise would be difficult to catch. An important problem that was revealed was related to the ordering of communicated messages between processors and memory controllers. Since two processors within the Tile share memory blocks, there might be occasions when a synchronization miss and a wake up request for the same memory address are generated within a few cycles. For instance, the first processor might issue a Synchronized Load that causes a synchronization miss, and in the next cycle, second processor might issue a Synchronized Store, which generates a wake up. In such cases, if the memory controller receives the wake up request before the synchronization miss request, the wake up might be dropped leading to a dangling synchronization access which might never un-stalled.

During the process, a potential pitfall was also discovered about the way synchronization accesses are used by the programmer. In the applications and run-time system, there were places where non-blocking Set Store and Reset Load accesses were used instead of the blocking Synchronized Store and Synchronized Load. Although the program could potentially work properly in certain cases, there was no clear way

to guarantee the proper behavior in all the cases.

7.5 Discussion

The proposed methodology proves to be useful for control-oriented design projects where the implementation is hard to be formally verified due to state explosion. Such designs also have the largest number of the corner cases and timing/sequencing related problems that are difficult to identify using traditional verification efforts. Simulations detect errors in the implementation by comparing the expected output with the actual results. In contrast, in the proposed approach, formal model identifies errors by assuring validity of each state transition independent of factors such as time, sequence or values involved in each transition. However, this fact does not make this approach completely independent of the test vectors, since if none of the test vectors excites a specific abstract state or transition, they will not be verified.

Although this methodology is not entirely formal, it is more successful in detecting corner case errors because first, the test vector does not need to excite an observable error in the implementation. Second, the erroneous transition is detected immediately upon occurrence by pinpointing the exact transition in the implementation that violated the protocol invariants. This is in contrast to traditional verification method, where the errors need to be propagated to an observable point in the system in order to be detected.

The difficulty of applying this methodology lays mainly in the creation of abstraction functions. If the concrete state encoded in the implementation is not properly organized, developing and maintaining the abstraction functions as design changes becomes a demanding task. Note that the errors in the abstraction function itself can also be detected, since incorrect implementation of an abstraction function will cause an inconsistency between the concrete and abstract states, hence reporting an invalid abstract state transition. To keep abstraction functions simple and easy to develop, it is important to construct the abstract model in the early design stages, and communicate to designers the information required by each abstraction function.

Designers can contribute significantly to the realization of these functions, by making small changes to the design that simplify access to the information inside the implementation.

Another major factor which affects the development of the abstraction functions is the abstract model itself. The abstract model might need to be modified a number of times in order to not only cover the high-level properties of the protocol, but also properly represent the implementation of it. Consistency between the abstract model and implementation needs to be established and maintained as the implementation undergoes regular design modifications. Although not always trivial, constructing and maintaining the abstract model in parallel with the design helps in early detection of the system-level protocol problems, which are usually much more difficult to fix in the later stages of the design process.

Chapter 8

Conclusion

8.1 Perspectives

Using the concept of perspectives, it is demonstrated that verification early in the design is possible and useful to catch design problems early in the design process. The formal verification concepts of abstraction and refinement is well suited to model designs in the early stage. However, the preciseness and the non-ambigouity of the formal model descriptions make formal verification difficult in the early design stages because of the rapid changes that the design goes through during the early design process.

By developing a lightweight formal verification process, it is now possible to formally verify the design concepts of the abstract design quickly, and thus adaptable to design changes. However, using the perspective based verification, the model is not completely verified, and design flaws can be missed for the portions that was not explicitly modeled. However, there is no getting around this because of the incomplete nature of the design in the early design stages. In addition, being able to proving a formal model allow the verification engineer and the designers to pinpoint important aspects about the design that must be clarified to be able to complete the design correctly.

Using the concept of perspectives, various design flaws in different parts of the smart memory memory system was found early in the design process. The types of

design that it was able to handle varies from deadlock issues, cache coherence protocol, cache controller architecture design, and the system design for synchronization operation. In verifying these different aspects of the design, several perspectives are identified.

One of the perspective identified–The Parallel Transition Perspective–was further developed and the tool Transaction Diagram Verifier was written for verifying models written under this perspective. By developing this tool, it was demonstrated that many of the work involved in proving the correctness of the design for problems in Parallel Transactions can be encapsulated in the tool, thus the user of the tool is allowed to maintain small models and able to complete the verification quickly to allow the verification to keep up with the design process. It is also demonstrated that identifying problems into the correct perspective is important, as each perspective is only efficient solving problems in its domain.

8.2 Model Correspondence

One method to overcome the problem of incomplete verification with perspective based verification is to build the model into the simulator so that the abstract model can be co-simulated with the detailed design. This ensures that the design is implementing the verified abstract model. Although we have used this method in the smart memory cache protocol verification, the work required to do this verification is time consuming. Thus this methodology does not work well early in the design process. However, it does provide one solution to verify the detailed design against the perspective based abstract models.

8.3 Future Work

The Transaction Diagram Verifier was developed for the Parallel Transaction Perspectives. Although perspective such as the Transaction with error states can be solved with the widely used Model Checkers, the other perspectives identified in this Thesis does not have a tool geared toward solving problems in those perspectives.

Having tools available for the different perspectives is needed for a perspective based verification methodology.

Although the perspective based verification have been done on the smart memory project, one open question would be how much of the problem in existence in a typical designs today can be solved using the few perspectives that have been identified. It is hoped that a handful of perspectives can be used to uncover most of the design flaws that are hard for traditional verification to catch. However, whether this is the case or now remains unknown until the perspective concept can be applied to more real world designs.

Appendix A

TDV Correctness Proof in PVS

A.1 PVS File

```
TDV : THEORY
BEGIN
  PID : TYPE;
  BID : TYPE;

  PRI_STATE_OTHER : TYPE;
  PRI_STATE : TYPE = [# pc : BID,
                        other : PRI_STATE_OTHER #];
  SHR_STATE : TYPE;

  BLOCK : TYPE = [#
            Pre : [SHR_STATE, PRI_STATE, PRI_STATE, PRI_STATE -> bool],
            Guard : [SHR_STATE, PRI_STATE, PRI_STATE, PRI_STATE -> bool],
            Assign : [SHR_STATE, PRI_STATE, PRI_STATE, PRI_STATE,
                      SHR_STATE, PRI_STATE, PRI_STATE, PRI_STATE -> bool],
            Post : [SHR_STATE, PRI_STATE, PRI_STATE, PRI_STATE -> bool]
          #]
```

```
STATE : TYPE = [# shr : SHR_STATE,
                  pri : [PID -> PRI_STATE] #]

EXL : pred[PRI_STATE];
Link : [BID,BID -> bool];

default_block : [BID -> BLOCK];
blocks : [BID -> BLOCK] =
  LAMBDA(b:BID) : (#
    Pre := LAMBDA(shr:SHR_STATE,pri:PRI_STATE,po:PRI_STATE,pA:PRI_STATE) : (
             default_block(b)`Pre(shr,pri,po,pA) AND
             (EXL(pri) IMPLIES NOT EXL(po))
           ),
    Guard := LAMBDA(shr:SHR_STATE,pri:PRI_STATE,po:PRI_STATE,pA:PRI_STATE):(
               default_block(b)`Guard(shr,pri,po,pA)
             ),
    Assign := LAMBDA(shr:SHR_STATE,pri:PRI_STATE,po:PRI_STATE,pA:PRI_STATE,
                     shrX:SHR_STATE,priX:PRI_STATE,poX:PRI_STATE,
                     pAX:PRI_STATE) : (
                default_block(b)`Assign(shr,pri,po,pA,shrX,priX,poX,pAX) AND
                pri`pc=b AND (EXISTS(b2:BID): Link(b,b2) AND priX`pc=b2)
              ),
    Post := LAMBDA(shr:SHR_STATE,pri:PRI_STATE,po:PRI_STATE,pA:PRI_STATE) : (
              default_block(b)`Post(shr,pri,po,pA) AND
             (EXL(pri) IMPLIES NOT EXL(po))
           )
  #)

Frame_Condition : [STATE,STATE,PID->[PID,PID->bool]] =
  LAMBDA(s:STATE,t:STATE,pri:PID): LAMBDA(po:PID,pA:PID): (
    (pri/=po IMPLIES s`pri(po)`pc=t`pri(po)`pc) AND
```

```
    (pri/=pA IMPLIES s‘pri(pA)‘pc=t‘pri(pA)‘pc)
  );

  Exclusive_Exist : AXIOM
    FORALL(s:STATE): EXISTS(pA:PID): EXL(s‘pri(pA))

  INV : [STATE ->bool] = LAMBDA(s:STATE) : (
    (FORALL(i:PID, j:PID) : i/=j AND EXL(s‘pri(i)) IMPLIES NOT EXL(s‘pri(j)))
    AND
    (FORALL(i:PID,j:PID,x:PID) :
      i/=j AND EXL(s‘pri(x)) IMPLIES (
        FORALL(b:BID):
          s‘pri(i)‘pc=b IMPLIES blocks(b)‘Pre(s‘shr,s‘pri(i),s‘pri(j),s‘pri(x))
      )
    )
  );

  TRANS : [STATE,STATE -> bool] = LAMBDA(s:STATE,t:STATE) : (
    EXISTS(i:PID):
      FORALL(j:PID,x:PID):
        i/=j IMPLIES FORALL(b:BID): (s‘pri(i)‘pc=b IMPLIES
          blocks(b)‘Guard(s‘shr,s‘pri(i),s‘pri(j),s‘pri(x))
          AND blocks(b)‘Assign(s‘shr,s‘pri(i),s‘pri(j),s‘pri(x),
                               t‘shr,t‘pri(i),t‘pri(j),t‘pri(x))
          AND Frame_Condition(s,t,i)(j,x)
        )
  );

  verif1 : AXIOM
    FORALL(s:STATE,t:STATE) :
    FORALL(pri:PID, po:PID, p2:PID,pA:PID,pAX:PID):
```

```
      (
         pri/=po AND pri/=p2
      )
      IMPLIES
      FORALL(b1:BID, b2:BID) : (
        (
         EXL(s`pri(pA)) AND
         EXL(t`pri(pAX)) AND
         blocks(b1)`Pre(s`shr,s`pri(pri),s`pri(po),s`pri(pA)) AND
         blocks(b2)`Pre(s`shr,s`pri(p2),s`pri(pri),s`pri(pA)) AND
         blocks(b2)`Guard(s`shr,s`pri(p2),s`pri(pri),s`pri(pA)) AND
         blocks(b2)`Assign(s`shr,s`pri(p2),s`pri(pri),s`pri(pA),
                           t`shr,t`pri(p2),t`pri(pri),t`pri(pA))
         AND Frame_Condition(s,t,p2)(pri,pA)
        ) IMPLIES (
          blocks(b1)`Pre(t`shr,t`pri(pri),t`pri(po),t`pri(pAX))
        )
      )

  block_prop : AXIOM
    FORALL(s:STATE,t:STATE) :
    FORALL(pri:PID, po:PID,pA:PID,pAX:PID):
      FORALL(b1:BID) : (
        (
         pri/=po AND
         EXL(s`pri(pA)) AND
         EXL(t`pri(pAX)) AND
         blocks(b1)`Pre(s`shr,s`pri(pri),s`pri(po),s`pri(pA)) AND
         blocks(b1)`Guard(s`shr,s`pri(pri),s`pri(po),s`pri(pA)) AND
         blocks(b1)`Assign(s`shr,s`pri(pri),s`pri(po),s`pri(pA),
                           t`shr,t`pri(pri),t`pri(po),t`pri(pA))
```

```
        ) IMPLIES (
          blocks(b1)`Post(t`shr,t`pri(pri),t`pri(po),t`pri(pAX))
        )
      )

  link_prop : AXIOM
    FORALL(s:STATE) :
    FORALL(pri:PID, po:PID,pA:PID):
      FORALL(b1:BID, b2:BID) : (
% don't need EXL(pA) line, (not actually used) but it's a safe
% expansion for future if needed
          EXL(s`pri(pA)) AND
          Link(b1,b2) AND
          blocks(b1)`Post(s`shr,s`pri(pri),s`pri(po),s`pri(pA))
        IMPLIES
          blocks(b2)`Pre(s`shr,s`pri(pri),s`pri(po),s`pri(pA))
    );

  trans : THEOREM
    FORALL(s:STATE, t:STATE) :
      (INV(s) AND TRANS(s,t)) IMPLIES INV(t)

END TDV
```

A.2 PVS Proof file

```
(TDV
 (trans 0
  (trans-1 nil 3416231532 3416232345
   ("" (expand* "INV" "TRANS")
    (("" (skosimp*)
```

```
(("" (split)
  (("1" (use "Exclusive_Exist" ("s" "s!1"))
    (("1" (use "Exclusive_Exist" ("s" "t!1"))
      (("1" (skosimp*)
        (("1" (case "i!1=j!1")
          (("1" (inst -8 "i!2" "pA!2")
            (("1" (assert)
              (("1" (inst - "s!1`pri(i!1)`pc")
                (("1" (lemma "block_prop")
                  (("1"
                    (inst - "s!1" "t!1" "j!1" "i!2" "pA!2" "pA!1"
                     "s!1`pri(j!1)`pc")
                    (("1" (split)
                      (("1" (grind) nil nil)
                       ("2" (grind) nil nil)
                       ("3" (propax) nil nil)
                       ("4" (propax) nil nil)
                       ("5" (inst - "j!1" "i!2" "pA!2")
                        (("5" (grind) nil nil)) nil)
                       ("6" (grind) nil nil)
                       ("7" (grind) nil nil))
                      nil))
                    nil))
                  nil))
                nil))
              nil))
            nil)
           ("2" (inst -7 "j!1" "pA!2")
            (("2" (assert)
              (("2" (inst - "s!1`pri(i!1)`pc")
                (("2" (lemma "verif1")
```

```
(("2"
  (inst - "s!1" "t!1" "j!1" "i!2" "i!1" "pA!2"
   "pA!1")
  (("2" (split)
    (("1"
      (inst - "s!1`pri(j!1)`pc"
       "s!1`pri(i!1)`pc")
      (("1"
        (split)
        (("1" (grind) nil nil)
         ("2" (propax) nil nil)
         ("3" (propax) nil nil)
         ("4"
          (inst - "j!1" "i!2" "pA!2")
          (("4"
            (split)
            (("1" (inst?) nil nil)
             ("2" (grind) nil nil)
             ("3" (propax) nil nil))
            nil))
          nil)
         ("5"
          (inst - "i!1" "j!1" "pA!2")
          (("5"
            (split)
            (("1" (inst?) nil nil)
             ("2" (grind) nil nil)
             ("3" (propax) nil nil))
            nil))
          nil)
         ("6" (grind) nil nil)
```

```
                          ("7" (grind) nil nil)
                          ("8" (grind) nil nil))
                         nil))
                       nil)
                      ("2" (grind) nil nil)
                      ("3" (grind) nil nil))
                     nil))
                   nil))
                 nil))
               nil))
             nil))
           nil))
         nil))
       nil))
     nil)
    ("2" (use "Exclusive_Exist" ("s" "s!1"))
     (("2" (skosimp*)
       (("2" (case "i!1=i!2")
         (("1" (inst -7 "j!1" "pA!1")
           (("1" (assert)
             (("1" (inst - "s!1`pri(i!1)`pc")
               (("1" (lemma "block_prop")
                 (("1"
                   (inst - "s!1" "t!1" "i!2" "j!1" "pA!1" "x!1"
                    "s!1`pri(i!1)`pc")
                   (("1" (split)
                     (("1" (lemma "link_prop")
                       (("1"
                         (inst - "t!1" "i!2" "j!1" "x!1"
                          "s!1`pri(i!2)`pc" "t!1`pri(i!2)`pc")
```

```
                  (("1" (grind) nil nil)) nil))
                nil)
               ("2" (grind) nil nil) ("3" (propax) nil nil)
               ("4" (propax) nil nil)
               ("5" (inst - "i!2" "j!1" "pA!1")
                (("5" (split)
                  (("1"
                    (inst?)
                    (("1" (grind) nil nil))
                    nil)
                   ("2" (grind) nil nil)
                   ("3" (propax) nil nil))
                  nil))
                nil)
               ("6" (grind) nil nil) ("7" (grind) nil nil))
              nil))
            nil))
          nil))
        nil))
      nil))
    nil)
   ("2" (inst -6 "i!2" "pA!1")
    (("2" (assert)
      (("2" (inst - "s!1`pri(i!1)`pc")
        (("2" (lemma "verif1")
          (("2"
            (inst - "s!1" "t!1" "i!2" "j!1" "i!1" "pA!1"
             "x!1")
            (("2" (split)
              (("1"
                (inst - "s!1`pri(i!2)`pc"
```

```
                    "s!1`pri(i!1)`pc")
                   (("1" (split)
                     (("1" (grind) nil nil)
                      ("2" (propax) nil nil)
                      ("3" (propax) nil nil)
                      ("4"
                       (inst - "i!2" "j!1" "pA!1")
                       (("4" (grind) nil nil))
                       nil)
                      ("5"
                       (inst - "i!1" "i!2" "pA!1")
                       (("5"
                         (split)
                         (("1" (inst?) nil nil)
                          ("2" (grind) nil nil)
                          ("3" (propax) nil nil))
                         nil))
                        nil)
                      ("6" (grind) nil nil)
                      ("7" (grind) nil nil)
                      ("8" (grind) nil nil))
                     nil))
                   nil)
                  ("2" (grind) nil nil) ("3" (grind) nil nil))
                 nil))
               nil))
             nil))
           nil))
         nil))
       nil))
     nil))
```

```
          nil))
        nil))
      nil))
    nil))
 nil)
proved
((TRANS const-decl "[STATE, STATE -> bool]" TDV nil)
 (INV const-decl "[STATE -> bool]" TDV nil)
 (boolean nonempty-type-decl nil booleans nil)
 (= const-decl "[T, T -> boolean]" equalities nil)
 (block_prop formula-decl nil TDV nil)
 (blocks const-decl "[BID -> BLOCK]" TDV nil)
 (/= const-decl "boolean" notequal nil)
 (Frame_Condition const-decl
  "[STATE, STATE, PID -> [PID, PID -> bool]]" TDV nil)
 (BID type-decl nil TDV nil) (verif1 formula-decl nil TDV nil)
 (STATE type-eq-decl nil TDV nil) (SHR_STATE type-decl nil TDV nil)
 (PRI_STATE type-eq-decl nil TDV nil) (PID type-decl nil TDV nil)
 (Exclusive_Exist formula-decl nil TDV nil)
 (link_prop formula-decl nil TDV nil))
134769 1960 t nil)))
```

Bibliography

[1] Martín Abadi, Leslie Lamport, and Stephan Merz. A tla solution to the rpc-memory specification problem. In *Formal Systems Specification, The RPC-Memory Specification Case Study (the book grow out of a Dagstuhl Seminar, September 1994)*, pages 21–66, London, UK, 1996. Springer-Verlag.

[2] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification*, chapter 11.8, pages 344–365. Morgan Kaufmann, 2007.

[3] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification*, chapter 3.6.1.2, pages 47–48. Morgan Kaufmann, 2007.

[4] Giampaolo Bella, Lawrence C. Paulson, and Fabio Massacci. The verification of an industrial payment protocol: the set purchase phase. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 12–20, New York, NY, USA, 2002. ACM Press.

[5] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. *fmcad*, 0:81–88, 2006.

[6] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol, 2003.

[7] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Zenore D. Zuck. Verifying correctness of transactional memories. *Formal Methods in Computer Aided Design*, pages 37–44, 11-14 Nov. 2007.

[8] Christian Creveuil and Gruia-Catalin Roman. Formal specification and design of a message router. *ACM Trans. Softw. Eng. Methodol.*, 3(4):271–307, 1994.

[9] Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Form. Methods Syst. Des.*, 23(3):257–301, 2003.

[10] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.

[11] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[12] Abraham Ginzburg. *Algebraic Theory of Automata*. Academic Press, New York, 1968.

[13] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *isca*, 00:102, 2004.

[14] Roope Kaivola. Formal verification of pentium 4 components with symbolic simulation and inductive invariants. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2005.

[15] Roope Kaivola and Katherine R. Kohatsu. Proof engineering in the large: Formal verification of pentium 4 floating-point divider. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 196–211, London, UK, 2001. Springer-Verlag.

[16] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Trans. Software Eng.*, 10(4):325–342, 1984.

[17] Yogesh Mahajan, Carven Chan, Ali Bayazit, Sharad Malik, and Wei Qin. Verification driven formal architecture and microarchitecture modeling. *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on*, pages 123–132, May 30 2007-June 2 2007.

[18] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 161–171, 2000.

[19] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 161–171, 2000.

[20] Anmol Mathur and Venkat Krishnaswamy. Design for verification in system-level models and rtl. In *44th Design Automation Conference*, pages 193–198. ACM, June 2007.

[21] Ken L. McMillan. Fitting formal methods into the design cycle. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 314–319, New York, NY, USA, 1994. ACM Press.

[22] Ken L McMillan. *Getting Started with SMV*. Cadence Berkeley Labs, Berkeley, CA, 1999.

[23] Kenneth L. McMillan. A compositional rule for hardware design refinement. In *CAV*, pages 24–35, 1997.

[24] Marcio T. Oliveira and Alan J. Hu. High-level speci .cation and automatic generation of ip interface monitors. *dac*, 00:129, 2002.

[25] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[26] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[27] Discussion Panel. The 50-million transistor chip: The quality challenge for 2001. In *International Symposium on Quality Electronic Design*, March 2001.

[28] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 284–290, New York, NY, USA, 1998. ACM Press.

[29] H. Peng, S. Tahar, and F. Khendek. Comparison of spin and vis for protocol verification, 1999.

[30] Fong Pong and Michel Dubois. The verification of cache coherence protocols. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 11–20, 1993.

[31] Winston Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, pages 1–9, August 1970.

[32] Abhik Roychoudhury, Tulika Mitra, and S. R. Karri. Using formal techniques to debug the amba system-on-chip bus protocol. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10828, Washington, DC, USA, 2003. IEEE Computer Society.

[33] Rindert Schutten and Tom Fitzpatrick. Design for verification, April 2003.

[34] X. Shen. A methodology for designing correct cache coherence protocols for dsm systems, 1997.

[35] Joseph Sifakis. Property preserving homomorphisms of transition systems. In Edmund M. Clarke and Dexter Kozen, editors, *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 1983.

[36] Syed M. Suhaib, Deepak A. Mathaikutty, David Berner, and Sandeep K. Shukla. Extreme formal modeling for hardware models. In *5th International Workshop on Microprocessor Test and Verification (MTV'04)*, September 2004.

[37] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. Xfm: An incremental methodology for developing formal models. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):589–609, 2005.

[38] Serdar Tasiran, Yuan Yu, and Brannon Batson. Linking simulation with formal verification at a higher level. *IEEE Design Test of Computers*, 21(6):472–482, Nov.-Dec. 2004.

[39] Moshe Y. Vardi. Formal techniques for systemc verification. In *44th Design Automation Conference*, pages 188–192. ACM, June 2007.

[40] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 26(1):70–74, 1983.

[41] Sharad Malik Yogesh Mahajan. Automating hazard checking in transaction-level microarchitecture models. In *FMCAD07*, November 2007.