WRITING, VERIFYING, AND EXPLOITING FORMAL SPECIFICATIONS FOR HARDWARE DESIGNS

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

Kanna Shimizu

October 2002

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.

Dawson R. Engler

Approved for the University Committee on Graduate Studies:

Abstract

It is widely recognized that writing solid specifications for design interfaces is critical in this day of complex, multi-module hardware designs. However, many still resort to natural languages such as English to document interface protocols such as those of busses and IP (intellectual property) cores. Consequently, these protocols are often buggy and are frequently mis-communicated due to the inherent ambiguity of natural language specifications. Formal specifications, with precisely defined symbols and semantics, will resolve many of these problems, but in practice, they are seldom written. They are usually perceived as too difficult to develop and not very valuable beyond their role as documents.

A two-pronged approach to this problem will be presented. First, the thesis introduces a methodology that facilitates formal specification development for commonly used interface protocols. The popular PCI bus protocol and the Intel Itanium bus protocol (preliminary version) have been successfully specified with this method. Furthermore, by verifying the specifications, previously unreported bugs and issues were uncovered in these protocols.

Second, the thesis describes how formal specifications can be used to automate many procedures that are now done manually. For example, to verify a design using simulation, test inputs and checking properties must first be written, and this process require considerable expertise and effort. However, if there is a formal specification, the test inputs and the checking properties can be generated automatically from it. This technique was used to simulate and verify the I/O component from the Stanford FLASH project whereby new bugs were found in the design and manual work was kept to a minimum.

Acknowledgments

I would first like to thank my husband, Geoff Beach. With his unique sense of humor and inexhaustible patience, he has supported me unconditionally through the ups-and-downs of my Ph.D. career. This thesis dissertation would not have been possible without him.

Likewise, I thank my advisor, Professor David Dill, for getting me started on this project and acting as my intellectual lighthouse throughout its development. This project was a perfect match for my interests and skills; it proved to be a delectable cocktail of theoretical depth and practical impact.

I thank my thesis committee members, Professors Mark Horowitz and Dawson Engler, for their detailed feedback and insightful comments; my thesis is now more tightly focused and solidly developed because of them.

I would also like to thank my mentors, Shin Horie, Lily Yang, and Kathleen Corcoran, for their wisdom and guidance; my research group colleagues, Aaron Stump, Alan Hu, Chris Wilson, Clark Barrett, David Park, Husam Abu-Haimed, Jeffrey Su, Jens Skakkebaek, Madan Musuvathi, Robert Jones, Satyaki Das, Sergey Berezin, Shankar Govindaraju, Uli Stern, Vijay Ganesh, for enduring the many practice talks and conference paper drafts; and the Computer Systems Lab support staff, Kevin Colton, Charlie Orgish, Joe Little, Thoi Nguyen, Mina Mardigal-Torres, and Kersten Barney, for repeatedly helping me out.

I would like to thank my friends for all the fun I had during the past four years: from the weekend parties to the hiking trips and from the lunch get-togethers to the impromptu conversations (in the 3A women's bathroom, in the hallway, in Rains, in Gates 352, ...). Alice Yu, Alison Hottes,

Armita Peymandoust, Atul Salvekar, Azita Emami-Neyestanak, Ayako Saiki, Ben Chen, Brian Lee, Chris Foley, Cindy Chen, Coco Yeh, Cristina Hristea, Dave Lie, Erika Chuang, Helen Kung, Jeff Gibson, Joy Ku, Kinshuk Godvil, Muditha Kanchana, Ken Crozier, Lada Adamic, Lalit Nathawad, Liadan O'Callaghan, Maryam Sarjoui, Mina Leung, Naomaru Itoi, Rae Yip, Ravi Ramamoorthi, Ravi Soundararajan, Rui Zhang, Scott Yam, Sundar Iyer, Tracy Fujieda, Valeria Bertacco, Wei Yu, Xiaoli Liang, Xin Tong, Yukiko Nakamura, Yung-Hsiang Lu, and last but not least, Ziyad Hakura.

Finally, I thank my family; my father, Hiroshi, my mother, Machiko, and my brother, Shinichiro for their love and support over all these years. They're my biggest fans and my biggest critics. I would not have come this far if it weren't for them. Arigato gozaimasu, hontouni osewa ni narimashita.

To my parents,
Hiroshi and Machiko

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Where This Thesis Fits In

Advances in manufacturing technology have allowed for increasingly large circuit designs to fit on a chip. Unfortunately, designer productivity has not kept up with design complexity, and design costs have climbed precipitously. Especially in the area of functional design verification, where engineers check for consistency between the functionality intended and the logic actually designed, progress has been slow. Various tools and algorithms have been developed to assist with finding problems in the design blueprint, but the general consensus is that designs have become so large and complex that the tools cannot possibly handle them. Tool capability is currently lagging so far behind that a fundamental change in design methodology is needed in order for the tools to have a fighting chance. The core problem is that tools and other low-level machinery have been developed, but the ability to use them effectively is still amiss. This thesis addresses this issue, and demonstrates how a particular design approach has various advantages by maximizing the value of verification tools. Specifically, by developing and describing effective uses of the available tools, this work proposes a change in design methodology so that verification of large designs that are common in industry can be made tractable.

1.2 Motivation and Goal

Functional Design Verification This thesis focuses exclusively on functional design verification. One of the first steps in design is to write down in a software-like language, typically referred to as a hardware description language (HDL), the logical structure and behavior of the circuit. This logical description is later compiled into circuit elements. Functional design verification aims to find problems at this early stage of design by analyzing the description written in HDL. The behavior and the structure of designs are usually so complex that, in many cases, the design is not entirely correct and will behave differently than expected once implemented as a circuit. Since it is prohibitively expensive to fix problems after the design is fabricated, functional design verification proves to be indispensable by finding problems early on, before additional work is done on the design.

Component-Based Design Due to the size of current day circuits, designs are not monolithic creations but rather compositions of components designed separately by different engineers. Most components are designed by engineers within the company, but some are created by engineers outside the company. The trend for component-based design is so strong that some companies have adopted a new business model where they sell pieces of design for standard functionalities. This way the buyer can simply integrate the standard component instead of reinventing the wheel. Consequently, integration of different components, especially ones acquired from an outside party, has become an important and frequent procedure. Unfortunately, it is also an Achilles heel because assumptions about interface behavior are difficult to communicate between engineers. Often the component's behavior is misunderstood so the integration is incorrect and the design as a whole does not function correctly.

The Current State of Interface Specification Because of its pivotal role, the component interface protocol must be precisely defined and accurately communicated by a specification. However, specifications widely in use today are still written *informally*, in natural languages like English. Consequently, interface specifications such as those of standard bus protocols are often wordy,

ambiguous, and contradictory: all problems that can lead to different interpretations of the specification. Furthermore, because a natural language specification is not amenable to analysis, there may be unnoticed problems with the protocol. These "bugs" may cause the interface components to behave incorrectly under overlooked scenarios. Thus, even if two components correctly implement the protocol, because the protocol itself is incorrect, the interacting components may deadlock or lose a data phase. Such problems are hard to detect with an informal and incomplete description of the protocol. Considering the critical role that an interface specification plays in current day design methodology, this primitive state of specifications is surprising.

Formal Interface Specification In contrast to an informal specification, a formal version, which uses precisely defined logic symbols and operators, promotes logical clarity and allows for automated analysis. For example, consider the following informally stated rule from the specification of the PCI (Peripheral Component Interconnect) bus protocol [**?**], a popular protocol used in many designs: "only when IRDY# is asserted can FRAME# be deasserted." The formal version may look like this:

$$previous(irdy)\ IMPLIES\ previous(frame) \wedge \neg frame$$

("if *irdy* was true in the previous cycle, then *frame* can become false in this cycle").

A formal specification, by construction, does not tolerate ambiguity, and thereby forces the protocol designer to define the behavior precisely. This not only alleviates the communication problem, but also increases the likelihood of a correct protocol because formal development induces discipline. Another advantage is the ability for the specification to be analyzed by a computer. Automated analysis is immensely useful for debugging the protocol because it can simulate the protocol before any implementations are built. This way, any unexpected implications of the protocol can be discovered and resolved. Therefore, a formal specification promises a clearer description and a less problematic version of the protocol.

Thesis Goal Despite these arguments, in practice, formal specifications are rarely used. The

reason seems to lie with a perceived cost-value problem; they are often considered too costly for the benefits they promise. Specifically, they are criticized for their lengthy development time and the investment needed for formal verification training. For many, the value of a correct specification does not justify these costs, and precious time and resources are allocated for more pressing design needs. Thus, to counter these disincentives, this work developed a formal specification methodology that attacks this cost-value problem from two angles:

- **Cost Side:** Minimize the cost problem by making the formal specification process easier. Develop a specification style that leads to a correct specification with less effort than ad hoc methods.
- **Value Side:** Increase the value of a formal specification beyond its role as a documentation. Clearly, if the specification shortens implementation design time in more ways than one, the investment in a formal specification would be more acceptable.

1.3 Scope of the Thesis

This work focuses on signal level protocols as opposed to higher level specifications. The goal is to describe the behavior of each wire or pin at the interface so that the required binary values of all the signals would be specified for *every* clock cycle. In contrast, a high level specification describes desirable properties but doesn't necessarily specify the signaling configuration at each point in time. Properties such as "a data phase always completes" can be used to verify a design, but they are too high-level to fully describe the exact signaling protocol. High-level specifications have their uses, but this thesis aims for complete specifications that can be used to design implementations manually or automatically.

This signal-level focus is motivated by the core goal of making formal verification feasible for current designs in industry. Specification research has, until now, mostly concentrated on high level specifications with some success. But by focusing on low level specifications, we hoped for and achieved new advances in verification of large designs. To demonstrate this point, the two protocol

examples in this thesis are both signal-level bus protocols widely in use. They were chosen based on their importance for many applications and their complexity which made specification challenging.

It must also be emphasized that this work is limited to self-contained specifications of the interface. It cannot describe the actions at one interface in relation to an event at another interface, which would essentially be specifying global properties. Although admittedly such descriptions would be useful for verifying components with the two interfaces, unless both interfaces commonly come in pairs, the usefulness of the dual protocol specification would be very limited. Also, the complexity of such a description is beyond current capabilities and would be more appropriate for future work.

1.4 Background and Related Work

Many specification languages for formally describing behavior have been proposed over the years. The languages range in expressiveness, and their different levels of granularity determine their appropriateness for different applications. Some have become popular because they are input languages for powerful verification tools such as a model checker. Many simply emphasize their useful constructs and expressiveness to convince users to adopt the particular language. This section is a brief overview of the specification languages and description logic that specification writers can choose from.

1.4.1 CSP

One of the earlier specification language is the Communicating Sequential Processes (CSP) language. It was introduced by C.A.R Hoare in a 1978 paper [**?**], and has subsequently been adopted by many researchers, some who built upon it to develop their own notation. The language focuses on the communication between different processes, and the internal computations are usually not described[**?**]. This emphasis is similar to this thesis's goal for describing interfaces and treating components as black boxes. Another feature of CSP is its non-determinism construct which allows

description of non-deterministic behavior. As described later in section 2.4.1, this capability is critical for an interface description. However, despite these positive features, there are several reasons why CSP is not appropriate for signal level specifications. First, CSP, in its original form, is a high-level language which abstracts away timing. The precise times when events occur cannot be described, and for industrial applications where timing is of utmost concern, this drawback makes CSP inappropriate. For example, a protocol may require the completion of a handshake to occur within four clock cycles; such a requirement cannot be described using CSP. Second, it is not clear how CSP would allow the specification process to be easier or the resulting document to be more useful than the status quo. Although there is a tool called FDR (Failures/Divergences Refinement) which simulates CSP descriptions, there are other model checking frameworks that are more intuitive to use. In addition, application of a CSP specification as a verification aid other than as a formal document is not known.

1.4.2 Temporal Logic

Temporal logic, which forms the basis of many description languages, has a more precise notion of time than CSP. In linear temporal logic (LTL) [**?**], one can express properties such as "f is true next time ($\bigcirc f$)" and "f is true until g is true (fUg)." Temporal logic can be thought of as propositional logic with operators to describe changes over discrete points in time. There are mainly two types of temporal logic: LTL and computational tree logic (CTL) [**?**]. LTL is based on a linear sequence of events while the underlying model for CTL is a tree. Both are actively used to specify designs, and in a few cases [**?**] have been used to specify the signal-level protocols that are the focus of this thesis. However, perhaps because of their complex nature, they have not been adopted by many verification engineers in practice. Many who have used CTL, for example, can attest to the difficulty of precisely stating the desired formula; often, composition of the operators can result in an unintended and incorrect expression. As a result, CTL or LTL may not be appropriate for those who have not had formal verification training.

1.4.3 Languages in Industry

There are a few formal specification languages that are employed in practice. They are used, for example, to write assertions which check behavior at certain execution points of the design. Some languages also allow users to write constraints on the inputs so that the user can avoid explicitly writing input sequences. These formal languages are often used in a simulation environment as opposed to a formal model checking process; the designs are too large to be model checked, but the semi-formal method of combining the formal (formal description, constraint solving) with the informal (simulation) has proved to be very effective. The two most popular languages for this semi-formal framework are Synopsys's VERA and Verisity's Specman. These languages' strength lie with the fact that many verification engineers, driven by the usefulness of the accompanying tool, have chosen to learn the language and write formal properties. Thus, these languages have arguably accomplished the first goal of making the formal specification process reasonable. However, upon further inspection, it is clear that neither languages are used to write complete specifications but are instead used to write an ad hoc list of properties. These properties do not form the complete specification this thesis aims for and are closer in spirit to the high level checking properties described earlier. Therefore, the languages have yet to demonstrate their suitability for interface specification. Furthermore, each company has its own proprietary description language which is incompatible with tools from other companies. A verification engineer must re-learn a new language for every new tool and possibly write multiple specifications for the same interface because different tools are needed. These annoyances hinder the acceptance of formal specification as a standard procedure in design.

1.4.4 State of the Research Field

Until recently, formal specification research has focused mainly on developing tools or inventing languages. For tools, much effort has gone into *model checkers* which investigate whether a model of a design satisfies a given property for all possible execution paths. Some popular model checkers

are SMV [**?**], Murφ [**?**], and SPIN[**?**]. In addition to the vast effort invested in improving the raw horsepower of these tools, there has been a sustained research effort into languages. For example, researchers reason about the implications of composing two specifications or the mechanics of abstracting a design into a simpler model. Tools research aims to improve the dynamic capabilities of verification, while language research attempts to do the best static analysis and processing since the dynamic capabilities are limited.

A plethora of new languages, most of which are variations on the languages discussed in the previous section (CSP, CTL, LTL), have been introduced. The aforementioned model checkers all have their unique languages and additionally, there is LUSTRE [**?**], Esterel [**?**], and countless others, many of which are developed for a particular tool. This trend is problematic because many practitioners are reluctant to learn so many languages. Within this context, it is important to note that this work specifically does *not* propose a new language, but rather a specification *style* that can be applied to any existing language. The language independence allows for the methodology to be used for many LTL or CTL based frameworks such as SMV, LUSTRE, and FoCs [**?**], and even non-traditional specification languages such as Verilog.

This thesis answers the central question of how to write, debug, and use a specification once the language and the tools have been chosen. Until now, very little work has been done on ways to write specifications; tools assume the existence of them or are just used with an ad hoc list of properties. There is a lack of understanding of specification methodology, and consequently, specifications are written haphazardly, in a free form way, and are checked very unsystematically if at all. These issues cause specification development to be unnecessarily burdensome and problematic. *The main contribution of this thesis is the development of a general methodology that helps produce self-contained, complete, and correct specifications, while steadfastly adhering to the cost-value goal.*

1.4.5 Summary of the Thesis

In this section, the thesis's main results are discussed. First, the attributes of the specification writing style are described, followed by an outline of the specification debugging methodology,

without which the writing style would not be feasible, and concluded with an explanation of how the specification can be exploited in many different ways.

Specification Writing and Debugging

A simple yet powerful specification style that is resistant to errors is defined. Unlike many other specification frameworks, this writing style does not require nor does it allow non-deterministic expressions and CTL/LTL operators. The thesis demonstrates how even with this restriction, a specification can still describe non-deterministic behavior and specify rich properties. Furthermore, by not requiring these operators, the writing style can be used with Verilog or VHDL, a fact which opens the door to a much wider audience. Since understanding esoteric formal logic concepts such as non-determinism and temporal logic is a major barrier for specification in practice, this thesis's demonstration that a protocol can be specified completely without these concepts is a contribution for that reason alone. However, there are other advantages.

For one, by eschewing these constructs, a specification is executable within many environments and platforms. With no extra work, the specification can be simulated together with an implementation to verify the implementation's compliance with a specification for example. Or, because the specification is also synthesizable, it can be used in an emulation framework for hardware testing of the design. In contrast, a specification with non-deterministic expressions is only executable with special tools. Therefore, the only way such a specification can be used directly with an implementation is if the implementation is ported to the special tool framework, an unrealistic task in most cases. Thus, by restricting the methodology to basic operators, the resulting specification is made more useful.

Another advantage of this specification style is due to the trickiness of temporal logic. Many experts can attest to the difficulty of writing exactly the CTL or LTL formula that is needed to specify a property. In most cases, errors are introduced in the specification because the temporal properties are incorrectly written. However, this thesis demonstrates that these complex operators are not needed for most interface protocols, and so, specification writers can adhere to the basic

engineering principle of “KISS - Keep It Simple Stupid” to produce high integrity specifications.

In addition to avoiding non-deterministic and CTL/LTL operators, this methodology has the distinctive feature that *there is no explicit state machine representation of the protocol.* Instead, protocols are described by a collection of compact properties and small, standard state machines such as two-state set-reset machines. In contrast, most formal protocol descriptions are currently written as explicit state machines. The problem with this approach is that the state machines quickly grow large and unwieldy. Additionally, because most protocols are originally defined by a list of rules and not as a state machine, much processing effort and time is needed to design a state machine from these rules when creating a formal version of the specification. Not only is this process time-consuming, but in creating a state machine, many errors can be introduced by the specification writer. Clearly, simply formalizing each rule in the informal original specification to create a collection of formal properties would be much easier.

However, until now, this has not been done because it is commonly thought that to define protocol behavior exactly and completely with properties is too difficult. For example, it is not immediately clear how to ensure that there are enough properties in the specification to rule out all illegal and unexpected behavior. It seems more natural to design a state machine that describes what can happen and thereby avoid having to explicitly define everything that can’t happen. Also, properties may be contradictory in certain states, but it seems impossible to check this without an explicit execution model. Indeed, in general, checking a simple list of properties directly for correctness seems difficult because behavior is only implicitly, and not explicitly defined.

To counter this, *a specification debugging methodology that does not require an explicit state machine representation of the interface protocol but instead can be applied directly to the specification properties was developed.* This methodology makes property-based specification development feasible. In addition, this thesis *introduces a technique that takes a property-based specification and generates example protocol traces* so that with a waveform viewer, a user can view example transactions to become familiar with the protocol. This is useful because with just a list of properties, it is difficult to picture the protocol in action, especially when compared to an explicit state

machine, but this feature circumvents that problem.

Another possible drawback of using a collection of properties for a specification is that the protocol may not be implementable. When put together, the properties may be so unexpectedly constricting that some of the specified modes of behavior may not be allowed to happen at all. The key question is how to check that this illusory freedom of choice does not exist in the specification and ensure that all the behavior modes are indeed implementable. To address this question, a theorem is proven in this thesis; *if the specification properties all obey a certain stylistic rule and together satisfy a simple-to-check characteristic, then not only is there a state machine implementation that satisfies the specification, but for every choice offered by the specification, it is possible to implement that choice in a state machine.*

To summarize, there are many positive reasons for writing and structuring a formal specification in the way proposed by this thesis. Specification writing becomes more accessible by removing certain expertise requirements, less error-prone by keeping the operators used to the most basic, and less laborious by staying closer to the original form of the documentation. At the same time, the resulting specification can be used for many applications, debugged directly without any implementations, and guaranteed to be implementable. To demonstrate the methodology on a meaningful example, the widely-used bus protocol, PCI (Peripheral Component Interconnect) 2.2 [**?**], was first chosen. In the process of the specification development, the debugging methods discovered three previously unreported bugs in the standard itself. As a second example, a more advanced pipelined protocol, the Intel® Itanium™ Processor bus protocol, was formally specified. With this protocol, the specification writing and debugging successfully uncovered issues that later led to changes in the protocol. These examples illustrate the feasibility of the specification style and the power of the specification verifying methods.

Using the Specification

Once an interface is completely and correctly described by a specification, the logic encapsulated in that specification is too useful to ignore. And if a specification is written in the style prescribed

here, it can easily be used for many applications. *This thesis describes how, when verifying a design using simulation, the specification can be used to automatically generate input sequences, output checking assertions, and a simulation coverage metric for that design.* As mentioned earlier, a specification of this style is directly executable so without any extra work, the specification can be used as a output checker during simulation. Also, because of the style, legal input sequences for the interface can be generated automatically and dynamically using the specification. This feature is of significant value because input test bench development is very costly in terms of time and effort. Finally, a simulation coverage metric and appropriate input biases can be determined from the specification. These two are highly important for unrepetitive and meaningful simulation runs.

The core technique of generating input sequences from properties or logical constraints by itself is not new [**?**]. However, *this thesis is first in combining the input generation technique with methodology for writing and debugging complete specifications.* Therefore, unlike current methods, the properties used to generate the input sequences are complete and verified. Consequently, the inputs generated with this methodology are guaranteed to be correct and also the most general possible. In contrast, ad-hoc properties written with the express purpose of input generation are currently not verified mechanically, and thus, are very likely to be under-restrictive or over-restrictive. Under-restriction leads to the possibility of incorrect input sequences, and this hinders design verification because any discovered problem can be attributed to either incorrect inputs or a true bug in the design. Additionally, overly restrictive properties are a problem because, during simulation, the full input space cannot be explored and thus the input sequences may not exercise the design thoroughly enough so that meaningful bugs can be found. Similar principles hold when the properties are used to check the outputs of a module; if the properties are under-restrictive, protocol violations will be missed, and if they are over-restrictive, behavior will be incorrectly flagged as a bug. Therefore, the ability to verify properties directly without an explicit state machine model, so that properties are neither under-restrictive or over-restrictive is very important for these additional reasons.

Another contribution of this work is *an algorithm which exploits of the structure of the specification so that the input generation is orders of magnitude more space-efficient.* This space efficiency is critical for large interfaces where memory blowup problems may hinder automatic input generation. The structure or the style for the specification was originally developed to maximize correctness of the resulting specification, but the structure has proved to be invaluable for optimizing memory usage when using the specification dynamically. The algorithm accomplishes this by distinguishing the different roles, which are made clear by the structuring, the logic expressions play in the specification.

*The specification "use" methodology was demonstrated on an extensively debugged and fabricated design, the I/O component from the Stanford FLASH project [**?**].* An input generator, correctness checker, and coverage metric monitor, were all automatically derived from the formal PCI specification written in the prescribed style. These tools were used to simulate and verify the design, and as a result, previously unreported bugs were discovered. The experiment shows how the process of finding bugs is now nearly automated and also illustrates the effectiveness of the methodology by finding previously overlooked problems. Furthermore, it demonstrates the space efficiency of the algorithm by tabulating an order of magnitude reduction in the variable numbers and memory usage for this example.

Chapter 2

Specification Style

2.1 Introduction

This chapter introduces the specification methodology. The specification style was developed with hopes that it is applicable to as many protocols as possible. It is not tailored or optimized for any particular protocol, but it did evolve from attempts to describe the PCI bus protocol coherently. Although the whole range of interface protocols are of interest, by focusing on a well-used standard protocol such as PCI, it is felt that the impact of the work would be maximized. Also, from a cost-value perspective, the fact that the effort invested in a formal specification can be amortized over countless systems for popular protocols makes PCI a logical choice.

The advantages of a formal specification are described extensively in the introduction chapter, but its importance is reiterated here based on another factor. Formal specifications address a subtle point missed in practice; an informal specification may have inconsistencies that a human reader will not notice, or may be missing rules that a human reader may infer automatically, but, because it is inconsistent and incomplete, any correctness reasoning is impossible. A practical consequence of this is that a good protocol compliance checker can not be created from such a document. Thus, there are pragmatic reasons why formal specification development is important.

[0]This chapter is based on material first published in *Proceedings of the Third International Conference of Formal Methods in Computer-Aided Design* (FMCAD 00), 2000 [**?**].

2.1.1 Structure of the Chapter

After related work is discussed, achievements accomplished by the specification methodology is outlined in “Cost-Value Achievements” (section 2.3). The subsequent section, “Two Core Characteristics of the Style” (section 2.4), describes the foundation of the methodology and how it leads to the cost-value improvements. And on top of this foundation, the “Property Style Rules” (section 2.5), further assist in developing correct specifications.

2.2 Previous Works

There has been a few other bus protocol specification projects. In 1999, Chauhan, Clarke, Lu and Wang [**?**] specified PCI [**?**] using CTL. Our specification has advantages that a CTL one does not have as outlined in the first chapter. In 1998, Mokkedem, Hosabettu, and Gopalakrishnan formalized higher-level properties of PCI involving communication over bus bridges [**?**]. Their specification is almost unrelated to the one here, which focuses on the low-level behavior of individual signals. In fact, the methodology is more closely related to the following two papers. Ogoubi and Cerny describe a synthesizable observer written in an HDL in their 1999 paper [**?**], but they do not suggest the stylistic structuring proposed here. Kaufmann, Martin, and Pixley also have a similar approach as outlined in their 1998 work [**?**], which proposed using logical constraints for environment modeling. However, they do not give guidance on how the properties or constraints should be written.

2.3 Cost-Value Achievements

As outlined in the introduction chapter, the goal is to develop a specification methodology that allows the specification writing to be easier and the specification to be more useful. Throughout this thesis, this is referred to as the *cost-value* goal. This section outlines the specifics results achieved for this goal, while the next section (“Two Core Characteristics of the Style”) describes

how the style choices that were made *allowed* these results.

1. **LTL and CTL are not needed** Specification writing is made more accessible in two ways. First, formal verification expertise is not required to write the specification. In contrast, many specification frameworks require knowledge of LTL (linear time temporal logic)[**?**] or CTL (computation tree logic)[**?**] as evidenced by numerous projects such as CMU's PCI specification [**?**] and IBM Haifa's FoCs software [**?**]. Because this methodology does not require the complex constructs of these languages, the formal specification can be and has been written in a hardware description language such as Verilog, a language familiar to many engineers. This feature partly counters the training cost argument against formal specifications. More generally, the style is language-independent (indeed, the specification can still be written in LTL or CTL if desired). The methodology can be applied to any of the existing tool frameworks and languages such as SMV[**?**], FoCs[**?**] or LUSTRE [**?**].

2. **Many small properties instead of complicated state machines** Second, because the style is based on writing many small protocol rules and purposely avoids writing large state machines, the specifications have been found to be easier to write and maintain. Many other specifications are based on writing a large state machine with actions specified for each state. Designing such a state machine correctly is a complex, error-prone task. This method instead relies on many small and generic state machines, but the bulk of the specification is done using compact protocol rules. The formalization used here requires less manipulation and results in fewer opportunities for error because most existing natural-language specifications, such as the PCI 2.2 specification, are already written as a list of rules. To illustrate this point, a few examples from the document are listed here.

`''TRDY# cannot be driven until DEVSEL# is asserted.''` (section 3.3.1)

`''Only when IRDY# is asserted can FRAME# be deasserted''` (section 3.3.1)

3. **Implementability guaranteed** This specification style together with a simple-to-check property guarantees the existence of an implementation and more precisely, a stronger property called *receptiveness*. With receptiveness, not only is a single implementation guaranteed, but for every choice offered by the specification, it is possible to design a state machine that implements

that choice.

Whether a specification can be implemented as a state machine is important because much design effort can be wasted on a protocol that is inherently impossible to implement. And, receptiveness ensures that *all* behavior modes in a specification, and not just a single mode, are implementable.

4. **Application as an environment model and property checker** To maximize its value, the style allows the specification to have multiple functions. If the implementation design is small enough to be formally verified with a model checker, the formal specification is immediately useful. *With no extra work,* it can be used to constrain the inputs to the design under verification, and at the same time, it can be used as the list of properties to check for. The ability to model an environment (i.e. the inputs) without writing any implementation code allows the user to avoid a great deal of non-trivial, tedious work. For example, this technique was used by Govindaraju and Dill; using the PCI specification written in this style, they were immediately able to begin formally verifying a PCI driver implementation [**?**]. The specification constrained the state space, and also gave them properties to check for. This assume-guarantee approach is not new, but the style easily allows it while an *ad hoc* interface specification probably would not.

For designs that are too complex to be formally verified and can only be simulated, the specification can be used as a *monitor* during simulation runs. A monitor observes the outputs of the design under verification and flags behavioral errors during simulation. It is frequently used for functional verification and is also referred to as an *output checker* or a *snooper.* The specification can be directly used as a monitor because unlike many specifications, this specification is executable and can be written in a language such as Verilog. For example, at Intel®, the specification that was developed for the Itanium™ processor bus protocol has been mechanically translated into a C++ simulation checker. Again, writing properties to check simulation behavior is not new, but using an interface specification directly in this way is a novel contribution. There are other direct applications for the specification, and they are described in Chapter 6.

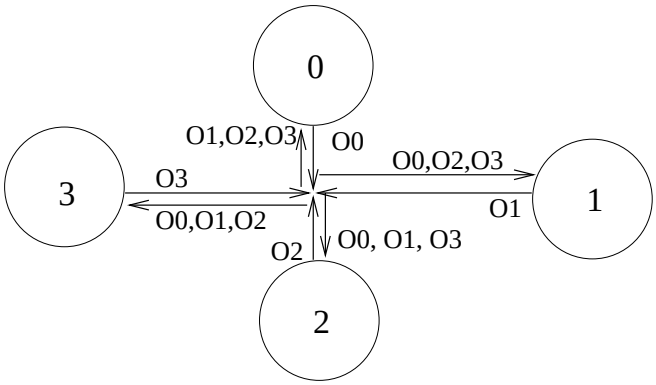


Figure 2.1: The System View

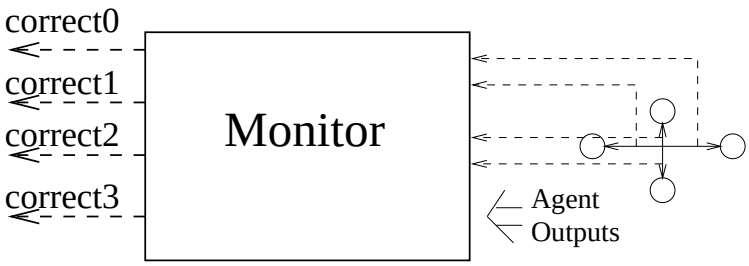


Figure 2.2: The Monitor

2.4 Two Core Characteristics of the Style

There are mainly two choices that were made on how to write a specification that allowed the advantages listed in the previous section. The first is to write the specification as a *monitor,* and the second is to focus on writing many small properties and many small state machines as opposed to writing the specification explicitly as one large state machine.

2.4.1 Characteristic 1 : Monitor-Based Specification

A monitor (Figure 2.2) is an observer in a group of interacting modules, or *agents* which communicate via a set of protocol rules (Figure 2.1). Its main task is to flag an agent as soon as it fails to uphold the protocol. It is a module where the agent output signals are its inputs, and boolean $correct_i$ signals are its outputs. When an agent (or several agents) violates the bus protocol, the monitor singles out the erring agent(s) by making the corresponding $correct_i$ signal(s) false. If

correct$_i$ is true, the monitor is indicating that the agent *i*'s current outputs conform to the specification.

The specification style is based on writing the specification as such a monitor.[1] After all, the monitor must have exactly the protocol information to decide on agent compliance so it is not unnatural for the monitor to be a form of protocol specification. For every clock cycle, a monitor specification defines correctness instead of defining behavior. The concept is not new; many specification frameworks, such as LUSTRE [**?**] are based on monitors or observers. In fact, a monitor specification has many of the advantages of temporal logic.

Another Type of Specification

Most other formal protocol specifications are *generator-style* descriptions (Figure 2.3). With this style, for every agent at the interface, there is a corresponding state machine specifying the correct outputs. The inputs and outputs for these specification state machines are exactly the inputs and outputs of the agent specified; this is in contrast to the monitor-based style where the specification outputs are *correct*$_i$ variables and not the agent outputs. A generator specification explicitly gives a correct output vector on every clock cycle instead of simply indicating the correctness of an output. For example, an agent based specification would state that protocol signal *out* is "1" (or "high" or "true") at time step 7, and for the same property, a monitor specification would specify that if *out* is not "1" at time step 7, the agent is incorrect. This difference seems to be subtle and inconsequential, but the generator style has one major drawback that the monitor-based style does not have; it has difficulty handling *non-determinism.*

Non-Determinism in Specifications

Since users are usually given leeway in implementing the protocol, non-determinism is an integral part of most specifications. For example, a protocol may allow a receiver implementation to respond with an "acknowledge" anytime from three to eight clocks after the sending event. Thus,

[1] Only the monitor is written by the specification writer. The agents in the figure are to be later implemented by someone else.

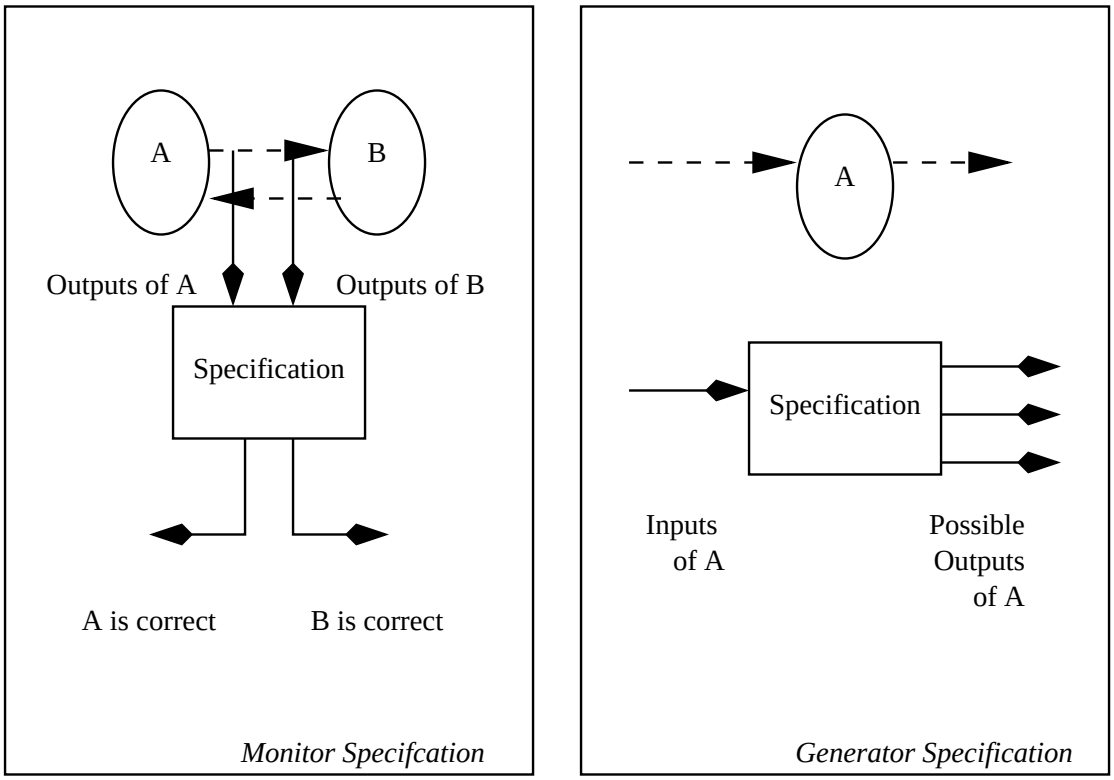


Figure 2.3: Monitor Specification vs. Generator Specification

from an observer's perspective, the agent may respond on the third clock, the fourth clock, ..., or the eighth clock, and this behavior is non-deterministic. The flexibility allows the implementor to choose between a fast response because the particular system requires it or a slow one because data processing is required.

For writing a specification, this flexibility can be an annoyance. Since the most general description is needed, the whole range of behavior must be considered correct. For a generator specification, the concept that "acknowledge can be true in the third cycle, or it can be false and be true in the next cycle, ... or it can be false and be true in the eighth clock cycle" must be expressed in order to describe non-deterministic behavior. Usually, this would require special constructs in the language where a *set* of possible output vectors can be specified. Therefore, only special languages with this feature can be used for generator specifications, and a verification engineer may not be able to use her language of choice. Furthermore, the special non-deterministic construct may make a specification more complicated and mechanical use of it difficult. For example, a document with such a construct is not directly nor immediately executable.

Advantages of a Monitor-Based Specification

1. A monitor specification can describe non-determinism deterministically. In contrast, a monitor-based style handles non-determinism with a simple solution. The crucial observation is that a single propositional formula can describe a range of acceptable behavior. The example property, "an acknowledge must happen sometime between three to eight clocks", can easily be specified in propositional logic with a simple counter (*counter*) and a set-reset variable (*ackHappen*).

$prev((counter = 7) \wedge \neg ackHappen) \rightarrow ACK$

"If it has been seven clocks and ACK has not been true yet, then, in the next cycle, ACK must be true"

$prev((counter < 2) \rightarrow \neg ACK$

"If it has been less than two clocks, then ACK cannot happen yet."

Since a monitor-based specification only requires descriptions of correct behavior, these propositional formulas are precisely what is needed in the specification. Thus, on each clock cycle, as long as the outputs satisfy all the formulas such as these, the $correct_i$ output is true. Consequently, even when describing non-deterministic behavior, the specification itself is deterministic because its outputs (the set of $correct_i$'s) can always be set deterministically (either the outputs satisfy the properties or they don't).

2. A monitor specification can be written as a list of properties. The monitor-based style also allows specification to be written as a conjunction of propositional formulas or *properties.* As described in the previous paragraph, properties can be used to specify behavior if only correctness has to be specified (monitor-style) and not explicit behavior. In comparison, a generator specification inherently relies on (often very large) state machines to determine outputs and cannot be written as a list of properties. Fundamentally, since the generator specification is so close in form to the actual implementation, it is inevitable that a generator specification would structurally resemble a very large state machine. A monitor specification does not have this handicap, and can be written in a form that's more maintainable, less error-prone, and simpler to write.

3. A monitor specification is executable. Because of the monitor-style's handling of non-deterministic behavior, a specification of this type is executable. This is valuable because as long as the monitor is written in the language of the implementation, it can be directly connected to the design. Then, it can flag errors and assign blame to the appropriate module in a system-level simulation. To use a non-deterministic specification, such as an generator one, in the same manner is not as straightforward.

4. A monitor specification can be used to model the environment. When model checking a design, the inputs can be set without writing an implementation of the environment but by using a monitor-based specification. Namely, one would model check the design by conditioning all the properties to be verified, with "assume the interfacing agents have been correct so far according to the monitor". For example, if p is the property to be model checked, and agents i and j (whose implementations do not exist) form the environment, the property to be model checked is "there

is no execution path where $correct_i \wedge correct_j$ have been true so far and p is false in the current state ($\neg$E[($correct_i \wedge correct_j$) U $\neg$p] in CTL)" where $correct_i$ and $correct_j$ correspond to the output signals of the monitor for i and j. The monitor and the condition in the checking properties correctly constrain the inputs to the design. This use of the monitor is very similar to what is described in [**?**]. Again, this application is possible because the specification is based on properties instead of explicit actions; it is difficult to see how an generator description would be used this way.

5. Interface-centric view leads to less work. With an generator-style approach, the specification writer would have to individually write a specification for each agent at the interface. Furthermore, specifications for *all* of the agents would need to be written before the writer can verify any of the agent specifications. If the writer took the interface-centric view and wrote a monitor specification, the interface can be specified in one fell swoop. Only one self-contained specification needs to be written instead of multiple specifications.

2.4.2 Characteristic 2 : Many Compact Properties and Generic State Machines

There are many ways to write a monitor specification. The most obvious way is to implement the monitor as a state machine. Depending on the current state of the monitor, correct outputs of the agents are defined, and the current state is defined by the preceding sequence of agent outputs. The drawback of this approach is that, again, the state machine can grow very quickly in size. This is mainly due to the non-deterministic nature of protocols. Since a state machine based monitor specification is essentially a deterministic finite automaton (DFA) equivalent to (i.e. recognizing the same agent output sequences as) the non-deterministic finite automaton (NFA) representing the protocol, the number of states in the monitor state machine is exponential (2^k) in the number of states (k) in the NFA representation. Thus, writing an explicit state machine representation of the monitor may easily lead to un-readable and error-prone specification simply because of the machine's size.

This leads to the second stylistic guideline for a formal specification: *write the monitor specification as a list of properties that should hold true for the interface protocol and avoid writing*

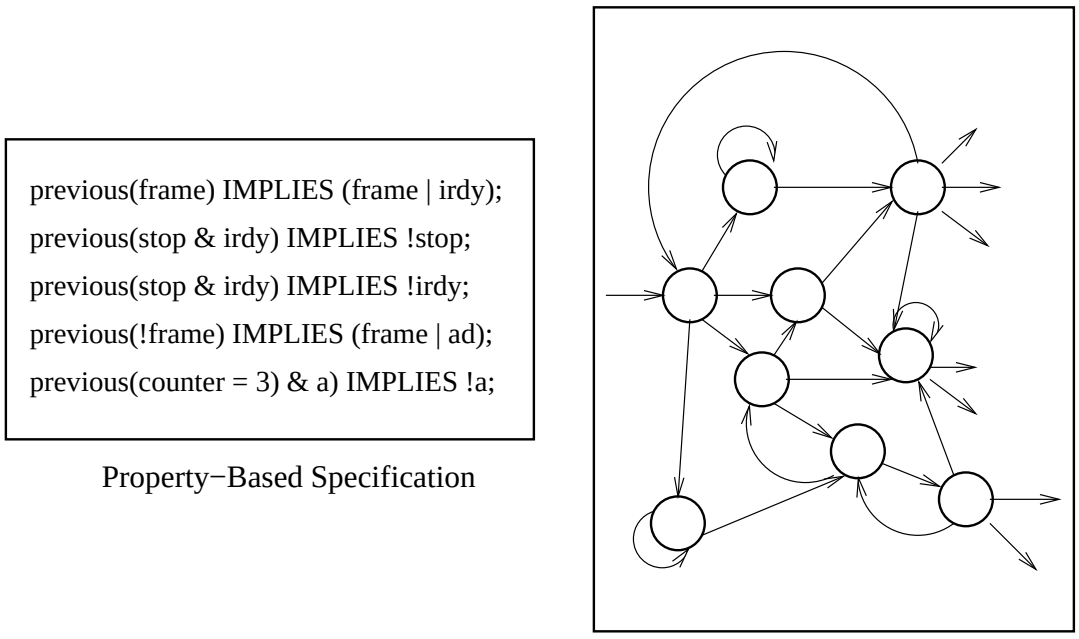


Figure 2.4: Property-Based Specification vs. State Machine Based Specification

an explicit state machine to implement the monitor (Figure 2.4). This approach is possible because the specification is written as a monitor, and was chosen because of the advantages of conciseness, ease of maintenance, and simplicity. The specification properties are of the following form:

$$prev(signal_0 ... \wedge \neg signal_j ... \vee variable_0 ... \wedge \neg variable_k) \rightarrow signal_i \vee ... \wedge \neg signal_n$$

where "$\rightarrow$" is the logical symbol for "implies". The antecedent, the expression to the left of the "$\rightarrow$", is a boolean expression containing the agent output variables and auxiliary state variables, and the consequent, the expression to the right of the "$\rightarrow$", contains *just the agent output variables.* The allowed operators are AND, OR, and NOT. The *prev* construct allows the state of the signal a cycle before to be expressed. It can be nested although in the examples attempted, no more than two nested *prev*s were needed. The *prev* construct is the only special construct that is needed, and can be easily implemented in most languages. Note that the more powerful constructs provided by LTL and CTL, which are difficult to implement, are purposely avoided. A contribution of this thesis

is that this decision does not hinder the description of some well-used signal-level bus protocols while allowing for many advantages. For clarification, an example property from PCI is described.

$$prev(trdy \wedge stop) \rightarrow stop$$

means "if the signals *trdy* and *stop* were true in the previous cycle, then *stop* must be true in the current cycle" where a "true" signal is asserted and a "false" signal is deasserted.

The *prev* construct is not enough to retain state information. State machines are needed for this. But instead of relying on large custom state machines, the style relies on many, small standard state machines which each track one thread of information. An example is a 2-state, set-and-reset machine which becomes set when a certain event happens and stays set until it is no longer needed. It is used to record certain information such as whether the transaction is a read or a write. Another example is a counter which counts the number of cycles from a certain event, or counts the number of occurrences of a special event. Only these two types of state machines were needed for describing both examples in this thesis. A key point is that a protocol can be written with a few standard state machines, even if it is as complex as a pipelined protocol.

To construct the monitor, the properties are directly used to determine the $correct_i$ signals. Assuming that each property constrains the behavior of only one agent, the properties are grouped by the agent which they constrain. When the agent output signals make all the properties of one agent true, the corresponding $correct_i$ signal is true; otherwise, it is false. Thus, $correct_i$ is a conjunction of all the properties specifying the behavior of agent *i*. The following is the assignment statement for $correct_i$, where $property_i^j$ pertains to agent *i*.

$$\begin{array}{c}\text{if } (property_i^0 \wedge property_i^1 \wedge ... \wedge property_i^n) \\ \text{then } correct_i = \text{true, else } correct_i = \text{false}\end{array}$$

Therefore, the monitor is a list of propositional formulas ($property_i^j$ definitions), auxiliary state variable assignments, and $correct_i$ assignments. There is no conversion of this to a state machine; this is precisely the code for the monitor.

2.5 Property Style Rules

The previous section discusses the foundation of the specification style; the specification should be monitor-based and be decomposed into multiple properties. Similar styles of writing have been done by others who write observer-based descriptions or by those who use temporal logic. While acknowledging that the method is not new, the thesis uniquely demonstrates and argues that this structure of writing is best for the signal-level protocols used in practice. In contrast, the stylistic rules introduced in this section are original contributions.

2.5.1 First Rule : Separability of the Properties Rule

Each property can only constrain outputs of one agent.

For each property, there should only be one agent to blame when the property is broken. Consequently, a property can only restrict the outputs of one agent; if it dictates the output behavior of two or more agents, multiple agents can be held responsible for a single broken property. Since it is exactly the current state variables that are constrained by the properties, all current state variables of a property, must be outputs of the same agent. Equivalently, since for a particular agent, outputs of other agents are its inputs, the current state variables must all be the agent's outputs and not its current inputs. If signals o_a and q_a are outputs of the same agent and r_b is an output of another agent, the first property obeys this restriction, while the second doesn't.

$$\text{correct : } prev(r_b) \rightarrow o_a \wedge q_a$$
$$\text{incorrect : } prev(r_b) \rightarrow o_a \wedge r_b$$

This rule is called the *separability* rule because it allows properties for different agents to be separated and evaluated independently. This separability factor is important for a specification because with it, a specification can be guaranteed to have an implementation as proved in section 3.6.3. The main need for the separability of a property is to uphold an important principle of specifications: *it should be possible to implement an agent based on information solely from*

the specification, and know that the implementation can interact correctly with any other agent upholding the specification.

Independent Implementability If multiple agents are responsible for upholding an inseparable property, the implementation of a single agent must be able to do the impossible act of predicting the behavior of the other agents. Thus, the only way such a system can be designed is for the different agents to be designed *together* which runs counter to the principle of independent implementability. For example, a bad, inseparable specification would be $a = b$, where a and b are outputs of two different agents. An implementer of one of the agents cannot safely set the value of a without knowing what the implementer of the other agent will set the value of b to. Such a functionality is not independently implementable. Equivalently, a monitor for such a specification cannot blame a single agent when the property is broken. If $a = b$ does not hold, it is impossible to decide whether a is wrong or b is wrong. Thus, it is apparent why "a broken property can only blame one agent" is a sufficient condition for the specification to uphold the above principle.

Removing Illusory Freedom Consider the specification "$\neg(a \wedge b)$," where a and b are outputs of different agents. Such a formula might result from an attempt to specify mutual exclusion. In this case, an implementer can only set $a = 0$ safely, in case the implementer of the other agent may set $b = 1$. But the implementer of b can only safely set $b = 0$, too, so the specification could just as well be $a = b = 0$. In other words, this specification allows *illusory freedom*, which is also undesirable. The separability style rule disallows such situations.

Specifying Moore Machines As the clock speeds of busses gets faster and faster, almost all bus interface protocols assume a Moore machine timing; namely, it is expected that the interfacing agent needs at least one cycle to respond to its inputs. The output separability rule can be thought of as a result of modeling the agents as Moore machines. For example, machine A has an input in_A and outputs o_A and r_A and is specified a property that breaks the separability style rule: $\neg in_A \vee o_A \wedge r_A$. This can be interpreted as whenever in_A is true, machine A must react immediately and assert its outputs o_A and r_A true. But since machine A is a Moore machine, this is an unreasonable specification.

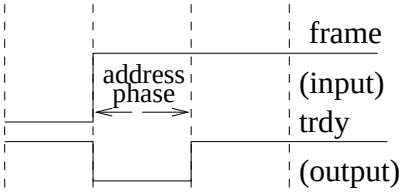


Figure 2.5: *trdy* behavior during the address phase

Mutual Exclusion The example of mutual exclusion is interesting because although it is frequently a desired characteristic of a bus, "only one agent can be driving the bus at a time", the style rule disallows it as a property as outlined in the "Removing Illusory Freedom" paragraph. Mutual exclusion is not a specification that can be implemented independently by several agents. Instead, it is an emergent characteristic that is implied by properties that can be specified independently for the agents. For example, in PCI, only the agent that is the current *master* can drive certain bus signals; this can be specified as a separable property. The arbiter in the system ensures that only one agent is the master at any time, and this is also a separable property. Together, these separable properties ensure the non-separable mutual exclusion characteristic of the bus. In fact, it is this work's conjecture that most high-level, non-separable characteristics can be implemented by low-level, separable properties.

An Un-Implementable PCI 2.2 Requirement Interestingly, there is an official PCI 2.2 specification [**?**] requirement which does not satisfy this separability rule, and is consequently unimplementable as stated. However, there is an equivalent, implementable requirement to replace it.

The requirement, which is in section 3.2.4 of the official PCI documentation, states that the signal *trdy* must use the address phase as a turnaround cycle. The "address phase" is when the signal *frame* is asserted, when it was de-asserted in the previous cycle (Figure 2.5), and a "turnaround cycle" for a bus signal is a cycle where no agent is allowed to drive that signal. Thus, the requirement translates to "if *frame* just became asserted in this state, do not drive *trdy*." The problem is *frame* and *trdy* can be driven by different agents, and so both must decide simultaneously how to meet

the requirement *together*. And a Moore machine agent cannot react (via the value of *trdy*) to its input (in this case, *frame*) in the same cycle. Thus, this requirement cannot be implemented in an agent as stated in the standard. However, this requirement can be stated in a different way with the same intended effect. The requirement should be "no agent may drive *trdy* if *frame* and *irdy* were both deasserted in the previous state.[2]" This obeys the output separability rule and it will enforce the desired property that all agents not drive *trdy* in the address phase.

2.5.2 Second Rule : Isolating Current Variables in a Property

If the property spans more than one time frame (i.e. involves at least the previous state) the property must be written in the form "past_conditions → current_state". Thus, all multi-state properties must be written as implications, and all prior states variables must be in the antecedent and current state variables must be in the consequent.

Example : ``Only when IRDY# is asserted can FRAME# be deasserted.''
Correct : $prev(frame) \rightarrow frame \vee irdy$
Incorrect : $prev(frame) \wedge \neg frame \rightarrow irdy$

Interestingly, this temporal implication is not the most straightforward way to formalize a protocol property. A more intuitive form is a *causal* implication, "*cause* → *effect*". For example, the incorrect version of the above property, $prev(frame) \wedge \neg frame \rightarrow irdy$, is a causal implication and is closer in form to the English version ("if FRAME# is deasserted, then IRDY# must be asserted.") The properties are written in the more awkward temporal implication form because it has advantages in debugging and in input generation. It is noted, though, that unlike the first rule, this rule is not required for the implementability of the specification.

Writing the properties as an implication with the past expression as the antecedent and the current expression as the consequent is a separation of functionality. In essence, the past history, when it satisfies the antecedent expression, requires the current consequent expression to be true; otherwise, the property is not "activated" and the interface signals do not have to obey the consequent

[2] A note to PCI experts: for an address phase following a back-to-back transaction, which won't have *frame* and *irdy* deasserted in the previous state, other properties ensure that there is a turnaround cycle for *trdy*.

in the current cycle. In this way, the temporal implication separates the *activating* logic and the *constraining* logic.

This separation makes contradictions in the properties easier to spot, as demonstrated with an example in section 3.2. Contradictions are a common problem with multiple-property specifications as two or more properties may unexpectedly place conflicting requirements on signals under certain conditions. In identifying the properties in conflict, the separation of the activating and constraining logic is a necessary step in the process. Thus, having the properties in the separated form to begin with clarifies the logic structure and leads to a specification that is easier to debug.

The "activating-constraining" separation is also the key insight that allows for efficient input generation, which is discussed in the "specification use" chapter (chapter 6). In a nutshell, the constraining logic is all that is directly needed for the input generation, and its isolation results in a dramatically smaller expression for manipulation.

This isolation concept also simplifies the first property style rule, the "property separability" rule. Since the constraining part is isolated in the form of the consequent, the rule requires only that the consequent is purely an expression of one component's outputs. Because of this simplification, compliance with the first rule can be easily checked.

Chapter 3

Verifying a Specification

3.1 Introduction

A very important practical question is how to verify a formal specification. A specification can be incorrect in a few ways. On one hand, it can be too restrictive. A catastrophic case of this is when a specification is contradictory for certain states, and consequently, it is impossible for any implementation to satisfy the specification for these states. Another problem caused by over-restriction is when a specification unintentionally disallows certain actions that are meant to be correct. On the other end of extreme, the specification can be under-restrictive. In this scenario, the specification allows certain behavior that should have been labeled illegal because it is incomplete and has "holes" in it.

To uncover these problems, we introduce techniques that work directly on the specification *without requiring any implementations or explicit state machine representations to be written.* This feature is significant for a variety of reasons. For one, designing implementations just to verify a protocol is both labor and time-inefficient. However, in practice, protocol designers must first implement models that interact using the designed protocol in order to verify the protocol. These models are then simulated or model checked so that the behavior can be compared to the specification and checked for deadlock or missing data phases. The method introduced here does not

require the time-consuming implementation process and instead checks the specification directly. Second, the main reason why the properties-based methodology has not been widely adopted, despite the positives outlined in the previous chapters, is because it was generally thought that a properties-based specification cannot be verified without an explicit state machine representation. The methods described in this chapter strengthens the argument for properties-based specifications by introducing a methodology that allows the properties to be verified directly. Third, some practitioners do use properties to check outputs and constraint inputs but do not currently verify these properties. These property verification techniques can lead to improved output checking and input generation.

To verify the properties, techniques based on model checking are used. Essentially, a model checking tool is used to exhaustively search the state space that conforms to the specification. This way, the user can check whether what is allowed by the specification is exactly what is intended to be allowed.

Because of its exhaustive search capability, a model checker in general is very effective at finding problems, but it is severely limited by the "state explosion" problem. For many designs, the number of states to be manipulated by the model checker, the *reachable states*, is unmanageably large. Consequently, descriptions that can be model-checked are limited to relatively small, low-complexity designs. There have been various approaches to this problem aimed at different points of the model checking process. Some have tried to resolve the problem by *abstraction* [**?**, **?**, **?**] where irrelevant information is abstracted away from the state space, and consequently, the number of reachable states is reduced. Another approach is *symbolic model checking* [**?**, **?**, **?**] where a large number of states may be efficiently represented by a small data structure, a binary decision diagram (BDD) [**?**]. In this approach, the smallness of the BDD compared to the large set of states it represents is instrumental in reducing memory usage. However, this compression efficiency is highly dependent on subtle characteristics of the system to be verified. In some cases, explicit state enumeration, which does not use this intermediate representation, outperforms the BDD-based method [**?**].

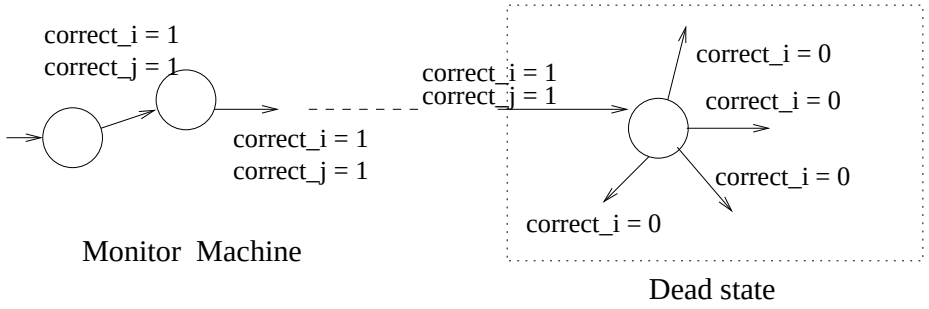


Figure 3.1: A dead state for agent *i*

Fortunately, both example specifications used in this thesis did not cause the "state explosion" problem with available BDD-based model checkers. Of these, Cadence's SMV [**?**], worked particularly well and was used extensively with this project. It will take a monitor-style specification and answer queries about the implicitly defined state graph. The debugging methods require these queries to be written in the branching-time temporal logic CTL[**?**]. It must be emphasized that the *specification* is written without temporal logic such as LTL or CTL, but the properties *to check* the specification are written in CTL.

This chapter will describe the verification methods developed for monitor-style specifications and the type of problems found for each method. Some were translation (from English to formal) errors due to misinterpretation of the specification, which are significant because they support the claim that an informal specification is prone to ambiguity and misunderstanding. However more importantly, some inherent problems in both the official PCI standard and a preliminary version of the Itanium™ bus protocol were discovered. (Details of these problems can be found in chapters 4 and 5 respectively.) These previously un-reported bugs and issues further stress the importance of using a formal specification style to develop and review a protocol.

3.2 Dead State Check

Dead states arise due to contradictions in the specification. For example, if one property for agent *i* requires p to be true in the current state and another requires $\neg p$ to be true in the same state,

there is a dead state. Defining dead states precisely, requires defining a few other concepts first. A transition in the monitor specification machine from a state is said to be *correct for agent i* if the monitor asserts $correct_i$ during the transition. A monitor state *s* is *correctly reachable*, if there exists a sequence of agent outputs that causes the monitor state to enter *s* from the initial state, while $correct_i$ for *all i* are continuously asserted (i.e. all agents have obeyed the specification up until state *s*). A *dead state for agent i* is a correctly reachable monitor state that has no correct exiting transitions for agent *i*; for all outgoing transitions of the dead state, $correct_i$ is false. Intuitively, a good specification should have no correctly reachable dead states because then, all possible agent outputs are incorrect according to the specification. (Others have observed the importance of dead states [**?**].)

To ensure the absence of a dead state in a monitor specification, a certain characteristic needs to hold for all the agents in the specification: "for every state in the monitor where no properties for any agent have been broken so far, there must exist at least one next state where all of the properties for the particular agent hold." This characteristic can be checked easily using a CTL model checker with the formula, for a particular agent *i*,

$$\neg E[(correct_0 \wedge ... \wedge correct_n)\ U\ ((correct_0 \wedge ... \wedge correct_n) \wedge (AX \neg correct_i))]$$

("There is no path where all the agents have been correct so far but agent *i* has no possible correct next state"). If there are any contradictions in the specification, the model checker for this property returns a counterexample indicating a dead state.

It must be emphasized that the conditional in the formula, "all the agents have been correct so far", is essential. A contradiction among properties *is allowed* if any properties have been violated earlier in the execution path. The absence of contradictions only has to hold for the set of correctly reachable states. It is not necessary nor desirable to guarantee satisfiability for every possible subset of properties. (In fact, if this were the case, the properties would be much more complicated, and the ease of writing and reading a specification would be greatly reduced.) This factor is why a reachability analysis (which determines the correctly reachable states) is required, and a plain-vanilla satisfiability check is not the right approach for finding contradictions.

The following is an incorrect logical translation of some PCI requirements, a mistake actually made by the author. The dead state check found a dead state which resulted from the conflicting properties,

$$prev(address\_phase) \rightarrow \neg trdy$$

$$prev(trdy) \rightarrow trdy \vee (irdy \wedge (stop \vee trdy)) \vee prev(irdy \wedge (stop \vee trdy))$$

$$prev(trdy) \rightarrow (prev(stop) \leftrightarrow stop)$$

The contradiction is not obvious from the expression above but if it is re-written to obey the *Isolate Current Variables* rule from section 2.5.2 (see below), and the values of the state variables in the dead state are known, it is possible to see that there is no legal next state when $prev(address\_phase \wedge \neg irdy \wedge trdy \wedge \neg stop)$ holds because $\neg trdy \wedge (trdy \vee (stop \wedge irdy)) \wedge \neg stop$ is unsatisfiable.

$$prev(address\_phase) \rightarrow \neg trdy$$

$$prev(\neg irdy \wedge trdy) \rightarrow trdy \vee (stop \wedge irdy)$$

$$prev(trdy \wedge \neg stop) \rightarrow \neg stop$$

$$prev(trdy \wedge stop) \rightarrow stop$$

Since the dead state check returns the dead state as "..., $address\_phase = true, irdy = false, trdy = true, stop = false$, ...", these variable assignments can be used to see which properties are in effect, by plugging these values into the left-hand-side of the implications (the antecedents). In this case, the first three properties are in effect and their consequents form the contradiction. This process is effective because the properties follow the *Isolate Current Variables* rule and the model checker can find a dead state as a counter-example to the above CTL formula. One advantage this check has over the other checks is its simplicity. No creativity or expertise is required; only the CTL formula given above and a model checker are needed.

There are a few limitations to this simple check. First, it lacks the ability to find profound bugs, and its usefulness is more akin to a spell checker than a full debugger. Specifically, the check proved to be more effective in catching errors introduced in the monitor writing stage rather than problems inherent with the protocol. Second, although the check can return a state description of

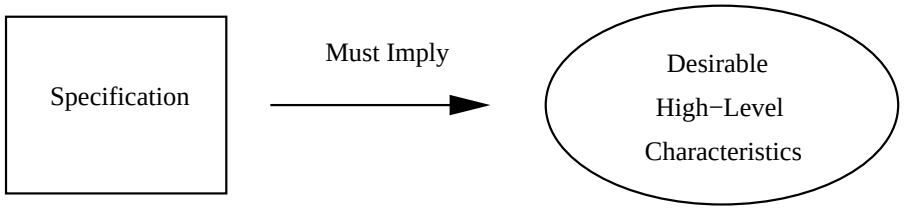
the dead state, it is unable to pinpoint which properties are in conflict. The problem is exacerbated by the fact that an arbitrary number of properties can contribute to the contradiction. The search complexity for the offending properties is exponential in the number of activated properties. Third, the check assumes that the interface agents are Moore machines. However, this assumption is reasonable in today's high clock speed designs, and Bryant, Chauhan, Clarke, and Goel have researched inconsistencies for combinational Mealy circuits [**?**].

Another similar check tests for under-restriction in the specification. It is reasonable to assume that in all states, at least one property is in effect. The check searches for correctly reachable states where *all* possible outputs for agent *i* are correct according to the monitor. The monitor can be checked for this property with the CTL formula, $\neg E[(correct_0 \wedge correct_1 \wedge ... \wedge correct_n)\ U\ ((correct_0 \wedge ... \wedge correct_n) \wedge (AX\, correct_i))]$ ("There is no scenario where all the agents have been correct so far and all possible next states for agent *i* are correct"). Although this check turned up no bugs in the PCI monitor specification, it is still a worthwhile check. Like the dead state check, this check requires no creativity or extra work.

3.3 Characteristic Check

A more powerful debugging method is checking for specific properties, or *characteristics* in the specification. These characteristics are CTL statements about agent events. If the specification is too constricting, certain agent actions which are intended to be possible will not be allowed by the specification (*over-restriction*). Thus, CTL statements (the characteristics) can be written to model-check that the expected agent actions are allowed to happen within the state space deemed legal by the specification. With this check, characteristics such as "data can be transferred in a retried transaction" and "an observer counter can reach a high value" are checked against the specification, which is the model that the model checker is checking in this case.

The characteristics can also check for *under-restriction* and incompleteness of the specification. If the specification is "too loose", actions that are intended to be illegal may be deemed legal



previous(frame) IMPLIES (frame | irdy)

CTL, LTL
"The bus must eventually reset"
"Signaling definitions must be disjoint"

With the characteristics check, rich properties about the protocol can be guaranteed.

Figure 3.2: The Characteristics Check

by the specification. Thus, characteristics such as “data transfer does not immediately happen during a master aborted termination” or “data cannot be transferred in a *disconnect without data* termination” check that the specification does not allow these “bad” events. Again, with the model checking framework, the specification is the model and the characteristics are used to check the model.

With a CTL based model checker, the characteristics can be checked against the specification properties in the following form. If C is the characteristic to be checked and $correct_i$ is true when all properties specifying agent i are true, then the CTL checking expression is:

$$\neg E[(correct_0 \wedge ... \wedge correct_n)\ U\ ((correct_0 \wedge ... \wedge correct_n) \wedge (\neg C))]$$

(“There is no scenario where all the agents have been correct so far but, in the current state, characteristic C does not hold.”) If this CTL formula is violated, the model checker returns a counterexample indicating how the specification does not obey the characteristic.

This debugging method furthers the case for formally specifying protocols because it found several bugs in the official PCI protocol standard and issues in a preliminary version of the Intel® Itanium™ bus protocol. In this way, it is more powerful than the dead state check which only found superficial problems with the specification. Also, although the *specification* properties are simple, bounded, linear time properties, the *checking* characteristics are more complex, unbounded, CTL

formulas. *This allows the specification to be synthesizable and yet guarantee rich properties* (Figure 3.2). Thus, this method is one way to express and specify rich properties without resorting to complex and cumbersome constructs. In fact, as long as the agent outputs comply with the protocol properties, these checks guarantee that the outputs also comply with higher level characteristics (that are true of the specification) such as "data is eventually transferred" or "an agent can only signal a single termination mode at one time".

One drawback of this characteristic checking is that a user must come up with the characteristic statements because they cannot be automatically deduced from the specification. It is also subject to false error reports when the characteristics themselves are incorrect. However, from the experience of verifying two protocols, it can be surmised that many of these characteristics are common among multiple interface protocols and thus can be reused. Consequently, the effort in developing effective, incisive, and correct characteristics to find problems in the specification, can be amortized over a number of interface specifications.

Example characteristics and the protocol bugs found with them can be found in chapters 4 and 5.

3.4 Generator-Based Check

Using a constraint solver, a *generator* can be created from a specification. This generator will output correct signals on every cycle by solving the properties in the specification. (The details of this generation scheme can be found in chapter 6.) If several generators, representing the different agents in the specification, are hooked up together, they will interact using the specified protocol. This way, example traces of the protocol can be produced *without the need to design state machines.* Waveforms like those that illustrate English protocol specifications (Figure 3.3) can be automatically generated from the specification. Not only is this useful for visualization of the specification, but by manually scanning through the traces, problems with the specification can be found. In fact, several serious problems were found with the PCI specification that had escaped

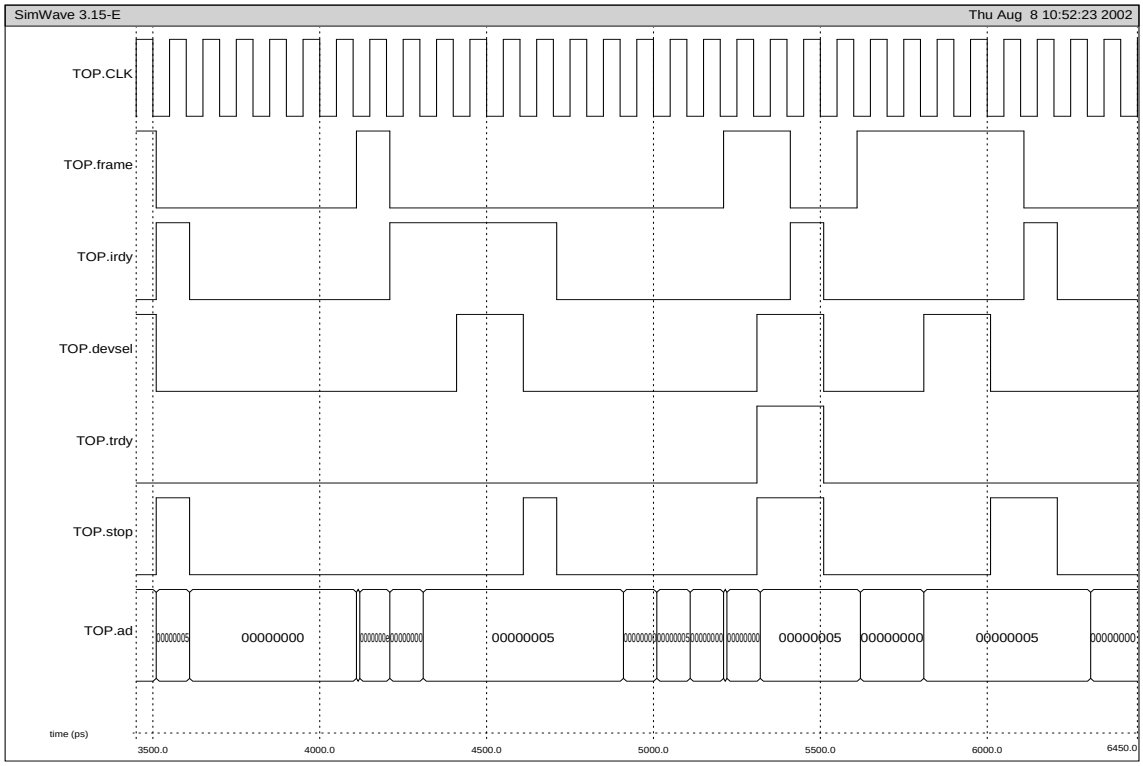


Figure 3.3: An automatically generated PCI protocol waveform

the two model-checking based methods listed above. This fact is an testament to the advantage of this method; a user does not have to dream up various illegal behavior that the specification may allow because the generators will autonomously exhibit the behavior during a simulation run. The creativity requirement is a major hindrance in model checker based techniques as it is very difficult to come up with an exhaustive list of "bad" scenarios to check for.

For the PCI specification, it was found that this generator-based technique is useful for finding missing properties. The following bug in the specification was discovered by simulating two PCI generators against each other. In a nutshell, a signal that should be kept steady during a communication handshake was allowed to change values repeatedly before the handshake was completed. Specifically, while the signal *frame* was held true and the reciprocal agent (the target) hadn't responded, the signal *irdy* was bouncing back and forth between true and false. A correct specification would not allow this and force *irdy* to be stable until the reciprocal agent responds, but there was a missing case in the specification. Fortunately, this problem was easily fixed by adding a new property that covered the "hole" in the description. Model checking techniques did not find this bug because it did not occur to the author to check for this particular scenario. This simulation-based technique proved adept at demonstrating what an incomplete specification would allow for agent behavior.

3.5 Vacuous Property

Another check that can be done with very little effort is the *Vacuous Property* check. With this test, a model checker is used to verify that within the correctly reachable state space, the antecedents of all the properties will each become true at some point during execution. If there is a property where the antecedent is always false, the property is never "fired" and is therefore vacuous. Since it is meaningless to have a property that is never in force, it can be safely assumed that the property is not stated correctly and requires modification. As with the dead state check, this method is an excellent "spell checker" and a few minor errors were discovered with this check.

3.6 The Receptiveness Proof

3.6.1 Introduction

In this section, it is demonstrated how the *implementability* and a stronger property, *receptiveness*, can be proved for specification. This proof is powerful for a number of practical reasons. Because there is no explicit state machine representation of the specification, it is unclear, at first glance, whether the collection of properties amount to a reasonable state machine. In the worst case, only a state machine that does nothing may satisfy all the properties. Or, perhaps, only some of the behavior modes can be implemented as a state machine, and the other modes described by the properties can not be, in which case, the specification purports to have flexibility that it does not really have. This theorem guarantees that all the choices offered by the specification can actually be designed into a state machine (receptiveness). Furthermore, the theorem guarantees an implementation that will behave correctly no matter what the environment does as long as the latter observes the interface specification (implementability). This guarantee is particularly important for standard interfaces where implementations are created independently by different parties and are expected to be integrated seamlessly.

Because of the theorem, it is easy to check whether a specification is implementable and receptive; the specification properties must be separable (as defined in section 2.5.1) and there must be no dead states (as defined in section 3.2) in the specification. The proof of this will be described in the following sections. The first section, “Setup for the Proof”, defines the assumptions and the vocabulary. The next section will state the theorem, the corollaries, the proof sketch, and a more precise definition of receptiveness.

3.6.2 Setup for the Proof

First, it is noted that a specification monitor is a *Mealy machine*. A Mealy machine has its inputs *and* its outputs on its edges; the outputs are not associated with a state as with Moore machines.

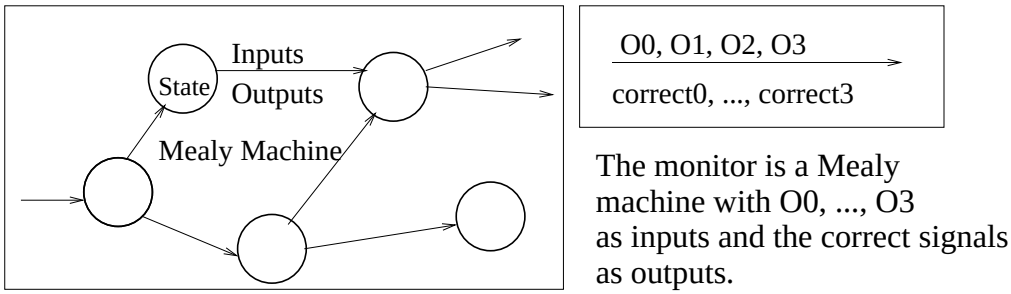


Figure 3.4: A Mealy Machine

A monitor is a Mealy machine because in order to determine the output $correct_i$ values, a combinational function on its inputs, which are the observed agent output signals, is sufficient (Figure 2.2, 3.4). Therefore, once the agents have set their outputs for a particular cycle, the monitor can instantaneously determine whether the outputs are correct or not.

Second, one can view the system of agents as "one agent (the agent of interest) and its environment (the other agents)." Using the same monitor, we now have outputs $correct_{Me}$ for the former and a $correct_{Env}$ for the latter; $correct_{Me}$ can be a particular $correct_j$ and $correct_{Env}$ would be a logical conjunction of $correct_i$ for $i \neq j$ (Figure 3.4). This particular setup can be viewed as a game between the agent of interest, *Me*, and the environment, *Env*. *Me* and *Env* output signals in a locked synchrony and do not alternate driving an output. As soon as *Env* breaks a specification rule ($correct_{Env}$ becomes false), *Me* has "won" and the monitor's $correct_{Me}$ value will remain true regardless of *Me*'s outputs. This is a reasonable restriction because an agent should not be required to uphold the specification if the environment has fed it illegal inputs. It will be proven that if two requirements hold for the specification of the system, it is guaranteed that a Moore machine *K* exists where no matter what the environment outputs to it, *K* will always output signals that will keep $correct_{Me}$ true; therefore, *K implements* the specification. With such a *K*, the environment will never be able to force *K* to output an illegal sequence.

The first of the two requirements is that there are no dead states (section 3.2) for agent *Me* in the monitor. For every correctly reachable state of the monitor, there is at least one transition out of

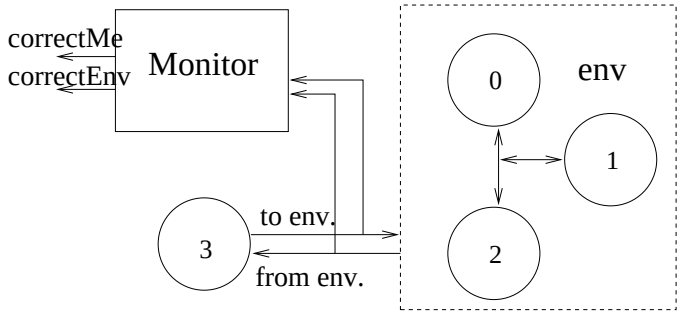


Figure 3.5: One Agent and the Environment

that state with the $correct_{Me}$ on the edge as true. The second requirement on the specification monitor, the output separability rule (section 2.5.1), is restated here as "the function which determines $correct_i$ must only be a function of the current state of the monitor machine and the current output of agent *i*, and *not the current outputs of any other agent.*" The output values of the other agents are used to determine the next state of the monitor machine but not the value of $correct_i$. With these two requirements, receptiveness and implementability can be shown.

3.6.3 Theorem and Corollaries

Theorem 1 *If a Mealy machine monitor, M, which obeys the following requirements exists for some specification, then a Moore machine implementation for a single agent, Me, is guaranteed to exist. The restrictions are,*

1. *The monitor must not have any dead states for agent Me.*
2. *The monitor must observe the output separability rule.*

And it is assumed that once the environment breaches the specification, $correct_{Me}$ *is infinitely true.*

Proof Sketch: The theorem is proved by constructing the Moore machine implementation of *Me*. The following is an outline of the construction algorithm, and the full proof can be found in Appendix A.

1. Start with the initial state.

2. To determine an output for this state, an output vector that satisfies $correct_{Me}$ is picked (from

potentially many choices) regardless of its current inputs. This is possible because $correct_{Me}$ is a function of the current state and outputs of *Me* only; *Me*'s current inputs cannot affect the correctness of its current outputs (restriction 2). Furthermore, the existence of at least one output vector that satisfies $correct_{Me}$ is guaranteed (restriction 1).

3. A next state is defined by the output chosen in the previous step and an input vector. A set of next states is defined by considering all possible input vectors. For all the states in this set which have yet to be visited, repeat step 2.

Interesting Note Although the states in this constructed implementation have an one-to-one correspondence to the states in the monitor, this need not be true for an implementation to satisfy the specification. In fact, in most implementations, there will be several implementation states which correspond to a single monitor state. Namely, an agent may cause the monitor machine to visit a state twice but the internal state in the agent machine is different for the two cases. This is why a monitor cannot be used to quantify simulation coverage precisely although it can be used as an approximation. A visited stated in the monitor does not indicate that testing does not have to cover this state again.

If one views this system once again as a multiple agent model (Figure 2.1) as opposed to one agent and an environment, an interesting corollary can be deduced from Theorem 1. The following guarantees that the *entire interface system* can be implemented as long as certain assumptions hold. This is in comparison to Theorem 1 which only guarantees an implementation for one agent and does not reason about the other agents.

Corollary 1 : A Set of Implementations Exist for a Specification *If a Mealy machine monitor, M, which obeys the following restriction exists for some specification, then a set of Moore machines which implement the specification is guaranteed to exist. The restrictions are,*

1. *The monitor has no dead states for all agents in the specification.*
2. *The monitor observes the output separability rule.*

An implementation can always choose the left branch in the specification to avoid the dead state. Receptiveness disallows such illusory freedom in the specification.

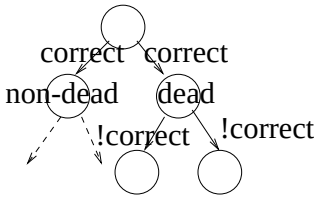


Figure 3.6: How even with dead states, an implementation can exist

Proof Sketch Apply theorem 1 to each agent in the specification and the theorem guarantees a correct implementation for every agent. Since the theorem guarantees specification compliance independent of the inputs to the agent (assumption 2), the agents can be implemented independently. Furthermore, Theorem 1 guarantees their correctness when composed together; because an environment can never trick an agent into erroneous behavior, composing the agents (i.e. the other agents form the environment) cannot lead any agent to violate the specification. Note that the output separability requirement is so strong that "assume-guarantee" reasoning is not required. One does not have to assume that the rest of the agents are correct; a particular agent is guaranteed to be correct irrespective of the correctness of the other agents.

Receptiveness A monitor specification is defined to be *receptive* [**?**] if for every correctly reachable state in the monitor, there exist agent implementations, when connected to each other and to the monitor, can cause the monitor to reach that state.[1] Receptiveness ensures that there is no illusory freedom in the specification. The following is an example of illusory or vacuous freedom.

$$prev(a) \rightarrow out_0 \vee out_1$$

$$prev(out_1) \rightarrow \neg c$$

$$prev(out_1) \rightarrow c$$

out_0 and out_1 are outputs of one agent and so the agent can always choose to assert out_0 instead of out_1 to avoid the inevitable error state caused by asserting out_1 (because of the last two properties which cause a contradiction on signal c in the next state). Thus, even with a dead state in the specification, there exists an implementation; this example illustrates how the absence of dead

[1] Ed Clarke has also recognized the relevance of receptiveness to bus specifications, but proposes using model checking algorithms that can check the property directly.

states is not a necessary (but a sufficient) condition for Theorem 1. Receptiveness is a tougher condition to satisfy than implementability, and ensures that there is no illusory freedom in the specification such as "$\vee$ out_1" in the first property. Every correctly reachable state must have a correct next state in order for receptiveness to hold for a specification.

Corollary 2 : Receptiveness *A specification is receptive (i.e. for every correctly reachable state in the monitor, there exists a set of agent implementations that can take the monitor to that state with correct outputs) if*

1. *The monitor does not have any dead states for all agents in the specification.*
2. *The monitor observes the output separability rule.*

Proof Sketch : State *s* is a correctly reachable monitor state and the sequence of correct agent *i* outputs $\{O_0^i, O_1^i, ..., O_n^i\}$ which lead to state *s* is known for $\forall i$ in Agents. Thus, we have $\{O_0^0, O_1^0, ..., O_n^0\}$, $\{O_0^1, O_1^1, ..., O_n^1\}$, ..., $\{O_0^m, O_1^m, ..., O_n^m\}$ which together lead the monitor to state *s*. Agents can be individually constructed so that they each output their respective sequences from above. Therefore, there exists a set of agents that can take the monitor to state s; it remains to be shown that these agents implement the specification, namely, that their outputs keep the $correct_i$'s true. Up till state *s*, it is obvious that the agents are correct implementations because the output sequences were chosen along the edges where all $correct_i$'s are true. Such a sequence exists because *s* is correctly reachable. As for after state *s*, there exists a next state s' such that the transition from *s* to s' is *correct*, because the monitor is dead-state free for all agents (assumption 1). The outputs along this transition can be used for the agent implementations. Inductively, it can be shown that at each step, a correct output can be independently chosen for all agents because of the two assumptions. Therefore, agents that implement the specification and take the monitor to state *s* can be created for any correctly reachable state *s*; the specification is receptive.

Chapter 4

PCI Results

4.1 The PCI Protocol

Inside millions of personal computers, the PCI bus protocol is being used for communicating video, graphics, and networking data to the processor. The protocol is used to interconnect peripheral add-in boards such as audio cards and graphic cards, and controller components such as LAN and SCSI controllers, with the processor/memory system. There are several configurations for the bus architecture: 32-bit or 64-bit address/data path at 33MHz, or 32-bit or 64-bit address/data path at 66Mhz. There are 49 signals for the master and 47 signals for the target. The *PCI Local Bus Specification, Rev 2.2* (called the "Official PCI Specification" from now on), includes the protocol, electrical, mechanical, and configuration specification, but the formal specification written here covers mainly the protocol definitions.

4.2 Formal Specification of the PCI Protocol

The formal specification covers the core protocol described from section 3.1 to section 3.6 of the Official PCI Specification. Therefore, bus bridges and 64 bit extensions are not covered. A large

[0]This chapter is based on material first published in *Proceedings of the Third International Conference of Formal Methods in Computer-Aided Design* (FMCAD 00), 2000 [**?**].

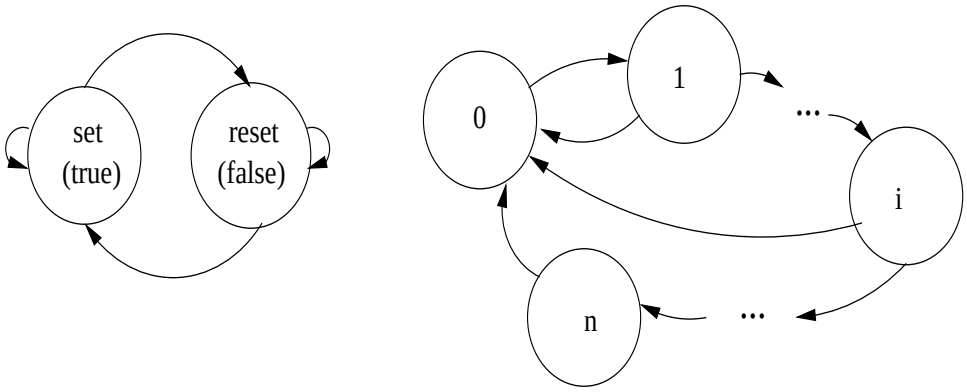
frame	master	Assertion indicates the beginning of a transaction. Deassertion indicates that the upcoming data phase is the final one.
irdy	master	Assertion indicates that the master is ready to receive/send data.
devsel	target	Used by a target to claim the transaction initiated by a master. If the address requested by the current master falls in the range of the target's address, the target must respond by asserting this signal.
trdy	target	Assertion indicates that the target is ready to receive/send data.
stop	target	Assertion indicates that the target cannot complete the transaction.

Table 4.1: The Control Signals in PCI

subset of the 49 required signals were chosen, and their behavior is specified. The required signals break down into the following categories: 6 are control signals, 37 are address and data signals, 2 are for error reporting, 2 are for bus arbitration (and thus are only needed for masters), and there is one clock pin and one reset pin. For the formal specification, all of the control signals, arbitration signals, clock, and reset are fully specified. The core control signals and their roles are described in Table 4.1. The error reporting signals, PERR# and SERR#, and the parity signal, PAR, are not included in the specification although adding them is not very difficult. As for the address and data signals, they too are specified although they are represented in an abstract form to reduce the complexity (i.e. the number of signals) of the specification for both model checking and readability purposes.

The resulting specification consists of 83 rules and 14 small state machines (7 counters and 7 set-reset machines, see Figure 4.1). For illustration, Figure 4.2 exhibits properties from the specification; note the *previous implies current* form and how the consequent is an expression purely of output variables for agent *i*.

The description file has 280 lines excluding comments, and these correspond to 70 pages in the Official PCI Specification underscoring the conciseness of formal specifications. Also, because of the high readability and re-usability of this specification, important advantages of this specification methodology, it has been used by several groups in their respective projects: Clarke *et al.* [**?**], Govindaraju *et al.* [**?**], and Aloul *et al.* [**?**]. The specification was originally written in a SMV-like



Only two types of state machines are needed to specify the PCI protocol: a set-reset (left) and a counter (right).

Figure 4.1: Two state machine types

language, but was later translated using a compiler-based tool into an executable and synthesizable Verilog description.

This specification was fully debugged using the different checks, and several interesting bugs, described in the following section, were found with the protocol. Currently, there are no dead states in the specification, and all of the 114 characteristics checked do hold for the description. The model checking of the specification was done using Cadence SMV [**?**] on a Pentium Pro system with 128MB of memory where all the model checking runs took less than 5 minutes. Because of the limited complexity of an interface protocol, model checking proved to be relatively effortless for the specification.

4.3 Formal Verification of the PCI Protocol and the Bugs Found

4.3.1 Dead State Check

As explained in section 3.2, the dead state check is more effective in catching errors introduced in the monitor writing stage rather than serious problems in the protocol itself. Specifically, the dead state check pinpointed typographical errors by the monitor specification writer, misinterpretations by the monitor writer due to the ambiguous wording in the protocol text, and exceptions to general

```
/* These are properties for agent i on the bus. */

/* If i is the target and has asserted trdy but the data phase has not
 * completed yet, then in the next cycle, i must keep trdy asserted by
 * driving the trdy bus (trdy_e = 1) and by outputting the
 * ‘‘asserted’’ value (trdy_o = 1).  */

previous((target_is = i) & trdy_e & trdy_o & !dphase_done)
  -> trdy_e & trdy_o

/* If i is the master, then it must deassert irdy on the clock
 * following the completion of the final data phase.  Deassertion of
 * irdy can either be done by not driving the irdy bus (irdy_e = 0) or
 * by driving the bus in the deasserted state (irdy_e = 1 & irdy_o =
 * 0) */

previous((master_is = i) & final_dphase_done)
  -> !irdy_e | (irdy_e & !irdy_o)
```

Figure 4.2: Properties from the formal PCI specification

rules not mentioned by the official specification. Because this check proved to be effective in finding the exact intent of the requirement, it proved indispensable for making the constraints precise. Four bugs in the formal specification which were due to misinterpretation of arguably ambiguous wording in the official documentation were found by dead state checking. Furthermore, the test aids the specification writer in realizing the boundary cases for general rules by demonstrating how a contradiction can occur in special cases. Thus, the dead state check helps the specification writer identify exceptions to the too generally-stated rules, *which are not mentioned by the official document.* The dead state check discovered six such cases where the monitor writer needed to refine the constraints to account for the special cases. For example, section 3.2.4 of the specification says the wire *irdy* must not be driven during the address phase of the transaction. The specification neglects to mention that the rule does not have to hold when the address phase marks the beginning of a back-to-back transaction (where, in this case, the *irdy* wire does need to be driven). Because of

this incorrect "blanket statement" rule, a contradiction existed for the address phase of a back-to-back transaction ("should the *irdy* wire be driven or not driven?"), and this problem was discovered automatically by the dead state check.

Another class of problems found by the dead state check consists of ambiguously worded requirements which are misinterpreted by the formal specification writer and thus contradict other requirements in the specification. The following example is from section 3.3.3.1: "IRDY# must remain asserted for at least one clock after FRAME# is de-asserted" which seemingly translates to the property,

1. $prev(prev(frame)) \wedge prev(\neg frame) \rightarrow irdy$

However, in section 3.3.3.2.1, it is stated that "the master must de-assert IRDY# the clock after the completion of the last data phase."

$last\_data\_phase = \neg frame \wedge irdy \wedge (trdy \vee stop)$

$prev(last\_data\_phase) \rightarrow \neg irdy$

$\Rightarrow$ 2. $prev(\neg frame \wedge irdy \wedge (trdy \vee stop)) \rightarrow \neg irdy$

The conjunction of properties 1 and 2 causes a conflicting requirement on *irdy* in the correctly reachable state where both antecedents are true: $prev(\neg frame \wedge irdy \wedge (trdy \vee stop) \wedge prev(frame))$. In the next state, the first property states that *irdy* must be true and the second, *irdy* must be false. It was concluded from guessing at the intention of the requirement that the first rule was misinterpreted and the correct interpretation of it is $prev(frame) \rightarrow irdy \vee frame$ which admittedly is puzzling because this property doesn't require the "one clock after" part of the requirement.

4.3.2 Characteristic Check

After the dead state check, the characteristic check was used to further solidify the specification. 114 characteristics were written in CTL and checked against the properties-based PCI specification. This checking method found sixteen bugs in the monitor which resulted from errors in the formal specification writing process, but more importantly, seven bugs in the official standard were found by this method. For finding actual problems in the official specification, this test proved to be more

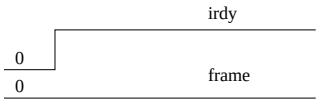


Figure 4.3: A non-Idle State

effective than the simple dead state check. Here are some characteristics that exposed problems in the official specification. These or similar characteristics are probably applicable for protocols other than PCI and are considered general system properties that should be checked for.

1. System Must Inevitably Return to the Idle State It is reasonable to assume that the system should always be *able* to reset into the *idle* state; if there are any deadlock states which forbid this from happening, checking for this characteristic should find such a problem. However, it is the stronger property, "the system must always *inevitably* reset and go back to the idle state" which found problems in the PCI protocol. In CTL, the characteristic for the specification is expressed as,

$\neg EG((correct_0 \wedge ... \wedge correct_n) \wedge \neg idle)$

where $idle = \neg irdy \wedge \neg frame$

"There is no path where the system can continuously obey the specification and stay in the non-*idle* state forever." In other words, the specification must always force the interface to reset to *idle* eventually. This characteristic has to hold for the PCI protocol because only when the bus is idle, can a new agent start a transaction. If the bus is never idle because one agent is constantly driving it, this agent has effectively taken over the bus never allowing other agents to use it. To avoid such a situation, the specification must force an agent to eventually relinquish the use of the bus as a master and let the bus state be idle.

There are three ways the current PCI protocol allows an agent to remain in a non-idle state and not relinquish the bus. Essentially in all three cases, *frame* is deasserted while *irdy* is asserted by the agent (Figure 4.3). An agent can keep the bus in this state because of the following. There is a timer counter which counts the number of cycles *frame* has been asserted and when it exceeds a preset value, the specification requires the agent to deassert *frame*. Thus, the protocol intends to limit one agent's use of the bus by observing the assertion of *frame*. The protocol's shortcoming is

in not recognizing that in an $irdy \wedge \neg frame$ state, $\neg frame$ keeps the timer counter deactivated but *irdy* keeps the bus non-idle.

1. From an idle state, the protocol allows an agent to assert *irdy* and remain in this non-idle-bus state ($irdy \wedge \neg frame$) forever. (Figure 4.3)

2. During the data phase of a single data phase transaction, *frame* is deasserted and *irdy* is asserted. If a target doesn't respond with a $trdy$ or a $stop$, the master can remain in this non-idle-bus state forever.

3. During the last data phase of a transaction, *frame* is deasserted and *irdy* is asserted. If a target doesn't respond with a final $trdy$ or a $stop$, the agent can remain in this non-idle-bus state forever.

2. Definitions are Disjoint Protocols allow agents to terminate transactions without successfully completing them. Usually, there are several different types of terminations and an agent asserts and de-asserts different bus signals to indicate which termination type it is executing. For example, in PCI, one termination type, *target_abort* is defined as $target\_abort = \neg devsel \wedge stop$ and another type, a *retry* as $retry = stop \wedge \neg trdy \wedge initial\_data\_phase$ in section 3.3.3.2 of the Official PCI Specification. The other agent involved in the transaction, namely the master agent, must react differently to each target termination type (Table 4.2), and so the ability to identify a termination type uniquely from the signals of the terminating agent is important. We can check whether these terminations are distinct by checking that within the correctly reachable space,

$$\neg(target\_abort \wedge retry)$$

is true. "There is never a state where the specification holds and *target_abort* and *retry* are signaled simultaneously". This check pinpoints a correctly reachable state where $\neg trdy \wedge stop \wedge \neg devsel \wedge initial\_data\_phase$ is true and these signal values are consistent with either *retry* or *target_abort*. Therefore, the protocol allows an agent to signal both termination types simultaneously. This is

Type	Target	Master
Retry	"Termination is requested *before any* data is transferred because the target is busy and temporarily unable to process the transaction (3.3.3.2)"	"Must unconditionally repeat the same request (section 3.3.3.2.2)"
Disconnect With Data	"Termination is requested *after* data was transferred because the target is temporarily unable to continue. Data is transferred on the data phase where Disconnect is signaled. (3.3.3.2)"	"Not required to repeat the transaction (section 3.3.3.2.2)"
Disconnect Without Data	"Termination is requested *after* data was transferred because the target is temporarily unable to continue. Data is not transferred on the data phase where Disconnect is signaled. (3.3.3.2)"	"Not required to repeat the transaction (section 3.3.3.2.2)"
Target-Abort	"An abnormal termination requested because the target will never be able to complete the request. (3.3.3.2)"	"Does not want the master to repeat the request again. (3.3.3.2.1)"

Table 4.2: Target Initiated Termination Types

a problem because a *retry* requires the master to repeat the transaction while *target_abort* forbids it. In fact, with *retry*, the target is signaling that it is busy and *temporarily* unable to process the transaction while with *target_abort*, the target is signaling that the transaction causes a fatal error in the target. These two are very different scenarios, and this protocol bug allows the target to signal both simultaneously. If the PCI protocol had been originally written in a formal form and verified for this characteristic, this ambiguity in the protocol could easily have been found and resolved before the protocol became a public standard.

3. Termination Types Should Not Change During a Single Transaction The third characteristic checks whether termination types can change during one transaction. For example, it checks whether an agent can signal a *target abort* in one clock cycle and then a *retry* in the next clock

cycle before the transaction ends. Checking the property

$$\neg((correct_0 \wedge ... \wedge correct_n) \wedge target\_abort \wedge EX((correct_0 \wedge ... \wedge correct_n) \wedge retry))$$

("there is no correctly reachable state where a target abort has been signaled, and immediately afterwards, retry is signaled") reveals that the PCI protocol does allow this. Again, this is problematic because the two termination types signal very different target scenarios and expected master responses. This un-intended scenario can happen with any combination of target termination modes. The problem specifically is the ambiguity in how, for example, the master agent should be implemented to react to a target which signals a *target abort* (target will not be sending data) and then immediately signals a *disconnect with data* (target will be sending data).

Chapter 5

Itanium Results

The Intel® Itanium™ Processor Bus Protocol is used for all bus architectures centered around the Itanium™ microprocessor. The bus can connect multiple processors, memory controllers, I/O bridges, and custom attachments, and has several interesting features for efficient data bus utilization. Pipelining and deferred transaction capabilities have been designed into the protocol in order to maximize the available bus bandwidth. By pipelining, idling of bus wires can be avoided so all the wires are actively communicating information for different transactions at all times. With deferred transaction completion, if the response agent (i.e. the target or the slave) is not ready to send the requested data, the completion of the transaction can be deferred so that another transaction can use the data bus. These advanced features increase the performance of the bus, but at the same time, they make the problem of formally specifying and then verifying the protocol more challenging. Along with its wide-spread use, this challenge is precisely why this protocol was chosen as the second example in this thesis work. After developing the specification style with PCI, the Itanium™ bus protocol was specified to show that the generalized methodology can be applied to more advanced bus protocols. Despite the added complexity of pipelining, the expressiveness of

[0]This chapter is based on material first published in the *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (CHARME 01), 2001 [**?**].

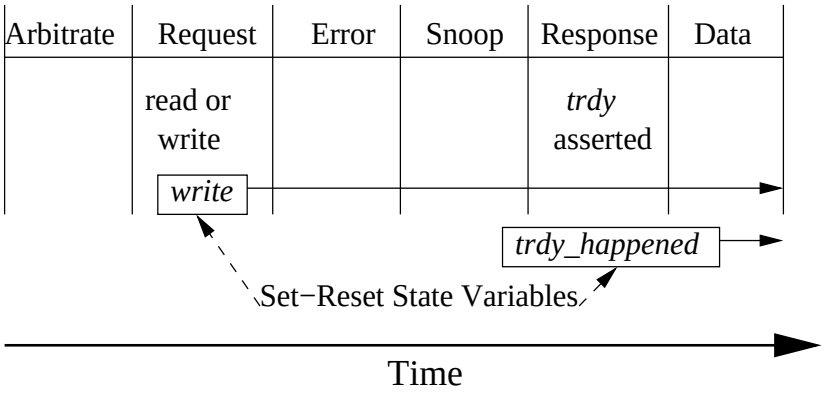
the style proved to not be a problem, and the protocol was specified in the monitor-style using multiple compact properties and small, standard state machines. In fact, the properties-based approach handles pipelining with a very elegant solution. The added complexity was also not a problem for model checker performance, and the verification methods developed with PCI worked well with this protocol too. In total, the time to write and verify the specification only took 2 man-months, which, again, attests to the ease of developing a specification using the methodology.

Additionally, this work with the bus is significant because it demonstrates how writing and checking the stylized specification reveals interesting issues during the development of the protocol. This project was done prior to the official release of the protocol, and the discovered issues were profound enough that the protocol designers re-designed the protocol to correct these issues before the release. The issues ranged from deadlocking of the bus to inadvertently missed data phases. This methodology's ability to verify a protocol before any implementations are built is very valuable in this sense.

5.1 Specification

A core subset of the Itanium™ Processor bus protocol was specified; the request phase, the snoop phase, the response phase, the data phase, and the deferred phase are all covered, but, for example, the arbitration phase is not. Higher level characteristics of deferred transactions, such as the assurance of completion, are not specified or checked because they are probably better treated in a different specification.

The specification for the core subset of the protocol consists of 46 independent constraints which can be replicated eight times for the pipeline depth of eight. To minimize model checking complications, a pipeline depth of two was used for debugging the specification. The description file for a pipeline depth of two, written in Cadence SMV [**?**], is 650 lines long, excluding the variable declarations. The formal specification was formally verified using model-checking based techniques; the current specification has no dead states and all characteristics checked for, hold.



The pipeline stages of the protocol and the state variables for the example rule

Figure 5.1: The pipeline and state variables

Model checking was done using Cadence SMV [**?**] and all characteristics checks were completed within two minutes on a Pentium Pro system with 128Mb of memory.

The following sections illustrate how the the specification style specifies advanced features that were not part of the simpler PCI protocol. There is a description of pipeline specification, the treatment of the protocol's time-unbounded rules, and an explanation of a reaction timing issue.

5.1.1 Pipelining

The protocol has six phases. Transactions must go through the arbitration phase, request phase, error phase, snoop phase, response phase, and data phase. The type of transaction, whether it is a write or a read, is determined during the request phase. An important signaling event, the assertion of *trdy*, happens during the response phase. Thus, the example rule, "*trdy* must be asserted for a write transaction," requires the transaction type (write) to be stored from the request phase until the response phase. Also, the end of the response phase is defined by the assertion of *rs* and so the rule requires that *trdy* be asserted before *rs*.

Consequently, to specify the property, two auxiliary variables are sufficient. First, there is a history variable *write* which becomes true during the request phase if the transaction is a write, and stays true until the transaction is completed. The second history variable, *trdy_happened*, becomes

true when *trdy* is asserted and stays true for the duration of the transaction (Figure 5.1).

Using these variables, the property becomes, "if a transaction is a write, then if *trdy* assertion hasn't happened yet, *rs* cannot be asserted (i.e. the response phase can't complete)."

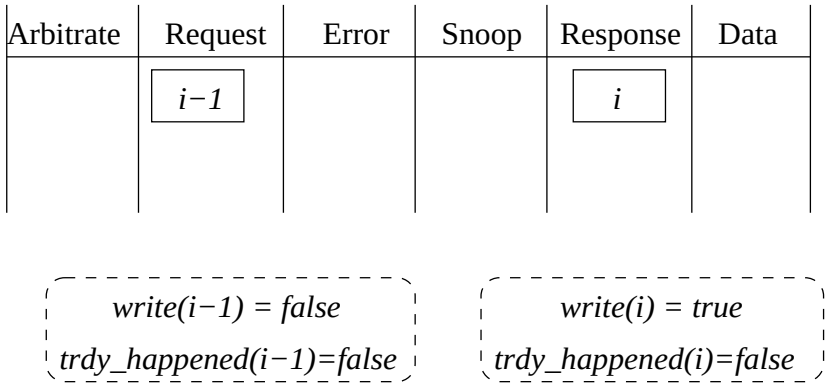
$$prev(write \wedge \neg trdy\_happened) \rightarrow \neg rs$$

For ease of understanding, the above property is for a single transaction. For a pipelined version, while transaction i is in the response phase for example, transaction $i-1$ can be processed concurrently, say in its request phase. Thus, there need to be multiple *write* variables and *trdy_happened* variables for each outstanding transaction (Figure 5.2). Assume that there is a mechanism to tag each transaction with an ID number. This same number can be used as a subscript on the history variables to create separate variables for each transaction. Thus, the history variables become $write_i$ instead of *write*, and $trdy\_happened_i$ instead of $trdy\_happened$. Also, the properties are activated only when a particular transaction reaches a particular pipeline phase. Consequently, each property developed in the un-pipelined version is (indirectly) indexed by a transaction ID in the pipelined version. The example property now becomes,

$$prev(write_i \wedge (response\_phase = i) \wedge \neg trdy\_happened_i) \rightarrow \neg rs$$

which is, "if transaction i is a write and is undergoing the response phase, then if *trdy* assertion hasn't happened yet, *rs* cannot be asserted." Thus, to create a pipelined version from an un-pipelined specification, the properties and history variables are replicated and indexed by transaction IDs.

The transaction ID assigning process is implemented by counters. In this scheme, the transaction ID corresponds to the order in which the transaction started. The first transaction that undergoes the arbitration phase (which is the first phase in the pipeline) is assigned the ID of 1. The ith transaction that undergoes the arbitration phase is assigned the ID of i. Since there can only be eight outstanding transactions, the IDs are assigned modulo 8. This ID scheme works because



At time *t*, a snapshot of the activity on the bus (two outstanding transactions) and the state of the variables

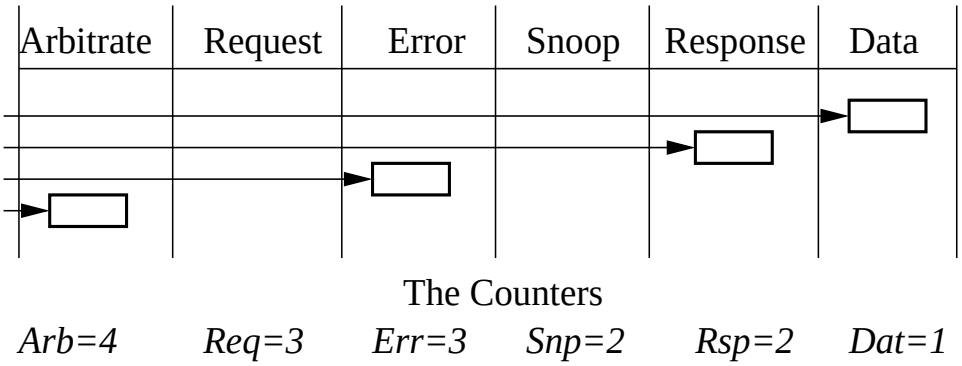
Figure 5.2: Two outstanding transactions and two sets of history variables

phases for a transaction happen in-order;[1] the *i*th transaction to undergo the request phase is the the *i*th transaction to go through the response phase.

For example, in Figure 5.3, four transactions have gone through the arbitration phase so far, and so, the corresponding counter *Arb* has incremented four times. Thus, the current transaction undergoing the arbitration phase is transaction 4, and all the specification properties that hold when transaction 4 is undergoing the arbitration phase are activated. Likewise, the transactions in the diagram can be labeled from right to left as 1, 2, 3, and 4. And from the counter values it is clear that transaction 1 is undergoing the data phase, 2 the response phase, 3 the error phase, and 4 the arbitration phase.

For each phase, there is a signal expression which indicates that the phase has completed. For a particular phase, if the number of occurrences of these "complete" signaling is known, the number of transactions that have undergone this phase so far is also known. In this way, the specification can keep track of which transaction is being processed in each phase. For example, the request phase's completion event is the assertion of *ads*. The request phase counter increments at every occurrence of this, and the counter value indicates the ID of the transaction currently in the request

[1]Data phases can be deferred and completed later so they do not necessarily happen in order, but this fact does not affect the ID scheme.



At time *t*, a snapshot of the activity on the bus with four outstanding transactions. The rightmost transaction (transaction 1) is undergoing its last phase and the leftmost transaction (transaction 4) is undergoing the first phase.

Figure 5.3: How transactions are assigned IDs

phase. As this transaction moves onto subsequent phases, each phase counter will have this same value.

Thus, the general form for properties is, "if transaction *i* is a write (or read) and it is currently undergoing the response (or request or snoop) phase, then *p* must hold."

$$prev(transaction\_type_i \wedge (some\_phase = i)) \rightarrow p$$

(where for example, $transaction\_type_i$ is a "write" and some_phase is a "response phase" counter)

Thus, using these counters and appropriate history variables, a pipelined protocol can be easily specified with small properties. The one drawback of this methodology is the linear increase in the number of state variables, which may be a problem when model checking. However, this is only a problem with verification and not the specification. The specification scales well; the properties only have to be duplicated with the counter values and variable subscripts changed. In fact, the methodology specifies a certain intricate Itanium™ bus feature effectively and simply. Although there are only five main pipeline stages, the protocol supports eight outstanding transactions at any time. The extra transactions are buffered at each stage. With the monitor-style properties, *the buffers do not have to be explicitly modeled* to specify the agent-bus behavior. There are simply

eight copies of the properties for the eight outstanding transactions.

5.1.2 Time-Unbounded Rules

Compared to the tightly timed PCI protocol, the Itanium™ Processor bus has a less constrained timing relationship among the different signal events. With PCI, most rules fall into the category, "exactly *n* cycles after event *a* happens, event *b must* happen." The PCI protocol is a *time-bounded* protocol where most events are guaranteed to happen within a certain time span.

In comparison, there are no such guarantees with the Itanium™ Processor bus protocol. Most of the rules follow the form, "any time after *n* cycles have elapsed since event *a*, event *b may* happen." The environment must expect that the event can happen at n, or $n+1$, or $n+2$, and so on, and be designed accordingly. An example from the protocol is "*trdy* may be deasserted a minimum of 3 cycles after the deassertion of the previous *trdy*".

Furthermore, there are no rules stating that an event must eventually happen (the so-called *liveness* property). In essence, the protocol is a *time-unbounded* protocol. Hence, the protocol allows the bus agents to have more freedom in ordering events, and optimizing bus performance. However, this extra degree of freedom leads to more corner cases in the specification that need to be checked.

Constraint Style

In the prior section, it is explained that the properties for this protocol are written in the form,

$$prev(transaction\_type_i \wedge (some\_phase = i)) \rightarrow p$$

The form of p will now be described. The most natural form, considering the "time-unbounded" characteristics of the protocol, is

$$p : \neg prev(q) \rightarrow \neg r$$

"If q is true in the previous cycle, then r *may* be true in the current cycle." The expression for q is a *trigger* condition which enables a certain exchange or a change of state. A trigger condition which signals that an agent is ready to receive data is one example. Another example is a trigger condition which indicates the completion of an event so that a bus signal can be deasserted.

One consequence of this time-unbounded feature is that the dead state check, in its original form, does not catch contradictions. This is due to the fact that *because most actions are not required to happen at any given time (the time-unbounded characteristic), the bus can always choose to loop in the current state and "do nothing."* If the check searches for a legal (all the specification properties are true) current state which has no legal next state, with a time-unbounded protocol, the current state is always a legal next state and so the check is vacuous. Therefore, the dead state definition needs to be expanded so that the test check for the following desired property,

$$\neg E[(correct_0 \wedge ... \wedge correct_n)\ U\ ((correct_0 \wedge ... \wedge correct_n) \wedge (AX(\neg correct_i \vee same)))]$$

where *same* is true if all the state variables, except the timer-like variables which increment at each clock, have the same value as in the previous state. Thus, this check ensures that, at every legal state, there is at least one possible legal next state where some change happens and the bus is not forced to stay in the current state. And so, a check that was effective for PCI is modified for the Itanium™ protocol so that a wider class of anomalies are detected.

5.1.3 2-Clocks or 1-Clock Reaction Time

Unlike the PCI protocol, the Itanium™ process bus is a latched protocol where there is a 1 cycle delay from when the bus agent asserts (or deasserts) a signal to when the action appears on the bus. Thus, when observing events on the bus, a reaction to a trigger event happens (at earliest) in two cycles instead of one. On cycle n, a trigger condition becomes true on the bus; on cycle $n+1$, the agent asserts a signal in response; and on cycle $n+2$, the assertion appears on the bus. And so most time-bounded properties are in the form, $prev(prev(input)) \rightarrow output$ where $input$ and $output$ are

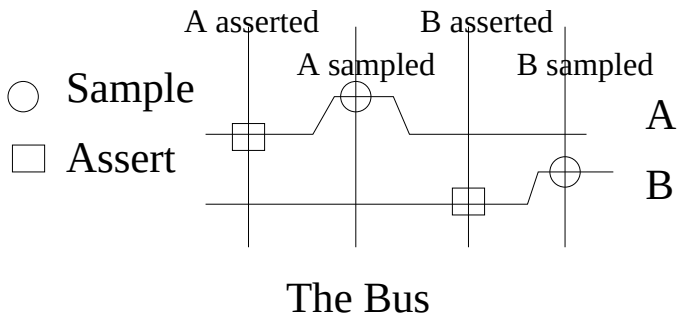


Figure 5.4: The Latched Bus Protocol - 2 Cycle Response Time

bus signal expressions.

However, the response to a trigger *from the same agent* may happen in one cycle. There is no one-cycle delay through the latched bus in this case. For example, there is a rule, "*ids* cannot be asserted in cycle N if *trdy* is sampled asserted on clock N." As a property, this becomes $prev(trdy) \rightarrow \neg ids$. This requirement is possible only because both *trdy* and *ids* are controlled by the same agent at all times. The agent does not need to wait for the trigger (*trdy*) to appear on the bus to react ($\neg ids$) to it.

The difficulty arises when the trigger condition of a rule is a *mix* of external and internal signals. An example is "$input \wedge output_0$ true requires $output_1$ to become true." If an external signal causes the trigger expression to become true ($output_0$ was already true and *input* just became true), then a two cycle reaction time is needed. However, if the agent's own signal causes the trigger condition to become true (*input* was already true and $output_0$ just became true), it can react in one cycle. Thus, the rule needs to be separated into two properties depending on the situation. However, the English specification fails to distinguish between the two cases, and states a blanket requirement allowing a one cycle reaction timing at all times. In fact, the dead state check detected a scenario where these misstated rules led to a contradiction.

5.2 Debugging

Several issues were found with the development version of the Itanium™ processor bus protocol using the methodology's debugging procedures. Some are omission of rules that are arguably implicit in the official specification, but violate the completeness concept. Others are serious enough to cause data phases to be missed unnoticed, or cause a deadlock. These were resolved by revising the Intel® protocol specification.

5.2.1 Found by Dead State Check

The dead state check mainly found cases of missing rules. The informal specification tends to state the sufficiency conditions of an action, while leaving necessary conditions implicit. For example, a sufficient condition for the assertion of *trdy*, "if the transaction is a write, then a *trdy* assertion must happen," is stated in the specification. Logically, it specifies $write\_transaction \rightarrow trdy\_happens$. However, a necessary condition that *trdy* can only be asserted at certain times, is missing. The specification should state that "*trdy* can be asserted *only if* the transaction is a write or has a snoop-initiated data transfer." Else, the system will reach an undefined state because the agents do not expect *trdy* to be asserted during a read, for example. By adding such a rule, the specification is made more complete so that a simulation checker can catch erroneous behavior at the earliest time. Overall, there were five cases of such omissions where the specification does not state that a particular event can happen *only if* certain conditions are true.

5.2.2 Found by Characteristic Check

Missing Trigger Condition and Resulting Deadlock

There is a pair of communicating agents: a data sending agent, the *Sender*, and a receiving agent, the *Receiver*. When the bus signal *trdy* is true, the Receiver is signaling that it is ready to receive data. When *dbsy* is true, the Sender is sending data. So when $\neg dbsy$ is true, the Sender is idle, and is ready to start the next data transfer. Thus, when $trdy \wedge \neg dbsy$ is true, both agents are ready;

consequently, $trdy \wedge \neg dbsy$ is a trigger condition that allows a new data phase to start.

The protocol is designed so that the Receiver keeps *trdy* true until the data sending agent is idle and *dbsy* is deasserted ($\neg dbsy$). Thus, the normal sequence of events is as shown in Figure 5.5. Note the one cycle delay between a signal change and its appearance on the bus because of the latched property.

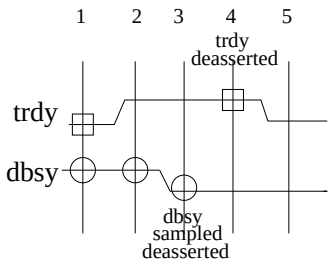


Figure 5.5: *trdy* and *dbsy* relation

Time	Receiver	Sender
1	asserts *trdy* and observes that *dbsy* is active	driving *dbsy*
2	keeps *trdy* asserted	deasserts *dbsy*
3	observes that *dbsy* is low	idle
4	in response, deasserts *trdy*	idle
5	idle	idle

What makes the protocol tricky is that if *dbsy* is already deasserted, as an optimization, *trdy* can assert and deassert right away (Figure 5.6). Note that, unlike the normal sequence, the trigger ($trdy \wedge \neg dbsy$) is true for only one cycle in this case. Thus, this scenario lets the idle state, $\neg trdy \wedge \neg dbsy$, happen a cycle earlier (that is why it's an optimization).

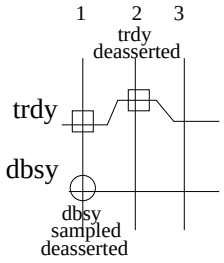


Figure 5.6: When Optimized : *trdy* and *dbsy* relation

Time	Sender	Receiver
1	asserts *trdy* and observes that *dbsy* is inactive	idle
2	deasserts *trdy* as an optimization	idle
3	idle	idle

It is important that *trdy* and *dbsy* never become true on the same clock because of the following possibility (Figure 5.7). On cycle 1, the Receiver samples *dbsy* low so it does the optimization, but because in cycle 2, *dbsy* is high, *trdy* is inadvertently deasserted before the trigger condition becomes true. So the awaited trigger condition never happens and a data phase cannot start.

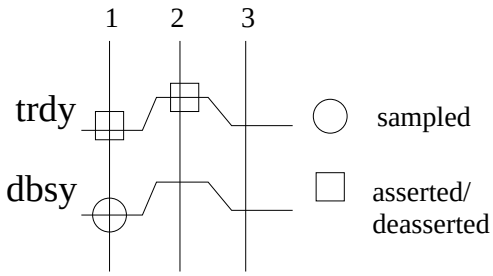


Figure 5.7: Missed Trigger Condition : $trdy \wedge \neg dbsy$ never happens

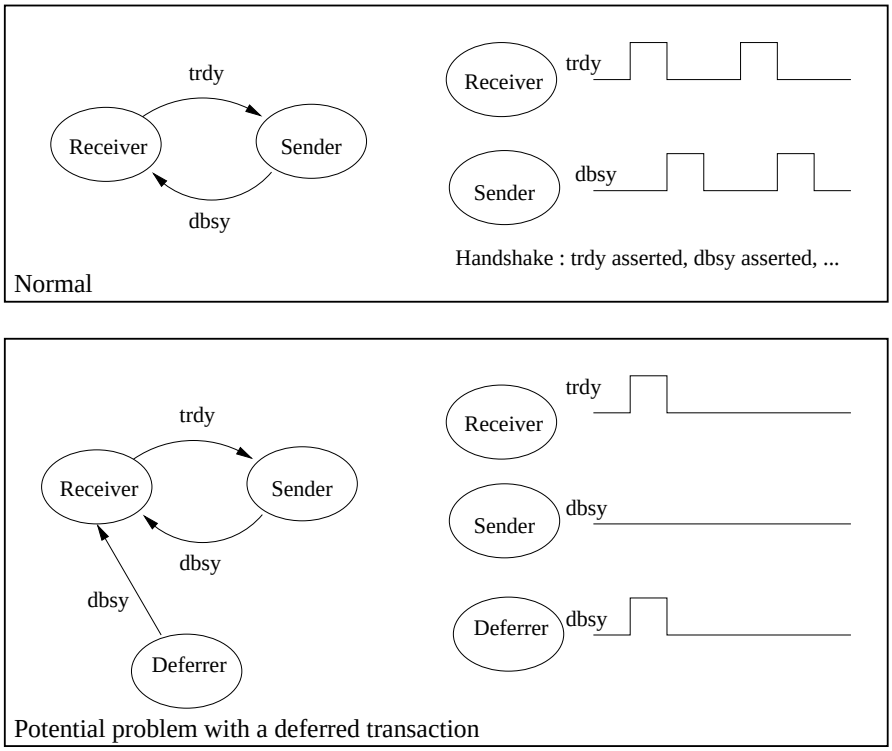
In most cases, the protocol is designed so that *trdy* and *dbsy* cannot become true on the same clock, but, in a special case, a loophole in the protocol allows this. Normally, because *trdy* and *dbsy* are handshake signals between two communicating agents, they are not asserted at the same time. However, there is a case where a third agent can assert *dbsy* and thereby break this rule (Figure 5.8). This special case is when an agent which previously deferred to complete a data phase (the *Deferrer*), takes advantage of the apparently idle bus to complete the deferred data phase. Meanwhile, there is a separate ongoing transaction where an Receiver agent is about to assert *trdy* to communicate to a Sender agent. The Deferrer asserts *dbsy* to start the data transfer at the same time the Receiver asserts *trdy*, and, as a result, *trdy* and *dbsy* are asserted at the same time. Note that, in this scenario, the *trdy* and *dbsy* assertions are not for the same data transfer. This overlooked case causes the normal data transfer to wait forever for the (missed) trigger condition of $trdy \wedge \neg dbsy$.

In a similar case,[2] a deadlock occurs because a transaction cannot proceed unless the data phase completes.[3] But the data phase can not happen because the trigger condition did not become true.

How the issue was found Since this trigger condition is crucial for data phases, and optimizations often lead to unexpected scenarios, whether the sequence shown in Figure 5.7 was allowed by

[2]For the protocol experts: this happens when 1. the responding agent and the requesting agent are the same and 2. it is a write with a snoop-initiated data transfer, where the second *trdy*, which is for the snoop-initiated transfer, happens exactly when the data phase (which was allowed to be indefinitely delayed because of 1.) for the first *trdy* starts.

[3]For the protocol experts: this requirement is because the second data phase is snoop-initiated, and it must happen together with the response.



Top Diagram : In the normal mode of operation, the Receiver and the Sender use the *trdy* and *dbsy* signals as part of a handshake communication. Thus, they are never asserted at the same time. Bottom Diagram : When there is a deferred agent from a previous transaction, this agent may assert *dbsy* and disrupt the handshake (i.e. *trdy* and *dbsy* are asserted at the same time and the trigger condition never happens).

Figure 5.8: Missing Trigger Condition That May Result In Deadlock

the specification was one of the first checked characteristics. Although these characteristics cannot be automatically deduced, after the specification process, characteristics testing for suspicious sequences can be developed with little difficulty.

Dropped Data Phase

Another class of issues was found with the characteristic check. Under certain circumstances, a write data phase can be delayed indefinitely.[4] The potential danger of this is that the data phase never happens, and the system proceeds without any trace of the phantom data phase. By using the characteristic check, such a scenario was discovered.

The basic signalling mechanism of a write transaction is,

1. The Receiver indicates a "ready" state by asserting *trdy* true.
2. The Receiver may deassert *trdy* false before the Sender starts the data phase.
3. The Sender, acknowledging the "ready" signal, starts the data transfer by asserting *dbsy*.

Thus, the normal sequence of events is, "*trdy* is asserted and then deasserted, data is transmitted, *trdy* is asserted and then deasserted, data is transmitted, ..." Now, consider the sequence "*trdy* is asserted and then deasserted, *trdy* is asserted again, data is transmitted, ... (Figure 5.9)." The one-to-one correspondence between a *trdy* assertion and a data transfer breaks down. The second *trdy* assertion should not have happened before the start of the first data transfer. Consequently, the data phase for the first *trdy* misses its window to start the transfer. By model checking the specification, it was found that this sequence can indeed happen in the case where a data phase is delayed indefinitely.

How the issue was found This was found by model checking whether the specification allows the characteristic, "a second *trdy* assertion can start before the start of a pending data phase". Coincidently, this problem was also discovered using simulation by the testing team at Intel, but

[4]For the protocol experts: this happens when the responding agent and the requesting agent are the same.

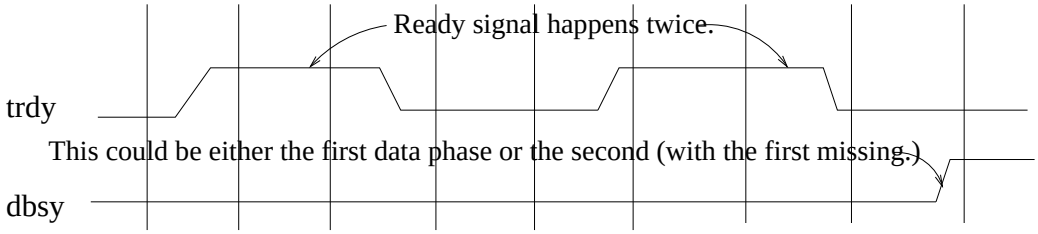


Figure 5.9: Early second *trdy* assertion

since our methodology does not require an implementation, it found the problem in a shorter time with less effort.

Chapter 6

Exploiting Specifications

6.1 Introduction

This chapter focuses on the second angle of the cost-value problem: to increase the value of a formal specification beyond its role as a documentation. It is based on the idea that if formal specifications can be used in novel ways that enhance design productivity, practitioners may be less reluctant to develop them. Furthermore, a stylistically structured and formally verified specification is very much primed to be exploited. Such a specification encapsulates information about the interface design in a highly usable form. One can imagine using the specification for design automation, synthesis, and simulation-based design validation. This thesis demonstrates how the specification can be used to enable verification of a design using a model checker (for smaller designs) or a simulation-based approach (for designs too large to be model-checked). This chapter, in particular, focuses on simulation-based validation, and demonstrates how a formal specification can automate many procedures that are now done manually.

[0]This chapter is based on material first published in *Proceedings Design Automation Conference* (DAC 02), 2002 [**?**].

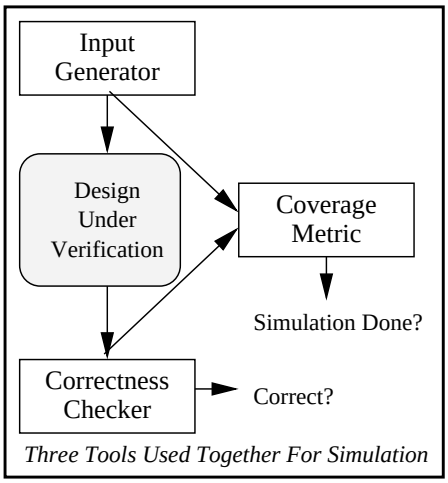


Figure 6.1: The Trio of Verification Aids

6.2 The Problem and the Approach

Given an HDL (hardware description language) design module to verify using software simulation, an engineer needs various additional machinery (Figure 6.1).

1. Input Logic There must be logic to drive the inputs of the design. One way to do this is to use random sequences. The problem with this approach is that because the inputs are not guaranteed to be correct (they might be "garbage" from the module's point of view), it is difficult to gauge correctness of the module design. A less haphazard method is directed testing where input sequences are manually written, but they are time-consuming to write and difficult to get correct.

2. Output Check Logic to determine the correctness of the module's behavior is needed since manual scrutiny is usually too cumbersome. There are two levels of correctness. One is interface protocol compliance where the design outputs are checked for protocol violations. The other is a more higher-level correctness where one checks, for example, whether the behavior at one interface of the module correctly corresponds to the behavior at another interface of the same module.

3. Coverage Metric Because complete coverage with all possible input sequences tested is not possible, there must be some metric that quantifies the progress of verification coverage. The verification engineer would like to know whether the functionalities of the design have been thoroughly

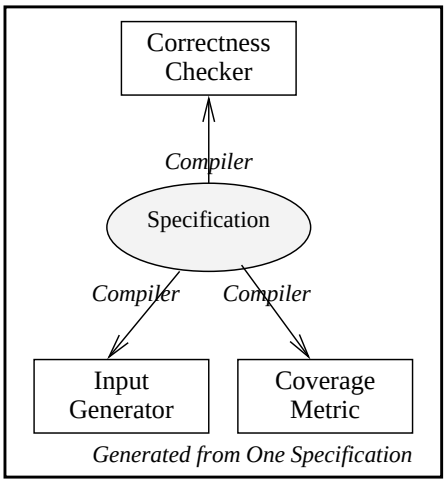


Figure 6.2: The Unified Framework Approach

exercised and all interesting cases have been reached during the simulation.

The Approach At the foundation of the methodology is a *unified framework approach* where the three tools are generated from a single source specification (Figure 6.2). This is possible because all three are fundamentally based on the interface protocol. The input generator generates sequences based on what is allowed by the interface protocol; the output checker compares the output to what is deemed correct by the protocol; and, coverage can be quantified by exploiting the fact that the protocol defines the set of all possible interface events. Thus, the interface specification can be transformed into the three tools using automated methods. In comparison, currently, the verification aids are each written from scratch, and a tremendous amount of time and effort are required for their development. By eliminating this step, the methodology enhances productivity and shortens overall development time.

Furthermore, a thoroughly debugged, solid specification invariably leads to correct input sequences, checking properties, and coverage metrics. The correctness of the core document guarantees the correctness of the derivatives. In contrast, with current methods, each verification aid needs to be individually debugged. The advantages of this are most pronounced for standard interfaces where the correctness effort can be concentrated in the standards committee and not duplicated

among the many implementors. Furthermore, when a change is made to the protocol (a frequent occurrence in industry), one change in the protocol specification is sufficient to reflect this because the verification aids can be regenerated from the revised document. Otherwise, the engineer would have to determine manually the effect of the change for each tool.

Generating the Three Tools The derivation of a behavioral checker from the specification is the most straightforward of the three. The checker works on-the-fly; during simulations, it flags an error as soon as the module violates the protocol. It checks for protocol conformance but does not check for the higher-level correctness described earlier. For example, a PCI checker will verify that a module obeys the protocol rules $trdy \rightarrow devsel$ or $prev(trdy \wedge stop) \rightarrow stop$. The specification is guaranteed to be executable by the style rules (described in section 2.5), and so the translation from it to a HDL checker requires minimal changes.

The bulk of this chapter addresses the issue of automatically generating input sequences. The method produces an input generator which is dynamic and reactive; the generated inputs depend on the previous cycle outputs of the design under verification. In addition, these inputs always obey the protocol, and the generation is a one-pass process. On every clock cycle, the algorithm solves the constraints currently imposed by the specification on the inputs. From the design's perspective, it cannot discern the difference between this setup and interacting with an actual HDL implementation of the environment.

Although input generation using constraint solvers is not by itself novel, this thesis's approach is the first to use and exploit a complete *and verified* specification. Writing constraints on an *ad hoc* basis for the express purpose of generating signal sequences is commonly done. However, very few have succeeded in transforming an existing, complete specification into a generator without a debilitating computational overhead. Thus, this methodology delivers on the goal of increasing the value of an interface specification by using an existing technique.

More importantly, many who use constraint-based input generation do not verify the constraints in any systematic way. Consequently, it is very likely that the constraints are incorrectly

over-constraining or under-constraining. With over-constraining constraints, not all legal input sequences will be generated, and the design will not be fully exercised. With under-constraining constraints, incorrect input sequences will be generated, and a correct design may be erroneously flagged for errors. Thus, it is very important to verify the constraints to avoid false positives and false negatives. However, in practice, this is not done, and from developing the PCI and the Itanium™ specifications, it can be said that mis-specifying constraints (properties) is a common and frequent problem.

Finally, a new simulation coverage metric will be introduced, and the automatic input biasing based on this metric will also be described. Although more experiments are needed to validate this metric's effectiveness in measuring coverage, its main advantage (currently) is that it is specification-based and saves time: extra work is not needed to write out a metric or to pinpoint the interesting scenarios for they are gleaned mechanically from the specification document.

6.3 Previous Work

Clarke et al. also researched the problem of specifications and generators in [**?**] but they only outlined an algorithm and did not persuade nor demonstrate that their methodology will work with a large RTL design. The technique described here is most closely related to the *SimGen* project described in the 1999 paper [**?**] by Yuan et al. As with the SimGen work, this thesis focuses on practical methods that can be used for existing complex designs. Within that framework, there are mainly two features that differentiate the thesis's approach from SimGen. Both are possible because of the stylistic structure imposed on the specification. This structure is exploited to achieve, one, more memory-efficient input generation and, two, to automatically guide simulation coverage.

Memory Efficiency The memory efficiency can be attributed to the dynamic selection of relevant specification properties on a cycle-by-cycle basis to create a smaller BDD. In contrast, the SimGen software uses a statically-built BDD which represents the entire input specification logic.

Because the interface protocol can be complex and involve many signals, such a BDD can be problematically large.

This selective-dynamic BDD building approach results in a smaller BDD for two reasons. One, only a small percentage of the protocol logic is relevant on each cycle, and so the corresponding BDD is always much smaller than the static BDD representing the entire protocol. Two, the BDD *contains only the design's input variables and does not contain state variables or the design's outputs.* Thus, for the PCI example, instead of a BDD on 161 variables, the BDD only has 15 variables, and typically BDD size grows exponentially with the number of variables. This reduction is based on the observation that the sole role of the state variables and the design's outputs is to determine which parts of the protocol are relevant (and thus required in the BDD) for a particular cycle. Otherwise, these variables are not needed to calculate the inputs. It is believed that these two reasons for smaller BDDs would hold for many interfaces and therefore allow input generation for a large interface that may otherwise be hindered by BDD blowup.

Automatically Determined Biases SimGen has capabilities for input biasing, but unlike this thesis's framework, it requires the users to provide the input biases. A unique contribution of this thesis is the automated process of guiding the simulation to an interesting scenario by determining the corresponding set of input biases.

It is noted that all these advantages are possible because the method exploits the structure of a stylized specification whereas SimGen is applicable for more general specifications.

6.4 Style Rules Re-Visited

In Chapter 2, the basic structure of the specification and the style rules were introduced. In this section, the style rules will be re-visited to illustrate how the input generation algorithm takes advantage of them.

6.4.1 Style Rule: Isolating Current Variables in a Property

This style rule requires the properties to be written in the following form.

$$prev(signal_0 ... \wedge \neg signal_j ... \vee variable_0 ... \wedge \neg variable_k)$$
$$\rightarrow signal_i \vee ... \wedge \neg signal_n$$

where the properties are written as an implication with the past expression (events from any state previous to the current state) as the antecedent and the current expression as the consequent. In essence, the past history, when it satisfies the antecedent expression, requires the current consequent expression to be true; otherwise, the property is not "activated" and the interface signals do not have to obey the consequent in the current cycle. In this way, the *activating* logic and the *constraining* logic are separated. For example, the PCI protocol property, $prev(trdy \wedge stop) \rightarrow stop$ means "if the signals *trdy* and *stop* were true in the previous cycle (the activating logic), then *stop* must be true in the current cycle (the constraining logic)".

This separation is key to memory efficient signal generation; it identifies the relevant (i.e."activated") properties on a particular cycle so that only these properties are used to generate inputs . The other properties, since they are not activated, can be ignored for this particular cycle. Also, the separation allows just the constraining half of each property to be used in the final constraint formula. The activating half can be discarded because they are used *only* to determine the relevance of the property in a cycle. Thus, for these two reasons, the final formula is much smaller, and memory efficiency is greatly improved.

6.4.2 Style Rule: Separability of Properties Rule

The *separability* rule, requires *each property to constrain only the behavior of one module (agent).* Thus, a single property should only constrain outputs from one module and not its inputs nor other modules' outputs. Equivalently, because the constraining part is isolated from the activating part (due to the previous style rule), the rule requires *the consequent to contain only outputs from one module.*

When generating signals, because only the consequent halves are used in the final formula (while the antecedent half is discarded) and the consequents only contain outputs from one module, the number of variables in the property formula is greatly reduced. Otherwise, the formula will contain internal state variables and output variables from other modules. Consider the PCI property, "master must raise IRDY# within 8 cycles of the assertion of FRAME#" which translates to "IF the agent is the master (*masteris = true*) and it has been 7 cycles since *frame* was asserted and *irdy* has not been asserted yet (*frame8 = 7*) and *irdy* is not asserted in this cycle ($\neg$*irdy*), THEN the output *irdy* must be true in the next cycle."

$$prev((masteris = true) \wedge (frame8 = 7) \wedge \neg\ irdy) \rightarrow irdy$$

Without the separation technique, the final constraint formula will have to contain *masteris* (a 1-bit state variable), *frame8* (a 3-bit counter), *irdy* (a 1-bit state variable), and *irdy_out* (a 1-bit free variable whose value we will choose). With the technique, the final formula contains only *irdy_out* if the property is activated and nothing if it is not. This is why for the PCI example, the separation technique reduced the number of boolean variables in the BDD data structure from 161 to 15.

6.4.3 Dead State Free

It has been determined in the "Verifying a Specification" chapter that a good specification is free of a certain type of contradiction, dead states. Dead state free-ness effectively guarantees that an output vector satisfying all of the properties always exists for a module as long as the output sequence so far has not violated the properties.

Recall the formal definition: For every state in the specification monitor where no properties for any module have been broken so far, there must exist at least one possible next state where all of the properties for a particular module are satisfied. For a particular module *i*,

$$\neg E[(correct_0 \wedge ... \wedge correct_n)\ U\ ((correct_0 \wedge ... \wedge correct_n) \wedge (AX \neg correct_i))]$$

("There is no scenario where all the modules have been correct so far but module *i* has no possible correct next state"). This characteristic is necessary to guarantee that on every clock cycle, the

constraint on the outputs is satisfiable.

6.5 Generating Example Waveforms

After a protocol is fully specified with properties, these properties can be used to generate example waveforms. This can be done before any implementations have been designed when designers would like to see example waveforms to learn and understand the protocol. The ability to generate waveforms is important for property-based specifications because there is no explicit state machine to study to understand interface behavior; after all, visualizing a series of signal transitions from a collection of properties is difficult. Furthermore, generating example waveforms is a quick and easy way to check whether the specification is complete and there are enough properties to correctly constrain the interface. If there are obvious "gaps" in a specification, they can be found quickly by examining the generated waveforms; the waveforms will demonstrate odd behavior that is easily detectable when required properties are missing in the specification. With the PCI specification, cases of missing properties were discovered using this method *after* model checking had been done. This illustrates the advantage of this specification debugging method over model checking which requires the user to specify checking characteristics that will pinpoint the error (and is thus limited by the creativity of the user).

6.5.1 Algorithm

The setup for this scheme is based on automatically creating *stub modules*, which each represent a particular agent or module at the interface. By using the properties, the stub modules generate outputs on every cycle, and demonstrate how the interface modules would interact using the protocol (Figure 6.3).

1. Group the properties according to which interface module they specify. (This is possible because of the "Separability" rule.) If there are n interface modules, there will be n groups. Each group represents a stub module.

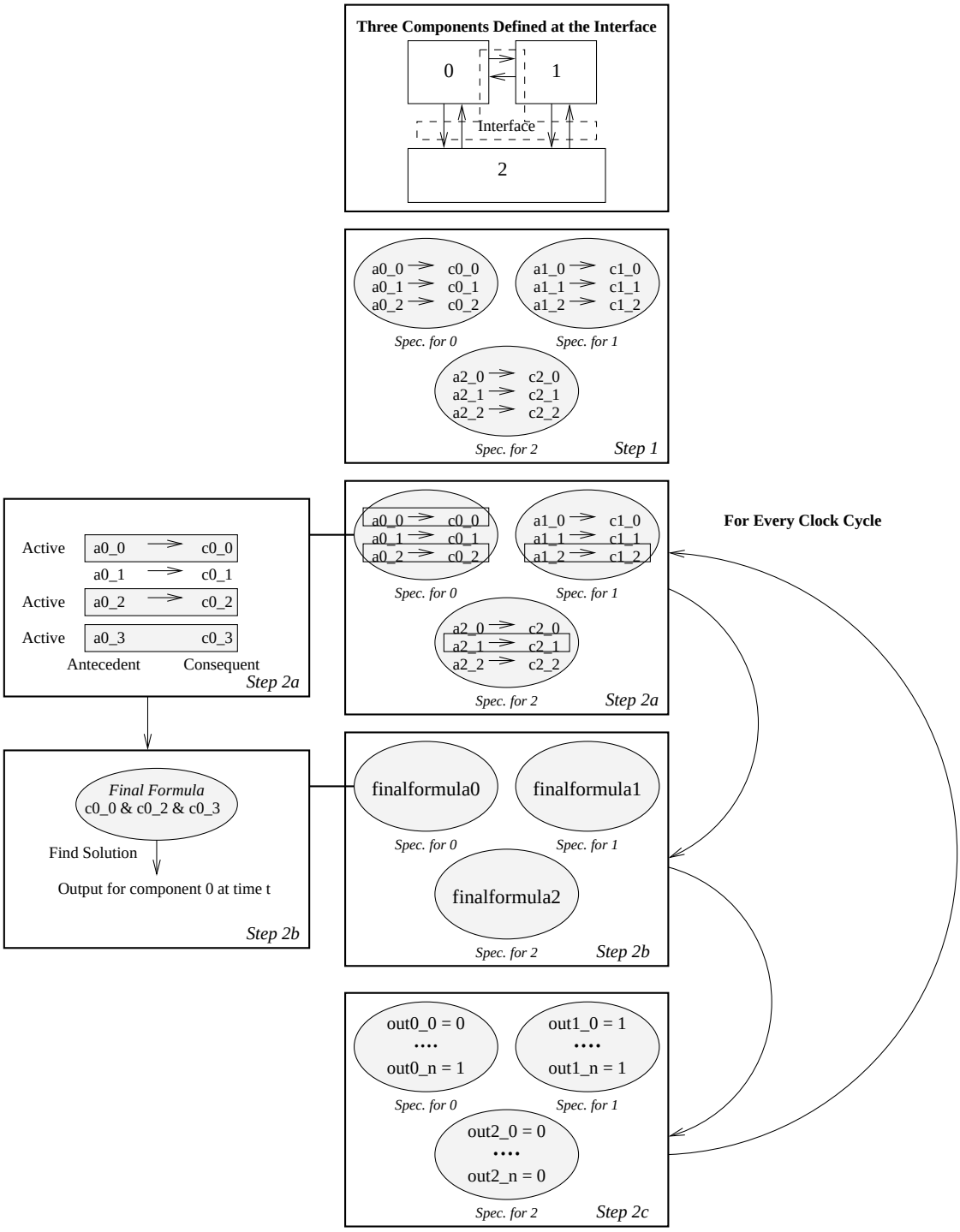


Figure 6.3: The Waveform Generation Algorithm

2. For each group of properties, do the following on every clock cycle of the simulation run. The goal is to choose an output assignment for each stub module for the next cycle.

   (a) For each property, evaluate *just the antecedent half.* The antecedent values are determined by internal state variables and observed interface signal values. (For the initial state, the interface signal values are pre-defined. For subsequent states, the interface signal values are determined in the previous clock cycle.) For antecedents which evaluate to *true,* the corresponding properties are marked as *activated.* This activate or non-activate decision procedure is possible because of the "Isolate Current Variables" rule.

   (b) Within each group, AND together *just the consequent halves* of the activated properties to form the *final formula.* As a result, there is one final formula for each stub module. Because of the "Separability" rule and the initial grouping of properties, each final formula consists purely of output signals from a single module. Solving this final formula is equivalent to finding an output assignment for a stub module that satisfies the specification for the next clock cycle.

   (c) A satisfiability solver based on binary decision diagrams (BDD) is described in section 6.5.2. The solver is used to determine a solution to each of the final formulas. A solution is guaranteed to exist because of the lack of dead states in the specification. These solutions are then assigned to the corresponding output signals of the stub modules.

   (d) Go back to step 2(a) on the next clock cycle.

Using this algorithm, example waveforms were generated from the property-based PCI specification. Some of the automatically created waveforms can be seen in Figure 6.4.

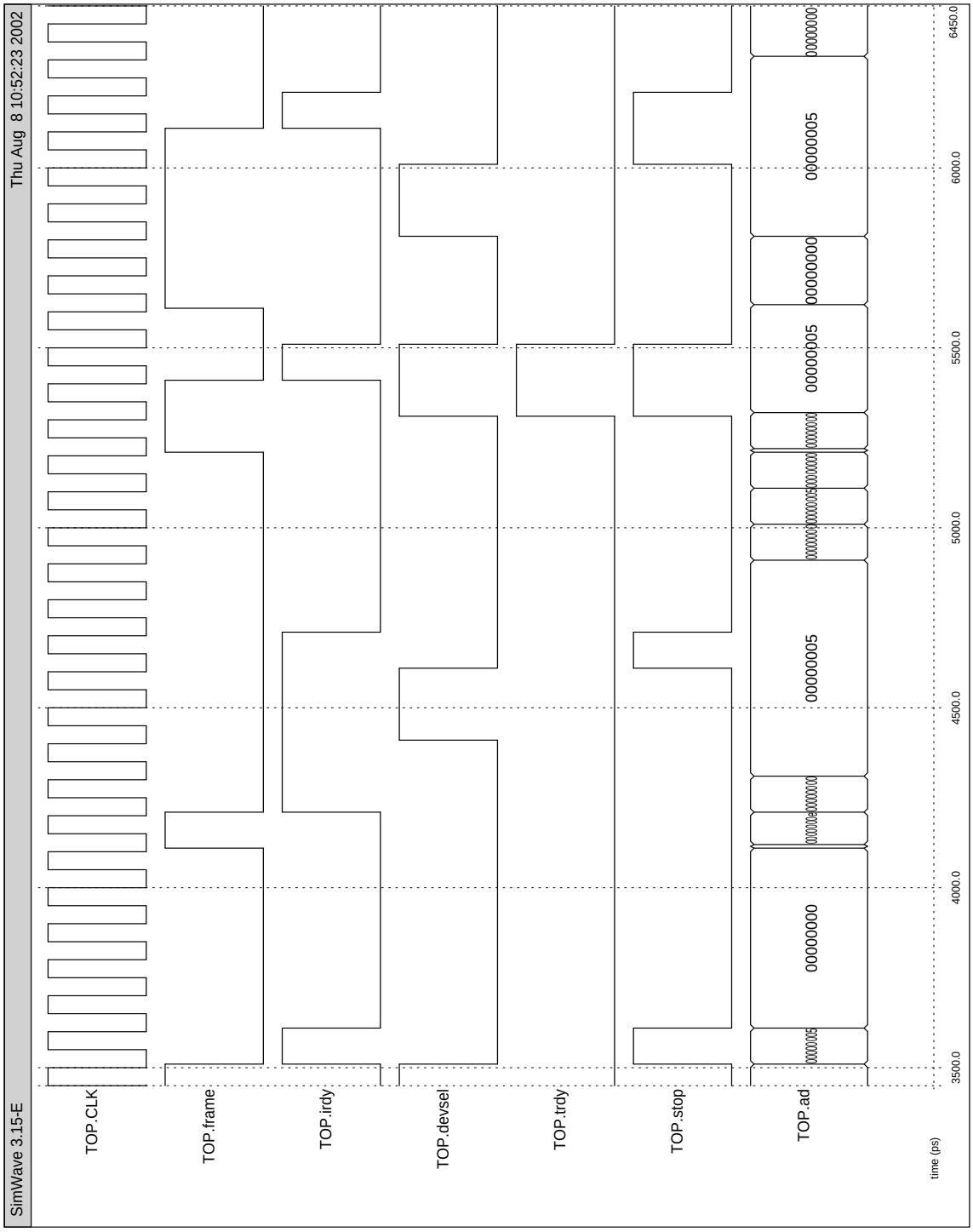
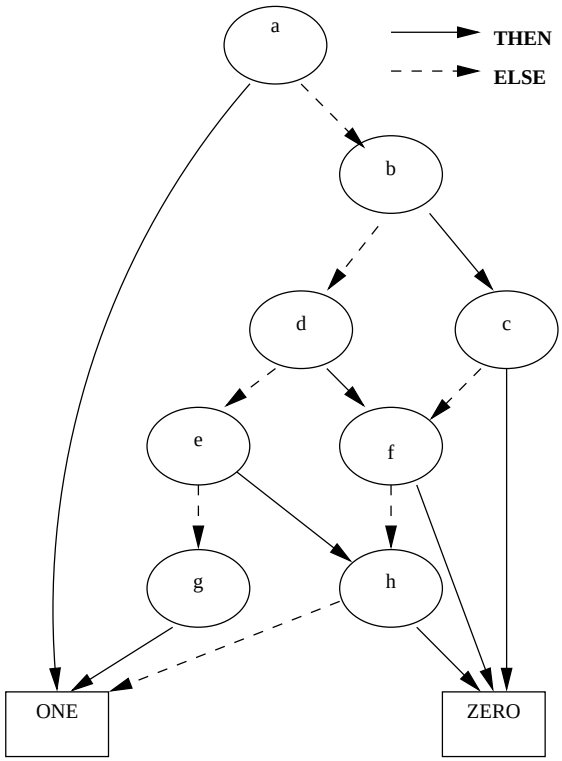


Figure 6.4: A PCI protocol example waveform generated using the protocol properties



By traversing a BDD, a satisfying assignment can be found for the constraint represented by the BDD.

Figure 6.5: Solving a Constraint by BDD Traversal

6.5.2 Binary Decision Diagrams

To find variable assignments that satisfy a boolean constraint, a Binary Decision Diagram (BDD) [**?**] is often used. A constraint can be transformed into a tree-like BDD where each node corresponds to a variable in the constraint (Figure 6.5). A node has two outgoing branches; the THEN branch is taken if the variable is set to true, and the ELSE branch if false. By traversing this tree, eventually one of the two leaf nodes will be reached. The ONE terminal node indicates that the choices of variable assignments along the path taken ($a =$ TRUE, $b =$ FALSE, $c =$ FALSE, ...) satisfies the constraint while the ZERO node indicates that the assignment does not. For example, in Figure 6.5, the path ($a =$ FALSE, $b =$ TRUE, $c =$ FALSE, $f =$ FALSE, $h =$ FALSE) leads to a ONE node so the assignment satisfies the constraint.

This BDD-based constraint solving technique is used in the input generation algorithm. For every cycle, after the final formula is determined, this constraint on a module's outputs is transformed into a BDD. All the nodes in this BDD are output signals of the module, and by traversing the BDD, an output vector satisfying the specification can be found.

A BDD-based constraint solver is used instead of a general satisfiability solver because the former is more amenable to biasing. Biasing is a technique where the statistics of individual signals are controlled. For example, a signal can be biased to be true 99% of the time or true 1% of the time. Biasing can lead to increased simulation coverage as explained in section 6.7.

6.6 Deriving an Input Generator

When an implementation for one module has been designed and correct inputs are needed to stimulate the design, the above waveform generation algorithm can be used to create an input generator. The stub modules (minus the one for the implemented module) are used to emulate the behavior of the other modules at the interface and act as the environment for the design. The following algorithm, which is a modified version of the waveform generation algorithm, outlines how the properties are used to generate inputs for the design (Figure 6.6).

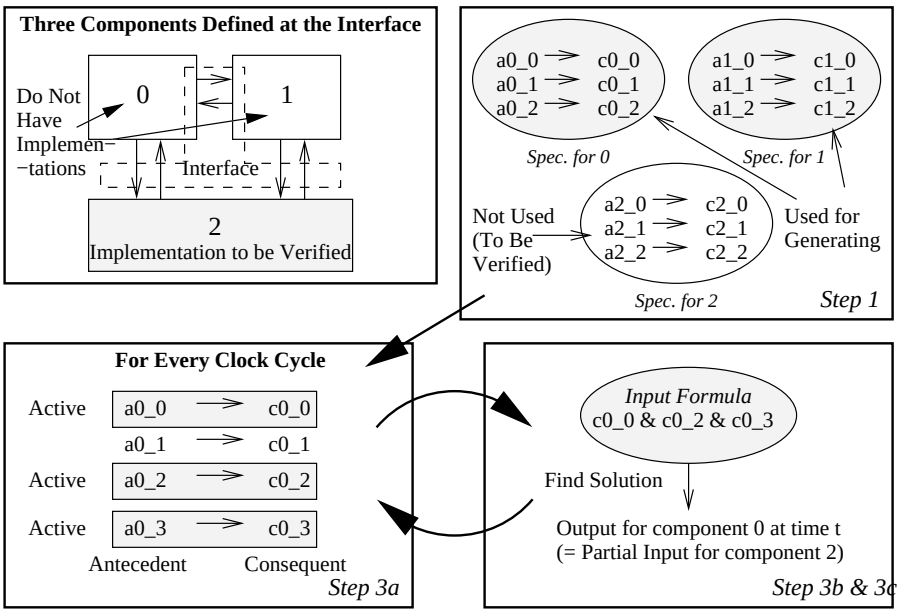


Figure 6.6: Inputs are generated automatically and reactively to a design.

6.6.1 Algorithm

1. Group the properties according to which interface module they specify. If there are n interface modules, there will be n groups.

2. Remove the group whose properties are for the design under verification. These will not be needed. Now, there are $n - 1$ groups of properties.

3. For each group of properties, do the following on every clock cycle of the simulation run. The goal is to choose an output assignment for each stub module (and consequently an input assignment for the design) for the next cycle.

   (a) Evaluate the antecedent halves to find activated properties. Through this process, the input generator is reacting to the design's outputs because the previous cycle outputs affects the truth value of the antecedents.

   (b) Within each group, AND together the consequent halves of the activated properties to form the final formulas.

(c) A solver determines a solution to each of the final formulas. Since the specification is flexible and allows a range of behaviors, there will most likely be multiple solutions. (In section 6.7, we discuss how a solution is chosen so that interesting simulation runs are generated.) The chosen solutions form the input vector for this cycle.

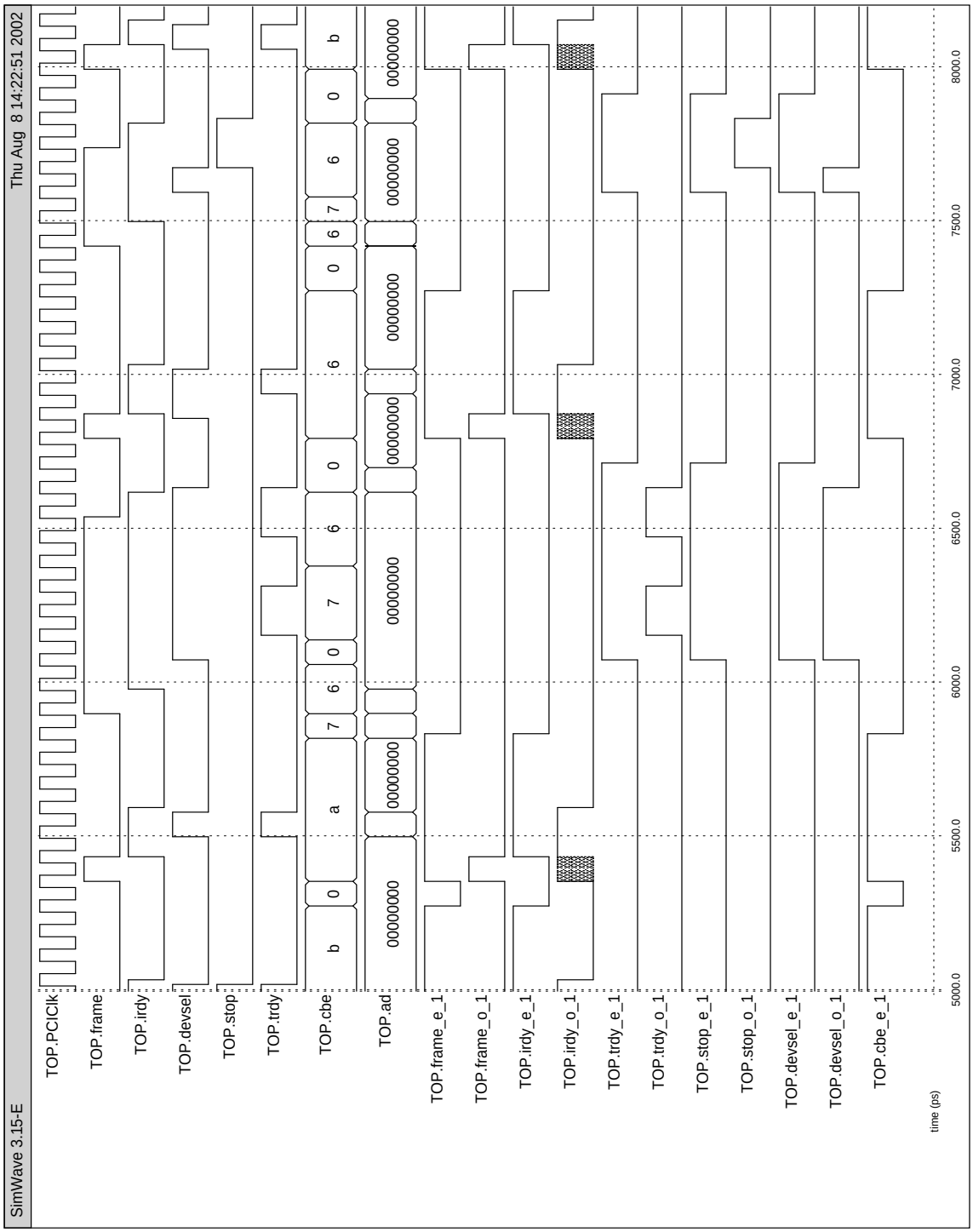
(d) Go back to step 3(a) on the next clock cycle.

This algorithm was used to generate input sequences for a PCI implementation. In section 6.8, the details of the experiment along with a description of the design under verification are outlined. But here, in Figure 6.7, the interaction between the design and the input generator are shown. The first seven signals after the bus clock (`PCIClk`) from `frame` to `ad` are bus signals, and they are inputs of the design. They are the incoming signals that the design sees in a bus system. The next eleven signals from `frame_e_1` to `cbe_e_1` are direct outputs of the design that drive the bus. The diagram illustrates the behavior of the design as a master initiating transactions (when `frame_e_1` or `irdy_e_1` are active) and as a target reacting to a transaction initiation (when `devsel_e_1`, `trdy_e_1`, or `stop_e_1` are active). Conversely, the input generator is acting as a target to the design's master behavior and a master to the design's target behavior.

6.6.2 Implementation

A compiler tool, which reads in a specification and outputs the corresponding input generation module, has been designed and implemented. The tool was used to automatically create a PCI input generator from the formal PCI specification. There are two parts to the input generator: the Verilog module which acts as the frontend and the C module as the backend (Figure 6.8).

The Verilog module contains all the antecedents of the properties, and based on its inputs (the outputs of the design under verification) and its internal state variables, determines which properties are activated for that clock cycle. Then, the indices of the activated properties are passed to the backend C module. The C module will return an input assignment that satisfies all the activated properties, and the Verilog module will output this to the design under verification. The choice of



A PCI interface design and the input generator are interacting in design-master mode and design-target mode.

Figure 6.7: A PCI interface design and the input generator

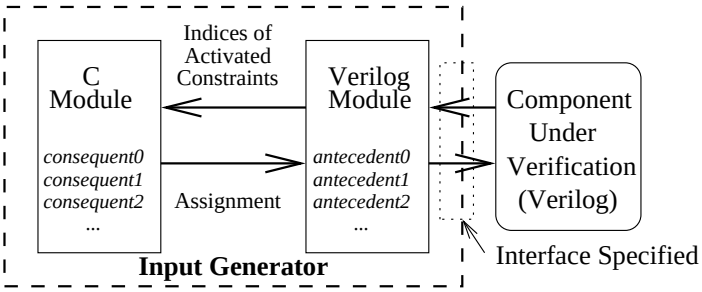


Figure 6.8: Implementation Details of the Input Generator

Verilog as a frontend allows many designs to be used with this framework.

The C module contains the consequent halves of the properties. It forms conjunctions (ANDs) of the activated consequents, solves the resulting formula, and returns an assignment to the Verilog module. It is initialized with an array of BDDs where each BDD corresponds to a property consequent. On every clock cycle, after the activation information is passed to it, it forms one BDD per interface agent by performing repeated BDD AND operations on activated consequents in the same group. The resulting BDD represents an (aggregated) constraint on the next state inputs of one agent, and by traversing the BDD until the "1(TRUE)" terminal node is reached, an assignment can be found. Once an assignment is determined for each interface agent, the complete input assignment to the design under verification has been established. The CUDD (Colorado University Decision Diagram) package [**?**] version 2.3.1 was used for BDD representation and manipulation, and Verilog-XL was used to simulate the setup.

6.7 Biasing the Inputs

6.7.1 Coverage Metric

We use the specification to define *corner cases*, interesting scenarios that a simulation run should cover. Once corner cases are determined, whether they have been reached or not can be used to measure simulation progress, and missed corner cases can be used to determine the direction of further simulations.

As a first order approximation of corner cases, the antecedents of the properties are used. This is because only when the antecedent clause is true does the implementation have to comply with the property clause. As an example, consider the PCI property, "master must raise irdy within 8 cycles of the assertion of frame." The antecedent is "the counter that starts counting from the assertion of frame has reached 7 and irdy still has not been asserted" and the consequent is "irdy is asserted." Unless this antecedent condition happens during the simulation, compliance with this property cannot be completely known. For a simulation run which has triggered only 10% of the antecedents, only 10% of the properties have been checked for the implementation. In this sense, the number of antecedents fired during a simulation run is a rough coverage metric.

There is one major drawback to using this metric for coverage. The problem is intimately related to the general relationship between implementation and specification. By the process of design, for every state, a designer chooses an action from the *choices offered by the specification* to create an implementation. As a result, the implementation will not cover the full range of behavior allowed by the specification. Thus, some of the antecedents in the specification will never be true because the implementation precludes any paths to such a state. Unless the verification engineer is familiar with the implementation design, he cannot know whether an antecedent has been missed because of the lack of appropriate simulation vectors or because it is structurally impossible. The only solution, in short of model checking the design, is to understand the design well.

6.7.2 Deriving Biases for Missed Corner Cases

To reach interesting corner cases, verification engineers often apply biasing to input generation. If problematic states are caused by certain inputs being true often, the engineer programs the random input generator to set the variable true *n*% instead of the neutral 50% of the time. For example, to verify how a component reacts to an environment which delays its response, *env_response*, the engineer can set the biasing so that the input, *env_response*, is true only 5% of the time. 0% should not be used because it may cause the interface to deadlock. With prevailing methods, the user needs to provide the biasing numbers to the random input generator. This requires expert knowledge of

the design, and the biases must be determined by hand. In contrast, by targeting antecedents, interesting biasing can be derived automatically. The algorithm works as follows:

1. Gather the properties that specify the outputs of the component to be verified. The goal: the antecedents of these properties should all become true during the simulation runs.

2. Set biases for all input signals to neutral (50% true) in the input generator described in section 6.6.

3. Run the simulation for some number of cycles.

4. Determine which antecedents have not fired so far.

5. Pick one missed antecedent, and use it to determine the variable biasing. If, for example, antecedent $\neg a \wedge b \wedge \neg c$ has not been true, set the following biases: a is true seldom (2% of the time), b is true often (98%), c is true seldom (2%).

6. Re-run the simulation and repeat from step 4. Continue until all antecedents have been considered.

There are a number of interesting conclusions. First, although effort was invested in determining optimal bias numbers exactly, biases that simply allowed a signal to be true (or false) "often" was sufficient. Empirically, interpreting "often" as 49 out of 50 times (98%) seems to work well. Second, an antecedent expression contains not only interface signal variables but also counter values and other variables that cannot be skewed directly. Just skewing the input variables in the antecedent is primary biasing, and a more refined, secondary biasing can be done by dependency analysis. This was done manually. For example, many hard-to-reach cases are states where a counter has reached a high value, and by dependency analysis, biases that will allow a counter to increment frequently without resetting were determined.

Time-Unbounded Properties

Recall the time-unbounded properties (section 5.1.2) in the Itanium™ specification. These properties need to be treated differently to determine coverage targets. The time-unbounded properties are in the form "if the trigger condition is true, the event *may* happen" unlike the time-bounded properties, which follow the form "if the trigger condition is true, the event *must* happen. Logically, these time-unbounded properties are expressed as:

$$\neg prev(q) \rightarrow \neg r$$

"If q is true in the previous cycle, then r *may* be true in the current cycle."

Note that with protocols in general, there is a *positive polarity*: the event r or trigger state q happening (i.e. the expression becoming true) is more interesting and significant than states where the expressions are false. Thus, taking this positive polarity into account, the property can be re-written in the following form. (Note that the two expressions are logically equivalent.)

$$r \rightarrow prev(q)$$

"If r is true in the current cycle, then it *must* be the case that q was true in the previous cycle." Thus, when an implementation does event r during simulation, the trigger condition q is checked for in the previous state. And so, only when r has become true during simulation, can it be verified that the implementation obeys the property. Consequently, the *consequent* expression r, and not the antecedent expression q, is the appropriate coverage target for time-unbounded properties.

6.7.3 Implementing Biasing

The actual skewing of the input variables is done during the BDD traversal stage of the input generation. The technique was introduced by Yuan *et al.* in [**?**] although it has been slightly modified

here. After the input formula BDD for a component has been built, the structure is traversed according to the biases. If variable *b* is biased to be true 49 out of 50 times, the TRUE branch from it is taken 49 out of 50 times (Figure 6.9). Likewise, if *b* is biased to be false 49 out of 50 times, the FALSE branch will be taken with that probability.

If the choice of branching (for example the THEN branch because the variable is biased to be true often) forces the expression to evaluate to false (i.e. the traversal inevitably leads to the "ZERO" leaf), the algorithm will backtrack and the other branch (the ELSE branch in this case) will be taken. As a result, even if *b* is biased to be true 49 out of 50 occurrences, the protocol logic can force *b* to be false most of the time. What is guaranteed by the biasing scheme is that whenever *b* is allowed to be true by the constraint, it will most likely be true.

An extra step is added to the input generation algorithm to accommodate the biasing. Often, for a single simulation run targeting a specific antecedent, only a few input variables are biased. The variables need to be re-ordered so that the biased variables are at the top of the BDD, and their truth value are not determined by the other variables. In Figure 6.10, variable *c* is intended to be true most of the time. However, since *c* is buried towards the bottom of the BDD, if $\{a = 0, b = 1\}$ is chosen, *c* is forced to be false to satisfy the constraint. In contrast, if *c* is at the top of the BDD, the true branch can be taken as long as the other variables are set accordingly (for example, $a = 1$). Fortunately, since the number of BDD variables is kept small, reordering for this purpose does not lead to BDD blowup problems.

Compared to the biasing technique used in SimGen, the biasing used in this framework is coarse. With SimGen, branching probabilities *are calculated* from the desired biases taking into account variable ordering. The calculations are based on chains of conditional probabilities. In contrast, this method directly uses the desired biasing as the branching probabilities; it requires no calculations and compensates for possible distortions by reordering. Although implementing the SimGen calculations is not difficult, the advantages of achieving more precise biasing are not clear from the examples attempted.

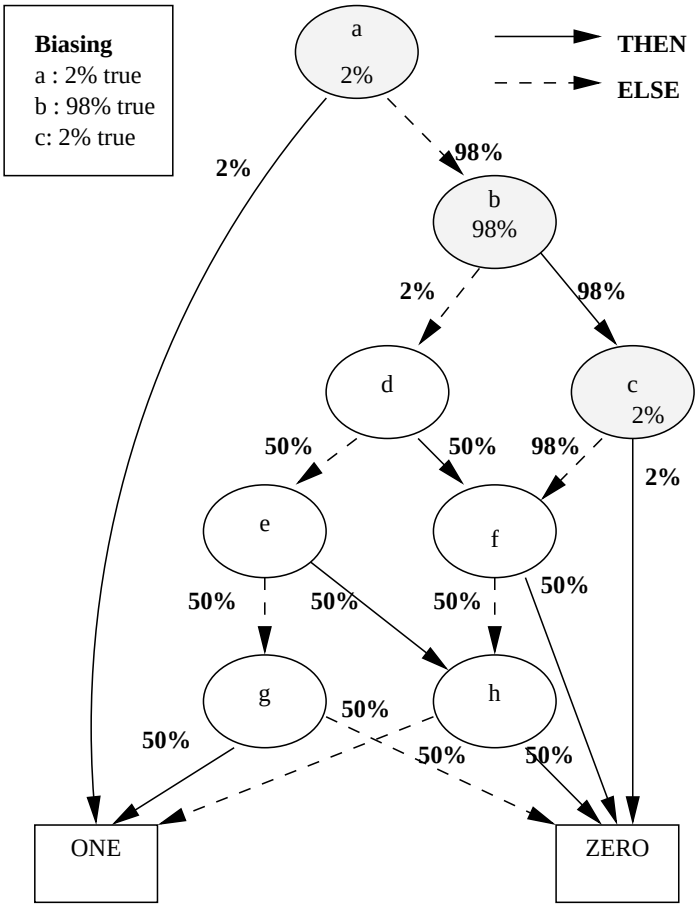


Figure 6.9: The Biased BDD Traversal

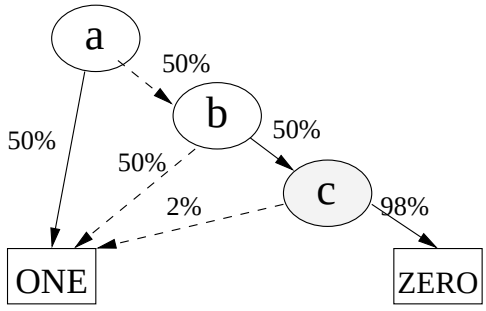


Figure 6.10: Incorrect Ordering

6.8 Experimental Results

To demonstrate the methodology on a meaningful design, we chose the I/O component from the Stanford FLASH [**?**] project for verification. The I/O unit, along with the rest of the project, had been extensively debugged, fabricated, and tested and is part of a working system in operation. The methods are evaluated on the PCI interface of the component.

The design is described by 8000 lines of Verilog and contains 283 variables which range from 1-bit to 32-bit variables: a complexity which renders straightforward model checking unsuitable. Approximate model checking was used by Govindaraju et al [**?**] to verify this design but no bugs were found because the design inputs were overly constrained and only a subset of the state space was explored.

The Setup A formal PCI specification was used to constrain the inputs and check the outputs at the PCI interface of the design. A simulation checker that flags PCI protocol violations was generated from the specification using a compiler tool written in OCAML [**?**]. The input generator, which controls the PCI interface inputs of the I/O unit, was also created from the specification via a compiler-based tool. The I/O unit (the design under verification), checker, and input generator are connected and simulated together, and results are viewed using the VCD (Value Change Dump) file. The inputs were skewed with different biases for each simulation run in order to produce various extreme environments and stress the I/O unit.

Verification Results Using the 70 assertions provided by the interface specification, nine previously unreported bugs were found in the I/O unit (Table 6.1). Most are due to incorrect state machine design. For example, one bug manifested itself by violating the protocol constraint, "once *trdy* has been asserted, it must stay asserted until the completion of a data phase." Because of an incorrect path in the state machine, in some cases, the design would assert *trdy* and then, before the completion of the data phase, deassert *trdy*. This can deadlock the bus if the counterparty infinitely waits for the assertion of *trdy*. The bug was easily corrected by removing the problematic and most likely unintended path.

The Violated Protocol Rule	Details
The design drives the address bus during idle.	The idle state is meant to be used as a turnaround cycle (when the bus is not driven by any agents) for the address bus.
The design drives the C/BE bus during idle.	The idle state is meant to be used as a turnaround cycle (when the bus is not driven by any agents) for the C/BE bus.
Under certain scenarios (the five described in the right column), the design does not keep the signal *trdy* (target ready) stable during a data phase. This violation breaks the basic handshaking protocol which requires *trdy* to be kept asserted (once asserted) until the data phase completes.	1. In the design, the boolean variable *doneDma* is true when the transaction has finished. However, there is a scenario where the final data phase has not completed but *doneDma* becomes true anyway (due to incomplete branching behavior description). This causes *trdy* to deassert prematurely because the design thinks that the transaction has completed. 2. The design intends to assert *stop* (the data *cannot* be transferred) but prematurely asserts *trdy* (the data *can* be transferred) and then immediately deasserts it so that *stop* can be asserted. 3. For certain sequences, the design sets the *trdy* signal to be equal to a boolean value which is true when there is new data waiting at the internal interface. In some cases, this causes the *trdy* value to be prematurely reset when there is no more data waiting. 4. The design deasserts *trdy* before data completion for certain cases of target aborted terminations. 5. The design has an incorrect and unexpected path (too complicated to explain) through the target state machine that prematurely deasserts the value of *trdy*.

Table 6.1: A Description of Some of the Bugs Found in the FLASH I/O Design

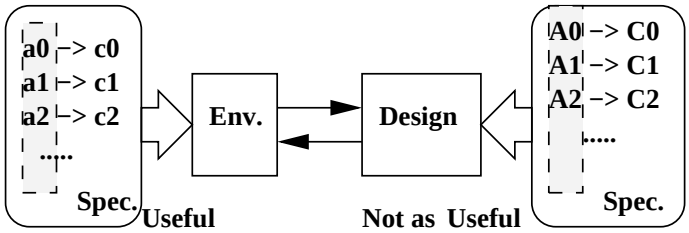


Figure 6.11: The Two Types of Simulation Coverage Metric and their Effectiveness

The setup makes the verification process much easier; the process of *finding* signal-level bugs is now nearly automated[1], and so, most of the effort can focus on *reasoning* about the bug once it is found.

Coverage Results Because the FLASH PCI design is conservative and implements a very small subset of the specification, basing the coverage metric on the *specification of the component* was not very illuminating. For example, the design only initiates single data phase transactions, and never initiates multiple data phase transactions; or, instead of having the flexibility to respond with any of three modes when it cannot handle a request, it always responds with the same mode. Thus, most of the antecedents remained false because the component never does many of the actions allowed by the specification.

However, using the metric to ensure that the environment is maximally flexible proved to be much more powerful. The motivation is to ensure that the design is compatible with any component that complies with the interface protocol. The design should be stimulated with the most general set of inputs, and so, using the missed antecedents from the constraints that *specify the environment* (in Figure 6.11, "a0, a1, ...") to determine biases was extremely fruitful; most of the design bugs were unearthed with these biasings.

Performance Results Performance issues, such as speed and memory usage, did not pose problems, and so, we were free to focus on generating interesting simulation inputs. However, to

[1] A caveat: the I/O unit has two-sides, an external PCI interface and an internal interface that interacts with the main processing core. This internal interface is not well documented, and thus its protocol was reverse-engineered by manually studying the design. Once this protocol was understood, properties were used to constrain the inputs on this interface.

	Number
Boolean Vars in Spec	161
Boolean Vars in BDD	15
Constraints on Env	63
Assertions on Design	70
Peak Nodes in BDD	193
BDD Memory Use	4 Mbytes
Bugs Found in Design	9

Table 6.2: Experimental Results: Input Generation for the FLASH PCI Design

Settings	User Time	System Time	Total
Random	0.53s	0.11s	0.64s
Constrained	0.77s	0.23s	1.00s
with Dump	0.77s	0.26s	1.03s
with Monitor	1.33s	0.29s	1.62s
with Coverage	1.54s	0.25s	1.79s

(for 12000 simulator time steps)

Table 6.3: Time Performance: Input Generation for the FLASH PCI Design

demonstrate the scalability of the method for larger designs, performance results were tabulated. The simulations were run on a 4-Processor Sun Ultra SPARC-II 296 MHz System with 1.28Gbytes of main memory. The specification provided 63 constraints to model the environment. The BDDs used for signal generation were very small; the peak number of nodes during simulation was 193, and the peak amount of memory used was 4Mbytes.

Furthermore, speed was only slightly sacrificed in order to achieve this space efficiency. The execution times for different settings are listed in Table 2. With no constraint solving, where inputs are randomly set, the simulation takes 0.64s for 12,000 simulator time steps. If the input generator is used, the execution time increased by 57% to 1.00s; this is not a debilitating increase, and now the inputs are guaranteed to be correct. The table also indicates how progressively adding signal value dumps, a correctness checker module, and coverage monitor modules, adds to the execution time.

Chapter 7

Conclusion

7.1 Impact on Verification in Practice

In the “Introduction” chapter, it was argued that this thesis work makes verification of large designs more feasible by researching how available verification machinery can be used optimally. This statement can be interpreted at many different levels. First, by focusing verification efforts on the interfaces of the components in the design, numerous problems can be expected to be found because the interface is arguably the weakest link. This thesis’s emphasis on interfaces is partly driven by the understanding that focus on problematic areas increases verification productivity. Second, with a solid understanding of the required behavior at the interface, components can be verified separately from each other. Realistically, only with a formal interface specification can the design be verified modularly. The ability to decompose a design and verify smaller parts is crucial for designs in practice.

Third, “checking engines” such as model checkers and satisfiability procedures are mostly hampered in practice by complexity and size issues. Since interface protocols have much limited complexity compared to design implementations, employing the checking tools on an interface specification is an optimal use of the tools.

Finally, because of the stylistic structuring of specifications proposed in this thesis, the value of

the specification as a tool is maximized. The creation of verification aids such as input sequences, checker properties, and coverage metrics can now be done in minutes instead of weeks. Due to complexity issues, most designs can only be simulated and not model checked. These verification aids are indispensable for simulation-based verification, and so, the thesis results have a direct impact on the ease and cost of verification in practice. In some cases, the technique in this work may allow a design that cannot be simulated with input constraints (because the solving of the constraints is too prohibitive) to be simulated with the same constraints.

7.2 Summary of the Methodology

The methodology proposed by this thesis is summarized as a verification flow (Figure 7.1). First, when the interface protocol is designed, it should be documented as a formal property-based specification. Then, this specification can be formally and informally verified to ensure the correctness of the protocol. This verification procedure can also check that the properties are not over-restrictive or under-restrictive when compared to what was intended. This is important later when the properties are used to verify a design implementation. As others need to learn and understand the protocol, the specification can be used to automatically generate example traces. Finally, when an implementation has been designed, the three verification aids can be generated from the specification so that the design can be functionally verified via simulation. Because the specification has been formally verified, the derivatives (the verification aids) are much less likely to cause false negatives (errors in the design are not caught) and false positives (an error is flagged but the design is correct and the error is actually in the verification setup). And if the design requires emulation or gate-level simulation, because the output monitor is synthesizable, it can be used in these frameworks as well.

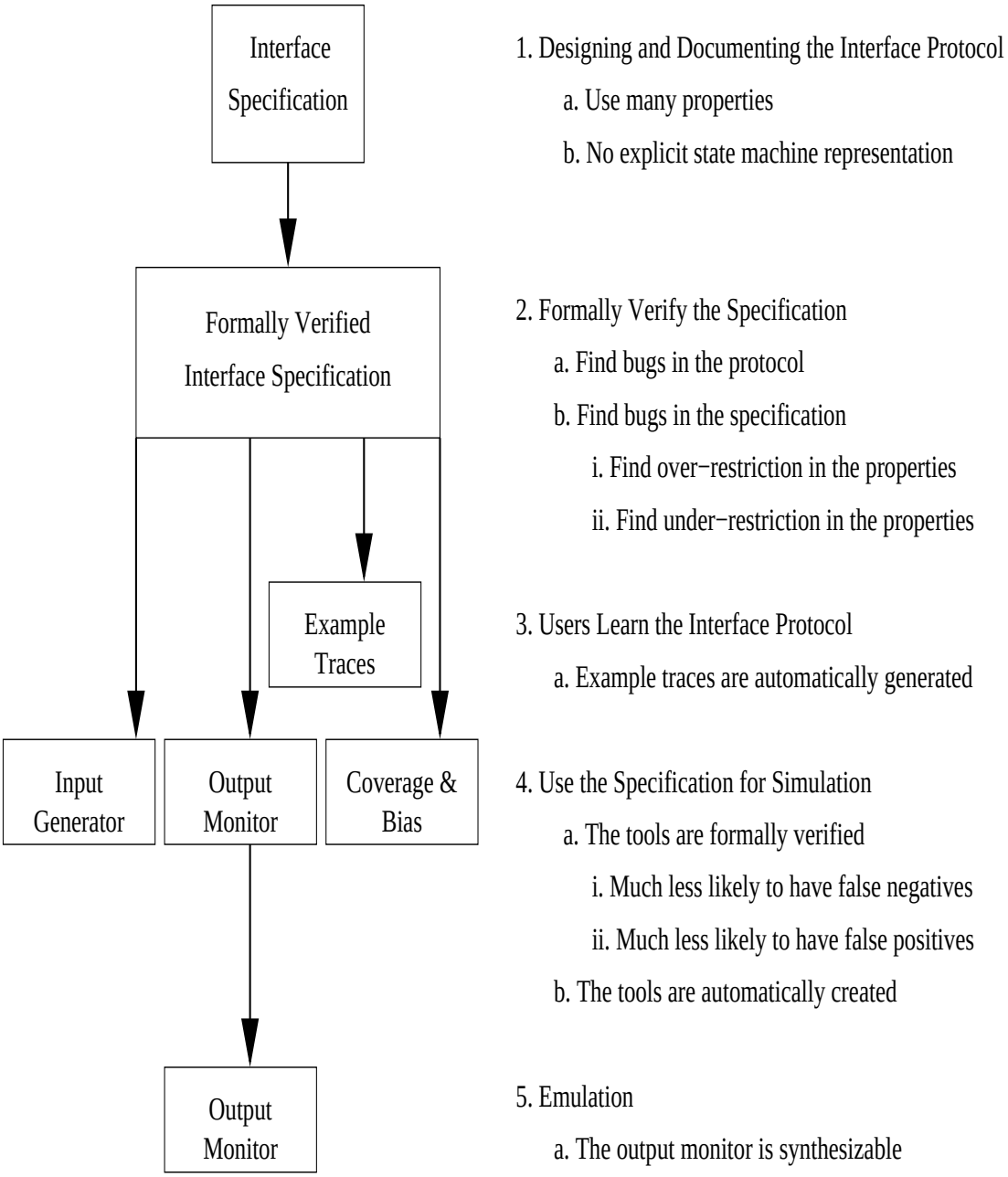


Figure 7.1: The Verification Flow

7.3 Future Work

Other Protocols An obvious next step for this line of work would be to attempt specifications of other interfaces, especially those with characteristics different from bus protocols, such as networking protocols. It is likely that different stylistic issues will arise, as well as complexity issues. It is also expected that simple data structures such as counters and set-reset state machines will not be enough, and more powerful data structures such as queues will be needed. However, it is believed that a property-based specification style can be retained as long as the appropriate data structures are used.

More Specification Debugging Tools Another direction of interest is to develop better tools for debugging the specification. For example, the dead state check is limited in that although it can return a state description of the dead state, it cannot pinpoint the conflicting properties. The problem is exacerbated by the fact that any number of properties can contribute to an unsatisfiable specification. A greedy algorithm using satisfiability on the properties such that the conflicting requirements can be found, should be developed for a more complete specification debugging tool environment.

Exploiting the Specification Exploiting a structured formal specification for other uses is also of interest. Perhaps incomplete designs can be automatically augmented by specification properties for simulation purposes. Or, useful synthesis information can be extracted from the specification. Additionally, better coverage metrics can probably be deduced from the specification. A straightforward extension would be to see whether pairs (cross products) of antecedents become true during simulations. Although some pairs may never be true, aiming for them may push the design into interesting scenarios. A fundamentally different approach will probably be developed as future work as well.

Experiments to determine whether designs that are too large for SimGen-type algorithms (where BDDs are built statically) can be handled by this thesis's algorithm (where BDDs are built dynamically) would further validate the methodology's memory efficiency. In practice, there are designs

so large that memory blow-up problems would prevent inputs from being generated at all. The power of the methodology can be illustrated with such examples. Furthermore, more extensive experiments to quantify the speed penalty for the dynamic BDD building would be useful for many. Speed is a key issue in simulation runs since designs have become so large. Since in academia access to large designs is limited, it is difficult to conduct experiments in order to determine the true speed penalty of the algorithm. However, perhaps in industry, the speed penalty can be benchmarked more meaningfully.

Chapter 8

Appendix A

A proof of the Theorem stated in section 3.6.3.

Proof: The monitor, M, can be characterized as follows.

Inputs : $I_M = I \times O$

Outputs : $O_M = correct_{Me} \times correct_{Env}$

Start State : $q_M^0 \in Q_M$

States : Q_M

Transition function, $\delta_M : I \times O \times Q_M \rightarrow Q_M$

Output function, $\lambda_M^{me} : I \times Q_M \rightarrow \{true, false\}$

Output function, $\lambda_M^{env} : O \times Q_M \rightarrow \{true, false\}$

The states are characterized by vectors of internal history variables. Thus, if $h_0, h_1, ..., h_n$ are the history variables, state q_M^i of Q_M is described by the values of $(h_0^i, h_1^i, ..., h_n^i)$. The monitor has the following four properties.

Property 1. *If the agent commits an illegal action while the environment is still correct, the monitor enters and stays in a state where no matter what the observed signals are, $correct_{Me}$ is false and $correct_{Env}$ is true. Furthermore, this state is entered only when this particular event happens. "Once the agent makes a mistake, the environment wins."*

For $\forall i \in I, \forall o \in O, \forall q \in Q$,

$$\lambda_M^{me}(o,q) = false \wedge \lambda_M^{env}(i,q) = true$$
$$\iff \delta_M(i,o,q) = q_M^{meError}$$

and for $q_M^{meError}$, for $\forall i \in I, \forall o \in O$,

$$\lambda_M^{me}(o, q_M^{meError}) = false,$$
$$\lambda_M^{env}(i, q_M^{meError}) = true$$

Thus, consequently, for $\forall i \in I, \forall o \in O$,

$$\delta_M(i,o,q_M^{meError}) = q_M^{meError}$$

Property 2. *"Once the environment makes a mistake, the agent wins."*

For $\forall i \in I, \forall o \in O, \forall q \in Q$,

$$\lambda_M^{env}(i,q) = false \wedge \lambda_M^{me}(o,q) = true$$
$$\iff \delta_M(i,o,q) = q_M^{envError}$$

and for $q_M^{meError}$, for $\forall i \in I, \forall o \in O$,

$$\lambda_M^{env}(i, q_M^{envError}) = false,$$
$$\lambda_M^{me}(o, q_M^{envError}) = true$$

Thus, consequently, for $\forall i \in I, \forall o \in O$,

$$\delta_M(i,o,q_M^{envError}) = q_M^{envError}$$

Property 3. *When both the agent and the environment make a mistake at the same time.*

For $\forall i \in I, \forall o \in O, \forall q \in Q$,

$$\lambda_M^{env}(i,q) = false \wedge \lambda_M^{me}(o,q) = false$$
$$\iff \delta_M(i,o,q) = q_M^{bothError}$$

and for $q_M^{meError}$, for $\forall i \in I, \forall o \in O$,

$$\lambda_M^{env}(i, q_M^{bothError}) = false,$$
$$\lambda_M^{me}(o, q_M^{bothError}) = false$$

Thus, consequently, for $\forall i \in I, \forall o \in O$,

$\delta_M(i, o, q_M^{bothError}) = q_M^{bothError}$

Property 4. *The monitor does not cause "dead states" for the agent. "As long as the agent has been correct so far, there always exists a legal action for the agent.*

For $\forall q \in Q - \{q_M^{meError}, q_M^{bothError}\}$,

$\exists o \in O such\ that\ \lambda_M^{me}(o, q) = true$

Given such a specification M, the construction of the implementation, K, can be done as follows. Let the characterization of Moore machine K be

Inputs : I Outputs : O

Start State : q^0 States : Q

Transition function, $\delta : I \times Q \rightarrow Q$

State labeling function, $\lambda : Q \rightarrow O$

Let Q be of the same type as Q_M. Thus, if h_0, h_1, ..., h_n are the history variables of the monitor, K has the same history variables and state q^i of Q is described by the values of $(h_0^i, h_1^i, ..., h_n^i)$.

Construction

Basic Idea : Start with the same initial state as the monitor. Use the monitor's correct functions, λ_M^{me} to determine the outputs at each state. The transitions are determined by the inputs, outputs, and the current state using the monitor's transition function.

Step 0. Initially, Q is empty. $Q = \emptyset$

Step 1. Let initial state q^0 = q_M^0. Add q^0 to Q.

$Q = \{q^0\}$.

Step 2. For q^0, $\exists o_0$ such that, $\lambda_M^{me}(o_0, q^0) = \lambda_M^{me}(o_0, q_M^0) = true$. (because of property 4 of the monitor, step 1 in the construction, and $q_M^0 \neq q_M^{meError}, q_M^{bothError}$.) Let $\lambda(q^0) = o_0$ and define transitions for all possible inputs. $\delta(i, q^0) = \delta_M(i, o_0, q^0)$ for $\forall i \in I$. If $\delta(i, q^0)$ is not in Q, add to Q.

Any new addition to Q, q', is guaranteed to be a member of Q_M because

i) $q^0 \in Q_M$.

ii) if $q \in Q_M$ and $q' = \delta(i,q)$, for some $i \in I$, then $q' \in Q_M$ because $q' = \delta_M(i,o,q)$ for some $o \in O$ and $\delta_M(i,o,q)$ is defined for all $i \in I$ and $o \in O$ for $q \in Q_M$.

Thus, by induction on i) and ii), for $\forall q, q \in Q \Rightarrow q \in Q_M$. [property a]

Furthermore, because $\delta_M(i,o,q) = q_M^{meError}$, $q_M^{bothError}$ only for o such that $\lambda_M^{me}(o,q) = false$ (due to property 1 and 3 of the monitor), for any new addition to Q, q', $q' \neq q_M^{meError}, q_M^{bothError}$ [property b]

3. For every newly added $q \in Q$, pick $o \in O$ such that $\lambda_M^{me}(o,q) = true$ and let $\lambda(q) = o$. (because of property 4, property a, and property b, it is guaranteed that such a o exists.) and for every $i \in I$, define $\delta(i,q)$ such that $\delta(i,q) = \delta_M(i,o,q)$ and if $\delta(i,q)$ is not in Q, add to Q.

4. Iterate back to step 3 and stop when no new q is added to Q.

Thus,

- $Q \subset Q_M$
- $q_M^{meError}, q_M^{bothError} \notin Q$
- $\forall q \in Q, \lambda(q) = o \Rightarrow \lambda_M^{me}(o,q) = true$

And so, K implements the specification M.