Specifying Properties for Generalized Symbolic Trajectory Evaluation

Edward Smith

Worcester College

University of Oxford

Submitted in partial fulfillment for the Degree of

Doctor of Philosophy

March 2008

To my parents

Specifying Properties for Generalized Symbolic Trajectory Evaluation

Edward Smith, Worcester College
Submitted for DPhil. Computer Science, Hillary Term 2008

Abstract

This dissertation proposes a two-layer approach for formal hardware verification using symbolic ternary simulation of gate-level circuit models. At the core of our approach we follow the methodology of generalized symbolic trajectory evaluation (GSTE), which has shown considerable promise for the verification of micro-processor components. Properties for GSTE have traditionally been expressed using diagrams called *assertion graphs*, but the graphical nature of these places limitations on formal reasoning and scalability. We recast GSTE techniques into a clean general logical framework for simulation, and use this framework to explore and characterize important verification steps.

We introduce *generalized trajectory logic* (GTL), a low-level temporal logic that provides a textual formal basis for specifying and reasoning about symbolic ternary simulations. We describe model checking for this logic and derive clean rules for property equivalence, decomposition and abstraction refinement. We then introduce *assertion programs*, which describe abstract specifications as high-level executable models. We show how term-rewriting based on weakest preconditions can be used to generate simulations that verify that a circuit refines an assertion program. Expressing these simulations using GTL, we show how they can be dynamically transformed during the generation process, to create particular schemes of model checking abstraction. We apply the entire verification framework to a first-in-first-out buffer and a micro-operation scheduler.

Acknowledgements

Tom Melham has been an invaluable source of advice and support throughout the entire scope of this work. Above all, Tom has provided me with the required perspective and insight to keep me focused on my aims and understand how everything fits together. His comments have always been thoughtful and considerate, and his passion for style and presentation has been inspiring. I thank Tom for the opportunity to work on this project, as well as for managing it so effectively.

I am grateful to Intel Corporation for generously funding this work, as well as hosting me for six months inside their Strategic CAD Labs. Jin Yang's expertise proved a significant guiding force in the development of my work, and I greatly enjoyed our many informative and fruitful discussions. I would like to thank John O'Leary and Noel Menezes for overseeing my stay, as well as Jim Grundy, Amit Goel, and Sava Krstić at Intel, for their support and friendship.

I am also lucky to have had a great deal of support at Oxford. I have had the pleasure of many interesting discussions with my research group (especially with Sara Adams, Magnus Björk, Ashish Darbari, Ziyad Hanna, and Carl Seger). I am very grateful to Orna Grumberg, Marta Kwiatkowska, Joël Ouaknine, and Mike Spivey for their useful feedback during my examinations. My friends, workmates, and fellow laboratory attic inhabitants (among others, Daniel Goodman, Bruno Oliveira, Zoltán Miklós, and Rui Zhang) have provided welcome relief with motivating and refreshing breaks. I would also like to thank Worcester College and Oxford University Computer Laboratory for their accommodating environments and their help with travel costs.

Finally, I would like to thank my parents and my sister, for their love and encouragement over the years, and Katharina, who has been there for me throughout this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we provide the context and motivation for developing specification notations for formal hardware verification by symbolic ternary simulation. Following an overview of general verification techniques, we introduce symbolic ternary simulation, and the specification problems that we aim to address. We then summarize our contributions and give an outline of the dissertation.

1.1 Verification

Digital systems are increasing in usage, providing unparalleled degrees of convenience and automation. They are responsible for performing many critical tasks, from real-time flight and automotive control, to secure and reliable transmission of sensitive communications. But as well as growing in number and responsibility, these systems are also tending to expand in design complexity, fueled by advances in digital processing and storage, and by the connectedness offered by global communications networks. This complexity is difficult to manage, often leaving the doors open to potential design flaws. To guard against these risks, there is a strong need for *verification* methodologies, which demonstrate that the specification of a system is in fact met by its design.

The most common form of verification is *simulation*. Simulation uses an executable model of the design to compute how it will respond to certain environmental scenarios. As well encompassing simple interactive debugging procedures, verification by simulation can also describe bulk batch attempts to find bugs by systematically covering as many system states as possible. A wide range of tools exist to structure and select such simulation sets, by maximizing the exploration of significantly distinct design states. But despite applying huge numbers of simulations, bugs are often missed, simply due to the size of the state-spaces at hand. For the Intel Pentium 4 design, tens of bugs were still undetected after a set of over 200 billion simulation cycles, requiring the capacity of several thousand machines, operating for many months [Ben01].

1.2 Formal Verification

Formal verification uses rigorous mathematical foundations to reliably deduce properties about how systems behave over ranges of execution conditions. There are many different approaches to formal verification, each suited to the particular qualities of the properties or systems under verification.

Symbolic simulation has been in use since the late 1970s [CJB79]. It extends traditional simulation to wider input ranges, by using symbols called variables to represent arbitrary unknown, but fixed, values. Expressions containing these variables are associated with components of the system, so that the overall state is represented in a symbolic encoding. This allows standard simulation to be generalized over the range of different operating conditions parameterized by these variables. The capacity of symbolic simulation was greatly enhanced by the introduction of Binary Decision Diagrams (BDDs) [Bry92] which provide particularly efficient representations for common Boolean functions. Despite this, symbolic simulation is still limited in capacity, as well as in its linear and bounded nature.

Model checking [JGP99] is a different approach, developed independently by Clarke and Emerson [CE81] and Quielle and Sifakis [QS82]. In model checking, the validity of some property, typically a formula of temporal logic [Eme90], is systematically and automatically checked with respect to a model derived from the system under analysis. When the property is found not to hold, an illustrative counterexample is typically generated. Different forms of specification and model checking algorithms exist, for different classes of models and properties. Unfortunately, standard model checking techniques are limited to those systems with state-spaces that are simple enough to be automatically explored. This is compounded by the *state explosion problem,* which says that the number of states of a system can, and often does, increase exponentially with the size of the design.

Several means of avoiding this problem have been developed. One approach is to use special data structures and algorithms for the analysis. As with simulation, the capacity of model checking techniques can be greatly enhanced with the use of symbolic techniques. Symbolic model checking [BCMD90, McM92] encodes both the sets of model states, as well as the model's transition relation, as Boolean formulas represented by BDDs. Another approach is to use BDD-based symbolic simulation for direct state-set image computation [CBM90]. Operations for BDDs are fast, and their space requirements decrease with the regularity of the predicates they describe. Since most designs tend to involve a reasonable degree of symmetry and regularity, the resulting state-set representations can often be kept fairly compact.

Another option for extending capacity is to make use of *abstraction*—the process of losing selected pieces of information that are not relevant to the task at hand. Abstraction is often achieved using sound simplifications of the design model. Other approaches discard information dynamically during the model checking algorithm itself. If too much information is lost, known as *over-abstraction*, then the resulting reduction in precision can lead to verification failure. Typically, a feedback loop of iterative *abstraction refinement* is used to reach a suitable level of abstraction, where there is a middle-ground between loss and retention of information. This process is often automated, although a degree of user interaction can sometimes prove critical to aid the selection of relevant information.

Theorem proving [GM93] is a type of formal verification that allows for a high degree of manual interactive involvement. Properties are expressed as terms in a logic, and deductive reasoning rules are progressively applied to determine term validity. If required, the user can gain complete control over the rules applied at each step. Since the terms of the logic can be considered to themselves be abstractions, theorem proving can therefore be viewed as the ultimate in interactive abstraction control. But due to the high degree of guidance, skill and time generally required for success, theorem proving can often be prohibitively demanding. As a result, there have been many attempts to mix theorem proving with other, more automated, techniques, to attain an efficient balance between the benefits of human intuition and those of automated deduction. One example of this is the use of interactive theorem provers to decompose large model checking problems into smaller ones. Here the user need only be concerned with the high-level insights that are key to surpassing the main capacity barriers.

1.3 Symbolic Ternary Simulation

Verification is of particular importance for microprocessor design, because of the importance that microprocessors play in the foundations of many other products, and the high cost of any necessary product recalls. As the number of transistors per processor continues to rise exponentially, this extra computation capacity leads to additional functional design complexity, enabling increasingly out-of-order, speculative machines with deeper pipelines, and advanced features, such as Hyper-Threading and multi-core architectures. This dissertation is based on formal hardware verification using *symbolic ternary simulation*, a technique that has shown success in verifying components of next-generation microprocessors beyond the reach of standard model checking techniques [Sch03].

Symbolic ternary simulation lies at the intersection of symbolic model checking, symbolic simulation and abstraction techniques. As with symbolic model checking, BDDs are used to explore the relationship between model and property states. Unlike traditional symbolic model checking, however, post-images are calculated using a form of symbolic abstract simulation that takes place directly on a low-level description of the circuit design. This eliminates the need for building a complete model transition relation, which is prohibitive for many industrial designs.

Symbolic ternary simulation is fundamentally distinguished from many other standard forms of model checking by the abstract representation that it uses for sets of circuit states. Abstraction is introduced by adding the additional simulation value X, which denotes *unknown* circuit charge. This results in what is effectively *partial* circuit simulation. One key advantage of this approach is that each particular set of circuit states can be represented by a *range* of different approximations. The less precise an approximation is, the less space it consumes in the model checker. Therefore an abstraction balance can often be achieved between losing too much information and requiring too much space. In order to do this, user feedback can direct the model checker on how to pick the best representatives for a given property.

There are several different flavours of model checking based on symbolic ternary simulation. We look at each of these in detail in Chapter 2. The most basic form, Symbolic Trajectory Evaluation (STE) [BBS91], uses single simulation sequences to verify properties written in a small linear temporal logic. The length of the traces checked by STE match the length of the abstract simulation itself, so properties are limited to being bounded in nature. *Generalized symbolic trajectory evaluation* (GSTE) [YS02] overcomes this limitation, by using fixed-points to simulate unbounded iterative behaviours. *Compositional GSTE* (CGSTE) [YS04] extends this approach by providing a mechanism for decomposing a run into separate concurrent blocks of simulation.

Properties for (C)GSTE are expressed using *assertion graphs* [YS02], which are graphical structures that resemble a variety of universal-automata [HCY03b]. Each path in an assertion graph represents a linear assertion made about which execution traces are allowable. These graphs are directly traversed by the model checking algorithms, so their shape can directly affect the simulation strategy. In order to control abstraction, therefore, manual transformations are applied directly to the graph structures themselves.

Although the graphical nature of assertion graphs can be useful for displaying simple properties in a visual manner, it places serious limitations on formal reasoning. Experience from from STE [AJS98, SJO$^+$05] has shown the benefits of formal reasoning for managing high-level verification steps such as property decomposition. In (C)GSTE, abstraction refinement also requires additional justification. Although some progress has been made on reasoning with assertion graphs [HCY03b, YYHS05], the resulting rules tend to be complicated by the graphical nature of the properties. Cleaner reasoning requires a property representation that is more structured and controlled.

1.4 Contributions

This dissertation describes a clean formal framework for verification using symbolic ternary simulation. The framework is split into two layers: a low-level temporal logic layer, for describing simulation structures, and a high-level synchronous programming language, for specifying behavioural models. The temporal logic and its associated reasoning rules have been published in *A Logic for GSTE* [Smi07]. The language and methodology for our high-level language is based on our publication *A Method for Generation of GSTE Assertion Graphs* [Smi05]. We will now summarize the contributions that we make.

Generalized Trajectory Logic We introduce *generalized trajectory logic* (GTL) as a linear temporal logic for expressing symbolic ternary simulations. The structure and semantics of GTL unifies the property notations of STE, GSTE and CGSTE, by aligning itself with the atomic simulation steps common to each. By operating at this finer level of atomicity, we expose underlying algebraic patterns that have not been otherwise apparent. This not only provides the fundamental logical characteristics necessary for formal reasoning, but also emphasizes the link to propositional logic, clarifying the nature of symbolic ternary simulation as a whole. Being a textual notation, GTL is furthermore easy to manipulate, express, and describe rules for.

Reasoning with GTL By examining the semantic characteristics of our logic, we develop a series of reasoning rules for applying practical verification management steps to simulations expressed using GTL. As with GSTE assertion graphs, the specifications describe both the property to be verified, as well as the verification approach. Equivalence rules can therefore be used to describe property-preserving transformations that optimize model checking or refine abstractions. We also introduce rules for property decomposition, and a rule for temporal induction.

Assertion Programs Since GTL describes simulations at a low level, it is not practical for expressing complex specifications manually. To surmount this problem, we introduce the synchronous language of *assertion programs* as a user-facing specification language. Assertion programs are used to express high-level reference models of the behaviour expected of a circuit. This approach allows specifications to be expressed using a familiar imperative programming style with high-level data-types, such as integers and lists. Unlike GTL, the notation also serves to separate property specification from model checking approach. The circuits and high-level models are connected via given *interface mappings* that form part of the specification.

Verifying Assertion Programs We provide a means to unify the semantics of GTL and assertion programs. We then use this to build a rule-based framework for generating simulation descriptions from assertion program specifications. The simulations produced verify that a circuit refines the given assertion program. Simulations are progressively built using symbolic backwards state-space exploration, based on weakest-precondition term-rewriting. By incorporating rules for abstraction and decomposition into the framework, users can control the resulting simulation approach through the choice and application of rules.

Case Studies We have implemented both GTL model checking and our simulation generation framework within the Forte [SJO^{+}05] verification environment, and used our implementation to verify a first-in-first-out buffer and a simple micro-operation scheduler.

1.4.1 Significance of Contributions

Our notation for simulations is at least as expressive as GSTE assertion graphs, so it enables the expression of GSTE simulation techniques. Since it is based around a formal logic with good algebraic properties, there are many rules for reasoning about the simulations that are otherwise difficult to describe for assertion graphs. Our work therefore shows significant promise for improving the outlook of symbolic ternary simulation, by fitting the successful techniques surrounding GSTE into a clean and manageable formal setting.

1.5 Outline

Chapter 2 provides an introduction to symbolic ternary simulation, explaining its history, theoretical foundations and usage methodology. We first describe the structure of our circuit models. Then we explain how regular simulation evolved via STE to GSTE and, more recently, to *compositional GSTE*, driven by demands for increased capacity and property expressibility. We provide an in-depth look at ternary abstraction and the GSTE specification notation of assertion graphs. We show how these graphs are used to control abstraction in practice.

Chapter 3 introduces generalized trajectory logic (GTL), starting with an introduction to the *symbolically indexed structures* that we use to model symbolic expressions. We provide a formal trace-based semantics of GTL, for which we demonstrate monotonicity and continuity. After introducing several shorthand notations, we describe and justify algorithms for both concrete and abstract model checking.

Chapter 4 describes a series of *reasoning rules* for GTL, illustrated with several small sample applications. The first section demonstrates simple equivalence rules for formulas, classified into Boolean, temporal and symbolic types. In the second section, rules for properties are considered, including several means of decomposition. The final section formalizes the notation of abstraction refinement for GTL, and expresses and classifies several common patterns of refinement previously used with GSTE.

Chapter 5 introduces the high-level modeling language of *assertion programs*. We first describe the refinement-based approach that we take to specification. Then, after providing an overview of the structure of assertion programs, we step through each language construct in turn, from variables and statements to the interface mappings that link the high-level models to the circuits. We then provide a formal semantics for the language, and a formal definition of satisfaction.

Chapter 6 provides a methodology for translating assertion programs into series of GTL properties that together verify refinement. We unify the semantics of assertion program expressions and formulas of GTL to allow reasoning between these two modes of specification. We extend GTL to a form called *vector GTL* which, although of a more complex form, scales better to large simulations. We then describe a series of compilation rules for building up vector GTL simulations based on the transition structure of an assertion program. We show how different applications of these rules results in different abstraction approaches during the resulting simulation runs.

In Chapter 7, we apply our verification methodology to two example circuits: a first-in-first-out (FIFO) buffer and a micro-operation scheduler. In each case, we introduce the circuit specification in English, describe an existing verification approach based on assertion graphs, and then describe our own approach based on assertion programs. At the end of the chapter we compare and contrast the two methodologies.

Finally, Chapter 8 provides some concluding remarks, and a discussion of potential future work.

Chapter 2

Generalized Symbolic Trajectory Evaluation

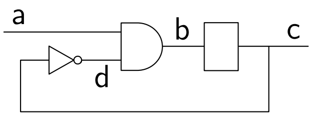
Generalized symbolic trajectory evaluation (GSTE) is a formal hardware verification technique lying at the boundaries of model checking, abstract interpretation, symbolic execution and circuit simulation. In this chapter, we explain its nature and history.

We first describe the gate-level circuit model used by the model checker, as well as a more abstract circuit model that is useful for reasoning theoretically about verification. We provide a brief introduction to standard binary circuit simulation, and then introduce *ternary simulation*. Ternary simulation extends the range of binary simulation by introducing a ‘don’t care’ value, denoted X into the simulation domain. By viewing the ternary simulation states as abstract representations of *sets* of binary states, we show how ternary simulation can be compared to other standard forms of model checking.

We then describe *Symbolic Trajectory Evaluation* (STE), which uses a *symbolic* form of ternary simulation for model checking. STE is sufficiently scalable to have found many applications in industry, but its range of application is limited to bounded properties, because of its roots in conventional simulation. This deficiency led to the development of *generalized* STE (GSTE), which extends STE to unbounded behaviours by introducing fixed-points in the simulation. We describe GSTE model checking, with its graphical specification notation called *assertion graphs*. Finally we describe a further generalization of GSTE, known as *compositional* GSTE that decomposes simulations of concurrent behaviour.

$$\begin{aligned} b &= a \wedge d \\ c' &= b \\ d &= \neg c \end{aligned}$$

(i) Circuit Netlist



(ii) Diagrammatic Representation

Figure 2.1: A Circuit Netlist

2.1 Circuit Models

We will describe two different forms of circuit model. The first of these models, the *circuit netlist*, is a gate-level circuit description used directly by the simulation tool. The second model, the circuit *Kripke Structure* [JGP99], is a more abstract model that is useful for *reasoning* about the verification. The types of circuits that we consider are synchronous systems, where the outputs at each time-step are functions both of the current state and the current circuit inputs, classing them as *reactive* [Hal98]. Each state in these models represents a particular stable charge configuration in the physical circuit. We use Boolean values, which are, in the hardware domain, conventionally written as *high*, 1, and *low*, 0, to model the charge that can be held at any one point in the circuit.

2.1.1 Netlists

The circuit *netlist* model, used by gate-level simulation-based techniques such as GSTE, is a collection of logical and stateful *gates* together with a description of their interconnections. Each gate is itself modeled using a Boolean *excitation function* that describes how its output behaves as a function in terms of its inputs. To allow for state-holding elements, this map can be a function of the *previous* inputs and output of the gate, as well as its current inputs. The gates are connected through a finite set of shared connections, given by the set of *circuit nodes*, $\mathcal{N}$.

Example An example circuit netlist is shown in Figure 2.1, together with its traditional graphical representation. The circuit nodes are labeled a, b, c and d. The excitation function for each node is listed as a Boolean expression, where b' represents the previous value of node b. The delay element, graphically represented as a rectangle, is a gate whose input, b, always matches the subsequent value of its output, written c'. Node a has no excitation function, and is therefore a circuit input.

Notice that netlist models are more structured than typical models used in formal verification, such as state-transition systems or automata. This is because they encode the topological layout of the physical circuit as well as its functionality. In this way, a netlist expresses a circuit model as a *product* of systems that each model one of the gates. This has benefits over a monolithic transition representation, as different parts of the circuit can be analyzed and simulated *independently*, and the model representation is kept concise.

2.1.2 Kripke Structures

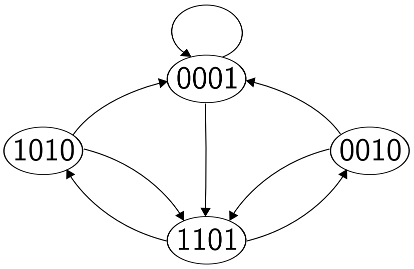
Although circuit netlists are the models used directly by simulation-based techniques, it is useful to have alternative abstract models for *reasoning* about such techniques. This is because the properties that we are concerned with typically only depend on the functionality of the circuit, not its topological layout. For this we model circuits as Kripke Structures [JGP99].

Definition 2.1.1 (Kripke Structure)**.** *A Kripke Structure is a pair $\mathcal{K} = (S, T)$, where S is a finite set of model* states*, and $T \subseteq S \times S$ is a total transition relation.*

The set of circuit states used throughout this dissertation, S, consists of those Boolean vectors $s \in \mathbb{B}^{\mathcal{N}}$ that are consistent with the constraints imposed by the circuit gates. We will write $s(\mathsf{n})$ for the value of node n in state s. The transition relation T will hold between those states where the constraints implied by the circuit's state elements are satisfied. We will regard the Kripke structure of the circuit under verification, $\mathcal{K}_{\mathbf{C}} = (S, T)$, to be a fixed constant throughout.

Example We can model the netlist shown in Figure 2.1 as the Kripke Structure shown in Figure 2.2. States are represented as four-bit vectors, written in the form abcd. The consistent states, S, are those states that satisfy $\mathsf{b} = \mathsf{a} \wedge \mathsf{d}$ and $\mathsf{d} = \neg\mathsf{c}$. The transition relation holds from state $s \in S$ to $s' \in S$ when $s(\mathsf{b}) = s'(\mathsf{c})$.

Kripke Structures contain no concept of initial state. This is in alignment with hardware systems in particular, where nothing is known about the state of the system until we start to interact with it. This differs from software, where state is typically initialized on allocation. Many hardware components do have some form of *reset*, to what might be regarded as an initial state. But many also do not, and reset logic, when it exists, is often sufficiently complex to require verification in its own right. Therefore resets are not a part of the circuit models for STE-based verification.



$$S = \{ \ \mathsf{0001}, \ \mathsf{0010}, \ \mathsf{1010}, \ \mathsf{1101} \ \}$$

$$\begin{aligned} T = \{ \ & (\mathsf{0001}, \mathsf{0001}), \ (\mathsf{0001}, \mathsf{1101}), \\ & (\mathsf{0010}, \mathsf{0001}), \ (\mathsf{0010}, \mathsf{1101}), \\ & (\mathsf{1010}, \mathsf{0001}), \ (\mathsf{1010}, \mathsf{1101}), \\ & (\mathsf{1101}, \mathsf{0010}), \ (\mathsf{1101}, \mathsf{1010}) \ \} \end{aligned}$$

Figure 2.2: A Kripke Structure

Definition 2.1.2 (Kripke Post-Image)**.** *For Kripke Structure $\mathcal{K} = (S, T)$, we define the post-image function, $\mathrm{post}_{\mathcal{K}} : 2^S \to 2^S$, to map a set of states to the set of its successors:* $\mathrm{post}_{\mathcal{K}}(R) = \{s' \in S \mid \exists s \in R \,.\, (s, s') \in T\}$.

Example For the Kripke Structure that represents our example circuit, the post-image of states in which b is 0 consists of those states in which c is 0:

$$\mathrm{post}(\{\mathsf{0001}, \mathsf{0010}, \mathsf{1010}\}) = \{\mathsf{1101}, \mathsf{0001}\}$$

This is what we would expect, seeing as there is a simple delay element that separates the two nodes.

We model the possible behaviours of our models as *traces*, which are non-empty finite words from S^+ corresponding to paths through the Kripke Structure.

Definition 2.1.3 (Kripke Traces)**.** *A finite non-empty sequence of states, $\sigma \in S^+$, is a trace of $\mathcal{K}$ if each step in the trace is a possible transition: $(\sigma_i, \sigma_{i+1}) \in T$ for* $0 \leq i < |\sigma| - 1$. *We will write* $\mathrm{tr}(\mathcal{K})$ *for the set of all traces of $\mathcal{K}$.*

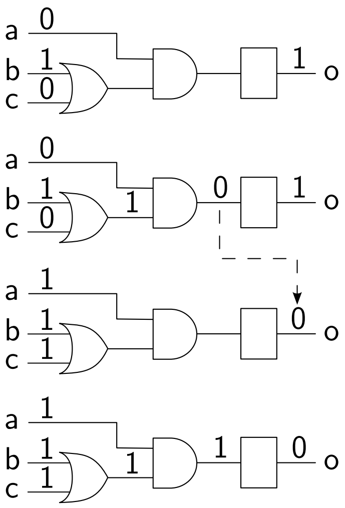
Throughout the dissertation we will use $\mathrm{last}(t)$ to refer to the last element in the sequence t, and $\mathrm{front}(t)$ to refer to the prefix consisting of t with its last element removed.

2.2 Binary Simulation

Binary circuit simulation is a complete software emulation of a circuit. Given a starting state and a sequence of concrete inputs, it determines exactly how a circuit will react by finding the complete state of the circuit at each time-step. For netlists, this is done by evaluating the excitation function for each gate in turn until a complete circuit state is reached.

Time	a	b	c	o
0	0	1	0	1
1	1	1	1	

(i) Input Trace



(ii) Simulation Calculation

Figure 2.3: Binary Circuit Simulation

Example Figure 2.3 illustrates the simulation of a simple gate-level circuit model. The four stages of simulation are:

- The inputs and state at time zero are assigned to their corresponding netlist nodes.
- The values are propagated forward to the rest of the circuit, by calculating the output of each gate in turn.
- A time-step is simulated by copying the values on delay gate inputs to their outputs, and then clearing the values assigned to all other nodes.
- Another propagation is then performed for the subsequent time-step.

The result of binary simulation is that the Boolean value of every node is calculated for the entire bounded time-frame for which the inputs were provided.

2.3 Ternary Simulation

As we have seen, binary circuit simulation allows us to calculate how a circuit will react to *one particular* concrete input trace over time. Verifying more interesting properties, however, typically requires us to reason over many input traces at once. For instance, if a property requires us to try all possible input values for a circuit, then

$\vee$	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

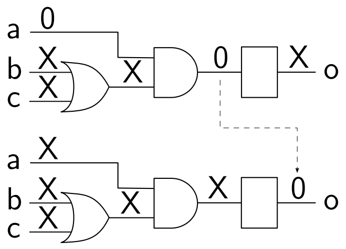
$\wedge$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$\neg$	0	1	X
	1	0	X

Figure 2.4: Ternary Logic

Time	a	b	c	o
0	0	X	X	X
1	X	X	X	

(i) Input Trace



(ii) Simulation Calculation

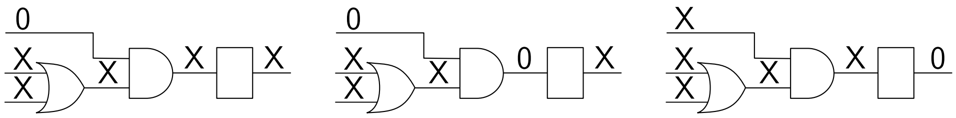
Figure 2.5: Ternary Circuit Simulation

the number of simulations required will be exponential in the number of these inputs, making binary simulation impractical.

Ternary simulation introduces an extra *unknown* value (or ‘don’t care’ value), written X, that indicates that nothing is known about the charge held on a particular node. Each simulation step assigns a *ternary value* from the set $\mathbb{T} = \{0, 1, \mathsf{X}\}$ to each node. Simulation states are therefore elements of $\mathbb{T}^{\mathcal{N}}$, which can also be seen as *partial* circuit states.

Ternary simulation proceeds in the same format as binary simulation, but with a new interpretation of excitation functions based on a three-valued logic over ternary values. The output of a gate can only be assigned 0 or 1 if enough information is known to deduce this output, given the partial information we have about its inputs. If there is not enough information then X is assigned instead. For example, if one input of an OR-gate is high, then we know that its output must also be high, so $\mathsf{X} \vee 1 = 1$. The same input conditions, however, are not enough to deduce anything about the output of an AND-gate, so $\mathsf{X} \wedge 1$ is evaluated to X. This is equivalent to interpreting the standard logic gates with the ternary logic shown in Figure 2.4.

Example Suppose we would like to verify, for the circuit in Figure 2.5 (ii), that if node a is low at Time 0 then the output o must be low at Time 1. To use binary simulation, we would have to run a simulation for each of the possible input combinations, totaling



(i) s (ii) $|s|$ (iii) $\mathrm{post}^\sharp(s)$

Figure 2.6: Ternary Simulation Steps

$2^6 = 64$ simulations. We can, however, instead verify all of these traces in a single ternary run, as shown in Figure 2.5.

2.3.1 Modeling Ternary Simulation

We introduce some notation in order to describe the steps in ternary simulation. Figure 2.6 (i) shows the ternary vector s that represents all consistent circuit states where the node marked 0 is low. Maximal propagation of these constraints, within a single time-frame, is formalized by the propagation operator $| \cdot |$, which is illustrated in Figure 2.6(ii). This operator has been described in detail in [RC06a], where it is referred to as the *forwards closure function* of the circuit. Its effect is to assign to the output of each node the least approximate output of each gate that can be deduced from the ternary inputs currently assigned to it. The abstract post-image function, $\mathrm{post}^\sharp$, represents first performing this propagation, then passing the resulting constraints across delay elements and marking every other node as X, as shown in Figure 2.6(iii). This results in a post-image state containing all the constraints that can be deduced by forward propagation from its predecessor.

2.3.2 Ternary Abstraction

Unlike binary simulation, ternary simulation can be seen as operating with *sets* of circuit states, much like many standard forms of model checking. This is because ternary states can be seen as abstract representations of sets of circuit states. For example, the ternary state X0X can be seen as an abstract representation of the set $\{000, 001, 100, 101\}$. In ternary simulation, these representations are used as *upper-approximations*, whose precision can be controlled by controlling the number of Xs. For example, the set $\{0011, 0001\}$ can be represented precisely using the ternary vector 00X1, and approximately by 0XX1 or even XXXX. Some sets cannot be represented exactly. For example, the most precise representation for $\{0011, 0101\}$ is 0XX1. STE has a symbolic mechanism for dealing with such cases, which we will come to later.

The state-set abstraction in ternary simulation is an example of a *Cartesian abstraction* [CC95], which is used to abstract away dependencies between model subsystems. Using ternary vectors forces the simulation to *ignore dependencies* between the circuit nodes. In effect, it represents each set of circuit states as the product of the set of states of each node, or, equivalently, hyper-cuboids in the space of circuit state bit-vectors.

This abstraction is successful in hardware verification because useful properties often correspond to simple constraints on a small number of circuit nodes, which can be efficiently and precisely encoded as ternary vectors. Furthermore, the vector representation fits naturally with gate-level simulation, since it assigns values to circuit nodes in the netlist. This allows post-image calculations to proceed via simulations, which are fast to calculate.

2.3.3 Abstract Interpretation Framework

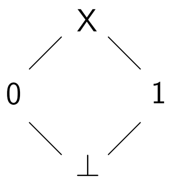
The relation between ternary vectors and sets of circuit states has been described in [Cho99, Pre04], using the theory of abstract interpretation [BG00, Cou01] to formalize the approximation. Concrete sets of circuit states are related to their corresponding abstract ternary vector representations via a *Galois connection*.

To describe this connection it is first necessary to model our abstract and concrete domains using partial orders that capture their internal degrees of approximation. For concrete sets of circuit states, this is simply modeled by set inclusion, $\subseteq$. We model approximation over ternary vectors using the partial order $\sqsubseteq$, which is interpreted as 'is less approximate than'.

Definition 2.3.1. *The ternary approximation order on ternary values is the least reflexive relation where* $\mathsf{0}$ *and* $\mathsf{1}$ *are both less approximate than* X*:* $\mathsf{0} \sqsubseteq \mathsf{X}$ *and* $\mathsf{1} \sqsubseteq \mathsf{X}$.

Extending $\sqsubseteq$ point-wise to vectors and adding a bottom element $\bot$, to represent the case of an over-constraint, creates a complete lattice of ternary vectors, $\mathbb{T}_{\bot}^{\mathcal{N}}$. This lattice is illustrated for vectors of length one in Figure 2.7(i), and vectors of length two in Figure 2.8 (page 18). Notice that $\mathsf{XX}\ldots\mathsf{X}$ is the top element of such a lattice, representing the set of all consistent circuit states.

The join and meet operations for the binary state-set lattice are trivially $\cup$ and $\cap$ respectively. For the ternary vectors the join is written $\sqcup$ and the meet $\sqcap$. Informally, the join takes those constraints that are common to both its operands, and the meet takes all those constraints that exist in either of its operands. For vectors of size one, these operations are shown in Figure 2.7. Higher-dimensional vectors follow the



(i) Hasse Diagram

$\sqcup$	X	0	1	$\bot$
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
$\bot$	X	0	1	$\bot$

(ii) Join

$\sqcap$	X	0	1	$\bot$
X	X	0	1	$\bot$
0	0	0	$\bot$	$\bot$
1	1	$\bot$	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

(iii) Meet

Figure 2.7: Ternary Lattice

point-wise application of this join and meet with a coalesced bottom (i.e. $\mathsf{01} \sqcap \mathsf{1X}$ equals $\bot$).

A Galois connection [CC79] is a pair of functions $(\alpha : C \rightarrow A, \gamma : A \rightarrow C)$ that can be used to link the ordered concrete domain, $(C, \subseteq)$, to an ordered abstract domain, $(A, \sqsubseteq)$. The *abstraction map*, α, maps elements of the concrete domain to their most precise abstract representation. The *concretization map* $\gamma : A \rightarrow C$ maps abstract representations to the least upper bound of the concrete set of elements that they represent. In order to be a Galois connection, the two maps must satisfy the condition that for all $a \in A$ and for all $c \in C$:

$$\alpha(c) \sqsubseteq a \quad \text{if and only if} \quad c \subseteq \gamma(a)$$

In ternary simulation theory, such a Galois connection is used to demonstrate that the ternary states are sound upper-approximations of the relevant sets of concrete binary states.

In previous models of ternary simulation [Cho99], the concrete domain has been based on all possible bit-vectors in $\mathbb{B}^{\mathcal{N}}$. We find instead that it is more appropriate to connect ternary vectors only to those sets of *consistent* bit-vectors, from $S \subset \mathbb{B}^{\mathcal{N}}$, which agree with the constraints imposed by the circuit logic. This provides a more accurate representation of ternary simulation, since our concrete domain is in one-to-one correspondence with the possible physical states of the circuit. It is also more faithful to actual implementations, where, for example, values are automatically assigned to circuit nodes that are marked as logically constant. Such a step can only be justified in our model.

Definition 2.3.2. *The Galois connection between ternary vectors and sets of binary valued circuit states is uniquely defined by the following concretization map:*

$$\gamma(a) = \begin{cases} \emptyset & \textit{for } a = \bot \\ \{s \in S \mid \forall \mathsf{n} \in \mathcal{N} : s(\mathsf{n}) \sqsubseteq a(\mathsf{n})\} & \textit{for } a \in \mathbb{T}^{\mathcal{N}} \end{cases}$$

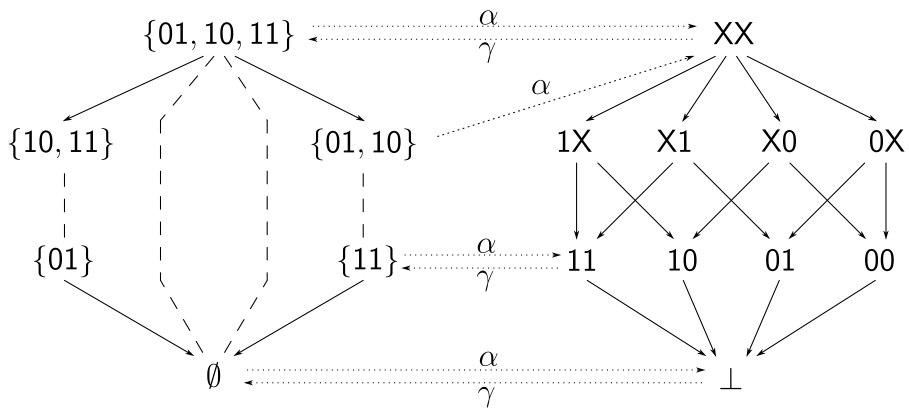


Figure 2.8: Galois Connection for Ternary Vectors

In this map, bottom represents the empty set of states, and each ternary vector represents the set of states that can be obtained by replacing X with any choice of Boolean values. Such a Galois connection is illustrated in Figure 2.8, for a model where 00 is the only 2-bit-vector that is not a consistent state.

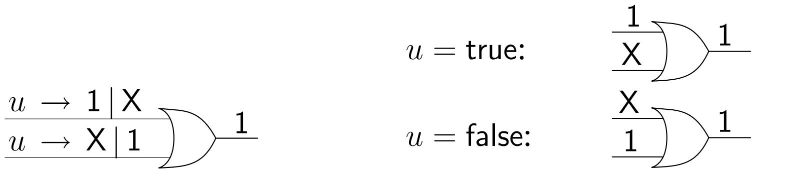
2.4 Symbolic Trajectory Evaluation

In Symbolic Trajectory Evaluation (STE) [SB95, BBS91] Boolean-valued variables are incorporated into ternary simulations to parameterize different circuit operating conditions. STE has shown considerable success in practice, and has been used on industrial hardware designs at Intel, Compaq, IBM and Motorola [AJS98, OZGS99, BMAA01, PRBA97, Kai05]. Where standard symbolic simulation uses variables to represent different *binary* simulations, Symbolic Trajectory Evaluation (STE) uses variables to represent different *ternary* simulations. We will call this technique *symbolic ternary simulation*.

For example, in a memory verification, we might like to check that a particular location of memory correctly stores *any* n-bit data value. Using ordinary ternary simulation will require 2^n runs—one for each possible data value. With STE, however, we can represent an arbitrary data value by using variables in the simulation. The verification can then proceed in one single symbolic simulation run.

STE uses a finite set, Var, of variables, called *indexing variables* [1], that parameterize the different simulations. Simulation then takes place over a domain of symbolic representations of ternary values [BBS91]. For the purposes of this dissertation, we

[1] Indexing variables are sometimes referred to as *symbolic variables* in (G)STE literature.


$u \rightarrow 1\,|\,\mathsf{X}$
$u \rightarrow \mathsf{X}\,|\,1$
1
$u =$ true:
1
X
1
$u =$ false:
X
1
1

(i) Symbolic Run (ii) Equivalent Individual Runs

Figure 2.9: A Run of STE

will write symbolic ternary values as either one of the ternary constants, 0, 1 or X, or as a conditional expressions of the form $Q \rightarrow f \,|\, g$, meaning 'if Q then f else g', where Q is a predicate over Var and f and g are also symbolic ternary vectors.

STE simulation lifts ternary simulation to the symbolic domain. Simulation propagation remains the same over ternary constants: $\mathsf{X}\wedge 1 = \mathsf{X}$ and $\mathsf{X}\vee 1 = 1$. Symbolically conditional inputs, however, result in symbolically conditional outputs, calculated on a case-by-case basis. For example,

$$(Q \rightarrow 1\,|\,\mathsf{X}) \wedge 1 \;=\; (Q \rightarrow (1 \wedge 1)\,|\,(\mathsf{X} \wedge 1)) \;=\; Q \rightarrow 1\,|\,\mathsf{X}$$

Since the simulation for each variable valuation is effectively treated independently, a symbolic run of STE effectively encodes multiple ternary simulations, as illustrated in Figure 2.9.

The interplay between Xs and symbolic simulation in STE allows the user to gain fine control over the precision with which state-sets are represented. The introduction of X values can be used to lose irrelevant information, whereas variables can be used to retain useful dependencies that are otherwise lost by the Cartesian abstraction. For example, with ternary vectors alone, the best approximation of $\{01, 10\}$ is XX. With symbolic ternary simulation, however, we can introduce a fresh variable, u, and use the symbolic ternary vector $(u \rightarrow 0\,|\,1)(u \rightarrow 1\,|\,0)$ to parameterize the two cases exactly. In the extreme, symbolic ternary representations can be complete representations of sets of states, when a unique variable is placed on every circuit node. The combination of Xs and variables can be used creatively in a variety of cases to balance loss of information against the cost of simulation [MJ02, PRBA97, VB98, ABMS07]. Goel has developed some theory to unify this interplay [Goe04, GB04].

The use of variables can be particularly effective when they are used for representing values on *datapaths* within a circuit. Datapaths carry the information being processed by the circuit, as distinct from *control buses*, which govern the circuit's mode of operation. In cases where there is little feedback from data to control, the use of variables enforces separation between the data and control aspects of verification. Furthermore, when no data transformation occurs, such as in memories and buffers, the symbolic data expressions remain constant in size.

The use of symbolic representations also increases the effectiveness of STE by allowing for *sharing* within the simulation. For example, suppose we wish to simulate a circuit which calculates $f(x, g(y))$, for inputs (x, y) in $\{(1, 0), (0, 0)\}$. This can be encoded as the single symbolic run where $x = (u \rightarrow 1 \,|\, 0)$ and $y = 0$. As a result, the simulation of g only takes place once, even though we actually calculate the result of two different input pairs. Given that g might represent an arbitrarily large piece of circuitry, this sharing can have a large impact on verification time.

2.4.1 Implementation

The implementation of STE inside the Forte verification platform [SJO+05] uses a canonical *dual-rail encoding* of symbolic ternary values for fast simulation. Each symbolic ternary value is represented by a pair (p, q) of Boolean predicates. Predicate p encodes the symbolic cases under which the expression may be high, and q encodes the symbolic cases under which the expression may be low. For example, X is represented by $(\mathrm{true}, \mathrm{true})$, 1 by $(\mathrm{true}, \mathrm{false})$, u by $(u, \neg u)$ and $u \rightarrow 1 \,|\, \mathsf{X}$ by $(\mathrm{true}, \neg u)$.

Each logic gate can then be quickly simulated by its appropriate interpretation on dual-rail values. In particular, the common logical operations from Figures 2.4 and 2.7 map to the following implementations for dual-rail representations:

$$\begin{array}{lcllcl}
\neg(p, q) & := & (q, p) & (p, q) \sqcup (r, s) & := & (p \vee r, q \vee s) \\
(p, q) \wedge (r, s) & := & (p \wedge r, q \vee s) & (p, q) \sqcap (r, s) & := & (p \wedge r, q \wedge s) \\
(p, q) \vee (r, s) & := & (p \vee r, q \wedge s) & (p, q) \sqsubseteq (r, s) & := & (p \Rightarrow r) \wedge (q \Rightarrow s)
\end{array}$$

A dual-rail value (p, q) is inconsistent when $\neg p \wedge \neg q$. Under such conditions, the node value represented can be neither high nor low. Such a condition therefore signals an inconsistency within the circuit and simulation assumptions.

To avoid over-approximation during simulation, such inconsistencies need to be propagated between time-steps. But an inconsistent value such as this may 'fall off' the edge of the simulation when it is assigned to a circuit output node. To rectify this situation, the STE simulator also keeps a global persistent *over-constraint* predicate that keeps track of the symbolic valuations that are inconsistent.

The dual-rail predicates are traditionally represented using Ordered Binary Decision Diagram (OBBDs) [Bry92] for compact and efficient operation implementations.

More recently, there have been attempts to use other forms Boolean reasoning, such as SAT-based STE [RC05, GSY07].

2.5 Generalized Symbolic Trajectory Evaluation

STE is an effective technique that scales to industrial size verification efforts. But STE can only check bounded properties of fixed temporal length because of its basis in scalar simulation. In contrast, many other standard model checking techniques verify more expressive properties, such as liveness properties [JGP99, McM92]. Generalized symbolic trajectory evaluation (GSTE) extends the principles of symbolic ternary simulation to handle such richer classes of properties.

The first step toward GSTE was by Seger and Hazelhurst, who added an *until* operator [Haz96] to their specification descriptions. Seger and Bryant proposed a means for checking iterative simulations of arbitrary length [SB95] using simple specifications based on regular expressions. Specifications were first generalized to arbitrary transition systems by Beatty [BB94], and a generalized model checking algorithm for these specifications was proposed by Nelson and Jain in [NJB97]. This algorithm was subtly refined by Chou [Cho99], leading to introduction of GSTE by Yang and Seger using the graphical specification notation of assertion graphs [YS03, YS00]. GSTE has a high capacity beyond the scope of traditional model checking techniques, because of its approach to the state-explosion problem [YS02, Sch03].

2.5.1 Set-Based Assertion Graphs

Properties for verification by GSTE are traditionally specified using *assertion graphs* [YS00, YS03, YS02, YG02]. An assertion graph is a directed graph with an initial vertex, where each edge is labeled with *antecedent* and *consequent* conditions. The antecedent conditions drive the simulation by providing the input stimulus, whereas the consequent conditions describe the resulting circuit responses to be asserted. Assertion graphs resemble input-output automata, where the antecedent resembles the input, and the consequent resembles the asserted output. They most closely align with the variety of automata known as *forall* automata [MP87], owing to the fact that a word is accepted only if it satisfied by the assertions made in *every* path through the graph. We will start off by describing set-based assertion graphs, where antecedent and consequent conditions are defined in terms of sets of circuit states.

Definition 2.5.1 (Set-Based Assertion Graphs)**.** *For a given Kripke Structure $\mathcal{K} = (S, T)$, an assertion graph is a triplet $\mathcal{G} = (V, v_0, E)$ where V is a set of vertices, $v_0 \in V$ is an initial vertex, and $E \subseteq V \times 2^S \times 2^S \times V$ is a set of doubly-labeled edges. For a given edge $e = (v, a, c, v')$, we say that $\mathrm{source}(e) = v$ is the source vertex, $\mathrm{ant}(e) = a$ is the antecedent, $\mathrm{con}(e) = c$ is the consequent, and $\mathrm{sink}(e) = v'$ is the sink vertex.*

Example An example assertion graph that might be used to verify a simple memory cell is shown in Figure 2.10. The edges are labeled with characteristic predicates in the form *antecedent/consequent*. This graph expresses that if write node wr is enabled with input 3, then the output of the memory cell should subsequently be 3, as long as no further writes take place.

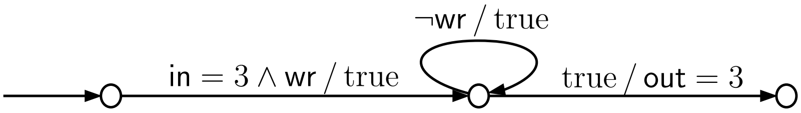


Figure 2.10: Set-Based Assertion Graph

The property described by an assertion graph can be understood by considering *all* finite paths that start at the initial vertex. Each of these paths expresses an assertion about all circuit traces of the same length.

Definition 2.5.2 (Initial Edge)**.** *For assertion graph $\mathcal{G} = (V, v_0, E)$, we will say that an edge $e \in E$ is initial if $\mathrm{source}(e) = v_0$.*

Definition 2.5.3 (Assertion Graph Path)**.** *A path of an assertion graph is a non-empty finite sequence of edges $\rho = e_0 e_1 \dots e_n \in E^+$, where e_0 is an initial edge and $\mathrm{sink}(e_i) = \mathrm{source}(e_{i+1})$ for $0 \leq i < n$.*

Definition 2.5.4 (Path Satisfaction)**.** *A model trace $\sigma \in \mathrm{tr}(\mathcal{K})$ is said to satisfy path ρ, of the same length, written $(\mathcal{K}, \sigma) \models (\mathcal{G}, \rho)$, if and only if whenever σ satisfies all the antecedents along ρ, it also satisfies all the consequents along ρ:*

$$(\forall i : 0 \leq i < |\rho| \,.\, \sigma_i \in \mathrm{ant}(\rho_i)) \;\; \textit{implies} \;\; (\forall i : 0 \leq i < |\rho| \,.\, \sigma_i \in \mathrm{con}(\rho_i))$$

Definition 2.5.5 (Assertion Graph Satisfaction)**.** *A Kripke Structure $\mathcal{K}$ satisfies an assertion graph $\mathcal{G}$, written $\mathcal{K} \models \mathcal{G}$, if and only if every trace σ of $\mathcal{K}$ satisfies every path ρ of $\mathcal{G}$ of the same length.*

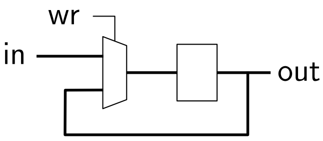


Figure 2.11: A Simple Memory Cell Circuit

Example We will consider whether the simple memory cell circuit shown in Figure 2.11 satisfies the example assertion graph from Figure 2.10. The possible paths through the assertion graph are of the form:

Antecedent	$\mathsf{in} = 3 \wedge \mathsf{wr}$	
Consequent		$\mathsf{out} = 3$

Antecedent	$\mathsf{in} = 3 \wedge \mathsf{wr}$	$\neg\mathsf{wr}$	
Consequent			$\mathsf{out} = 3$

Antecedent	$\mathsf{in} = 3 \wedge \mathsf{wr}$	$\neg\mathsf{wr}$	$\neg\mathsf{wr}$	
Consequent				$\mathsf{out} = 3$

Antecedent	$\mathsf{in} = 3 \wedge \mathsf{wr}$	$\neg\mathsf{wr}$	$\cdots$	$\neg\mathsf{wr}$	
Consequent					$\mathsf{out} = 3$

In order to satisfy the assertion graph, the circuit must satisfy all these bounded assertions. A trace satisfies one of these assertions if every time the antecedent conditions are satisfied, the consequent conditions are also satisfied. Table 2.1 shows the path of length 3 against some of the possible traces for the simple memory circuit. Trace 1 satisfies the path because the antecedent is not satisfied at the first edge. Trace 2 satisfies the path because the antecedent is not satisfied at the second edge. Trace 3 satisfies the path because both the antecedents and the consequents are satisfied at every edge. If we continue in this way for every trace and path then we will discover that the circuit satisfies the assertion graph.

Antecedent Consequent	$\mathsf{in} = 3 \wedge \mathsf{wr}$			$\neg\mathsf{wr}$			$\mathsf{out} = 3$			
Node	wr	in	out	wr	in	out	wr	in	out	Satisfied
Trace 1	0	3	4	0	6	4	0	7	4	✓
Trace 2	1	3	4	1	6	3	0	7	6	✓
Trace 3	1	3	4	0	6	3	0	7	3	✓

Table 2.1: Path Satisfaction Example

2.5.2 Model Checking

The standard GSTE model checking algorithm [YS03] associates a set of model states, $\mathrm{sim}[e]$, with each assertion graph edge e. This set collects the states that are reachable via a trace that satisfies the antecedent conditions along some path from the initial vertex to e. Intuitively $\mathrm{sim}[e]$ holds the combined post-image for the antecedent paths that precede it.

Like many forms of reachability analysis, the algorithm proceeds by repeatedly calculating and including the post-image of each transition in turn, until a fixed-point is attained. The algorithm keeps a queue of edges, queue, whose set of states has been updated but whose image remains to be computed. Initially, this queue is set to hold all the initial edges, and $\mathrm{sim}[e]$ is set to $\mathrm{ant}(e)$ for each of these edges.

To reach the required fixed-point, the algorithm repeats the following steps. First an edge e is removed from queue. Then the post-image of $\mathrm{sim}[e]$ is computed. For each edge e' succeeding e, $\mathrm{sim}[e']$ is assigned $\mathrm{sim}[e'] \cup (\mathrm{post}(\mathrm{sim}[e]) \cap \mathrm{ant}(e'))$. This adds those states from the post-image that satisfy the successor edge's antecedent. If this assignment adds new states, then e' is enqueued to have its own post-image recalculated. Once the fixed-point is complete, the containment check $\mathrm{sim}[e] \subseteq \mathrm{con}(e)$ is performed for each edge of the graph. This completes the algorithm, which is summarized in Figure 2.12.

It has been shown in [SSTV04] that the GSTE algorithm without ternary abstraction corresponds to a *partitioned* form of standard symbolic model checking (SMC) [McM92]. The difference concerns the way in which the connection between property states and model states is represented. In SMC, each symbolic predicate represents a subset of the property-model product state-space. In contrast, GSTE maintains a map, sim, from property states to model state-sets. Thus the GSTE approach partitions the representations according to property states. The two representations are isomorphic, but it has been shown that GSTE-style property partitioning is often advantageous [STV05].

2.5.2.1 Justifying Model Checking

We will briefly describe why model checking is sound and complete. First, to demonstrate completeness, suppose that model checking fails. Then there is some edge e such that $\mathrm{sim}[e] \not\subseteq \mathrm{con}(e)$. Therefore there is some state s such that $s \in \mathrm{sim}[e]$ and $s \notin \mathrm{con}(e)$. There is a model checking invariant that states that $s \in \mathrm{sim}[e]$ only if there is some circuit trace σ that satisfies all the antecedents along an initial path ρ that ends

```
// Initialization:
$\mathrm{queue} = $ **empty**
**foreach** $e \in E$
    **if** $\mathrm{source}(e) = v_0$
        $\mathrm{sim}[e] := \mathrm{ant}(e)$
        $\mathrm{queue.push}(e)$
    **else**
        $\mathrm{sim}[e] := \emptyset$
    **endif**
**endfor**

// Fixed point calculation:
**while** $\mathrm{queue} \neq$ **empty**
    $e := \mathrm{queue.pop}()$
    **foreach** successor $e'$ of $e$
        $\mathrm{update} := \mathrm{sim}[e'] \cup (\mathrm{post}(\mathrm{sim}[e]) \cap \mathrm{ant}(e'))$
        **if** $\mathrm{update} \neq \mathrm{sim}[e']$
            $\mathrm{sim}[e'] := \mathrm{update}$
            $\mathrm{queue.push}(e')$
        **endif**
    **endfor**
**endwhile**

// Consequent check:
**for** $e \in E$
    **assert** $\mathrm{sim}[e] \subseteq \mathrm{con}(e)$
**endfor**
```

Figure 2.12: Set-Based Model Checking Algorithm

in s. But such a path cannot satisfy ρ since it does not satisfy the last consequent. Hence the model does not satisfy the assertion graph.

Now suppose that the circuit model does not satisfy the assertion graph. Then there must be some path ρ and trace σ of minimal length such that σ does not satisfy ρ. This means that σ satisfies all the antecedents along ρ, but not all the consequents. Now, since ρ and σ are of minimal length, it must be on the last edge that the consequent fails, otherwise there would be a shorter prefix path and trace that also fails. After the fixed-point computation, we have by the model checking invariant that $\mathrm{last}(\sigma) \in \mathrm{sim}[\mathrm{last}(\rho)]$, since σ satisfies all the antecedents up to edge $\mathrm{last}(\rho)$. Therefore, it must be that $\mathrm{sim}[\mathrm{last}(\rho)] \not\subseteq \mathrm{con}(\mathrm{last}(\rho))$ and so model checking must fail.

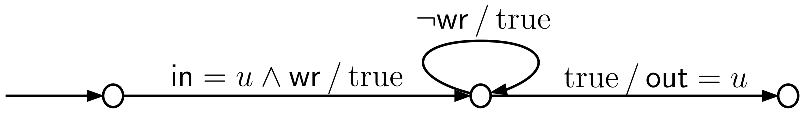


Figure 2.13: A Symbolic Assertion Graph

2.5.3 Using Symbolic Ternary Simulation

Like STE, GSTE is implemented via symbolic ternary simulation of the circuit netlist to combat the state-explosion problem. We show how the set-based algorithm presented so far can be adapted to this style of simulation.

2.5.3.1 GSTE Assertion Graphs

GSTE assertion graphs are just like set-based assertion graphs, except that the antecedent and consequent predicate labels are given as symbolic ternary vectors. We will write $\mathcal{V} = (\mathrm{Var} \rightarrow \mathbb{B})$ for the set of variable valuations. Symbolic ternary vectors then correspond to elements of $\mathcal{V} \rightarrow \mathbb{T}_{\perp}^{\mathcal{N}}$.

Definition 2.5.6 (GSTE Assertion Graph)**.** *For a given Kripke Structure $\mathcal{K} = (S, T)$, a GSTE assertion graph is a triplet $\mathcal{G} = (V, v_0, E)$ where V is a set of vertices, $v_0 \in V$ is the initial vertex, $E \subseteq V \times (\mathcal{V} \rightarrow \mathbb{T}_{\perp}^{\mathcal{N}}) \times (\mathcal{V} \rightarrow \mathbb{T}_{\perp}^{\mathcal{N}}) \times V$ is a set of edges.*

As with STE, the use of symbolic ternary vectors allows simulations to be generalized over different variable valuations. For example, the set-style assertion graph in Figure 2.10 can only test the memory cell with a single data value, 3. By encoding an arbitrary data value as a vector of variables, u, of appropriate size, the GSTE assertion graph in Figure 2.13 expresses that *any* data value is stored correctly.

2.5.3.2 Symbolic Ternary Model Checking

We update the model checking algorithm to use symbolic ternary simulation. It is necessary simply only to replace each operation on sets of states by the corresponding abstract operation, lifted to symbolic ternary vectors. To approximate union and intersection, GSTE uses the *join* $\sqcup$ and *meet* $\sqcap$ respectively, of the lattice of ternary propagations (see Figure 2.7). The post-image function post is replaced with the abstract post-image simulation $\mathrm{post}^{\sharp}$. Finally, the set inclusion test is replaced by checking the approximation order on the lattice of ternary vectors, $\sqsubseteq$.

The resulting algorithm is shown in Figure 2.14, where the operator symbols for ternary vectors have been overloaded with their symbolic counterparts. Notice that it has exactly the same flow as the algorithm for set-based model checking in Figure 2.12. All that has changed is that concrete set operations have been replaced with their symbolic abstract counterparts.

```
// Initialization:
$\mathrm{queue} = $ **empty**
**foreach** $e \in E$
    **if** $\mathrm{source}(e) = v_0$
        $\mathrm{sim}[e] := \mathrm{ant}(e)$
        $\mathrm{queue.push}(e)$
    **else**
        $\mathrm{sim}[e] := \bot$
    **endif**
**endfor**

// Fixed point calculation:
**while** $\mathrm{queue} \neq$ **empty**
    $e := \mathrm{queue.pop}()$
    **foreach** successor $e'$ of $e$
        $\mathrm{update} := \mathrm{sim}[e'] \sqcup (\mathrm{post}^{\sharp}(\mathrm{sim}[e]) \sqcap \mathrm{ant}(e'))$
        **if** $\mathrm{update} \neq \mathrm{sim}[e']$
            $\mathrm{sim}[e'] := \mathrm{update}$
            $\mathrm{queue.push}(e')$
        **endif**
    **endfor**
**endwhile**

// Consequent check:
**for** $e \in E$
    **assert** $\mathrm{sim}[e] \sqsubseteq \mathrm{con}(e)$
**endfor**
```

Figure 2.14: Abstract Model Checking Algorithm

2.5.4 Controlling Abstraction

Since both GSTE and STE model checking are based on symbolic ternary simulation, they both share many characteristics. Model checking is sound but not complete, and manual control over the property specification affects which parts of the circuit structure are simulated and what state information is retained. Although little has been written on abstraction in GSTE, the connection between these abstractions and cone-of-influence abstraction techniques has been explored in [YT00].

False negatives occur when a property holds, but there are too many X values in the simulation. The GSTE approach for dealing with false negatives is to refine the level of simulation abstraction by making changes to the structure of the specification of the property. Since the model checking flow is defined directly over the structure of the property, these changes can directly influence the precision with which state-sets are represented. Of course, such changes must also ensure that an equivalent or stronger property is verified as a result.

2.5.4.1 Case-splitting Edges

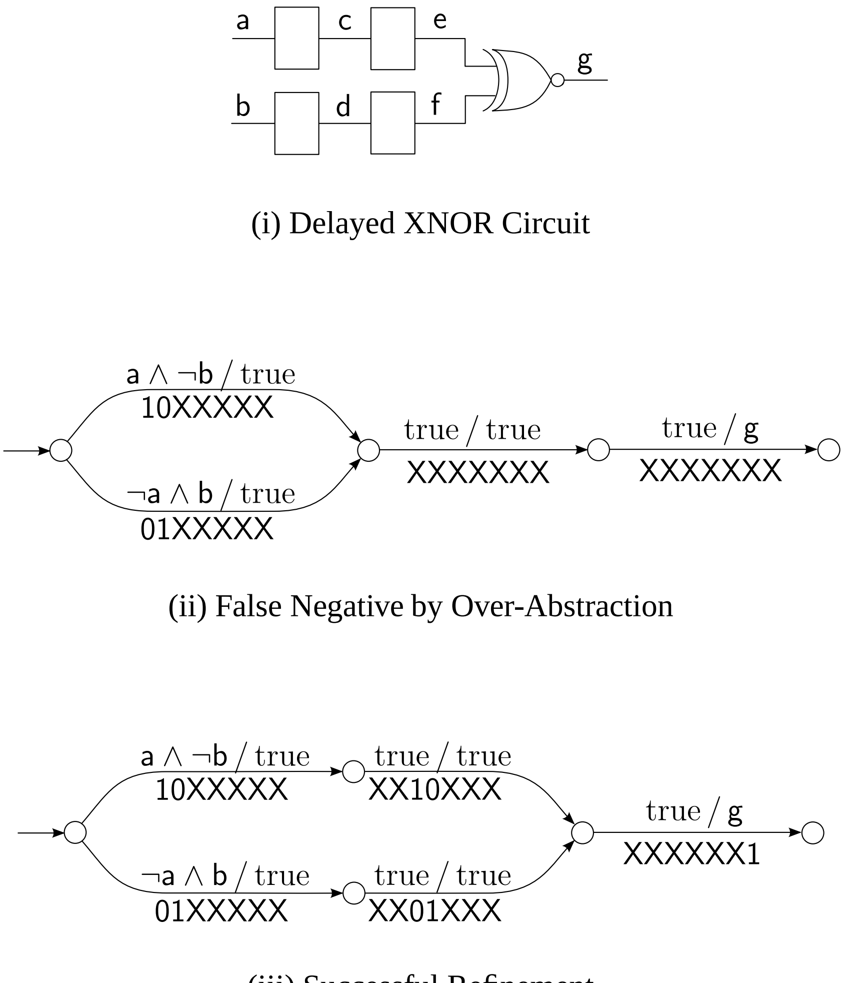
Over-abstraction most often occurs when model checking takes the union of two sets of states using the join operation $\sqcup$. This operation is often responsible for loss of precision because it carries forward only those state constraints that are common to *both* of the operands. The most common approach to prevent such information loss is to split a single assertion graph edge into two, separating out the approximations. This is akin to instructing the model checker to keep *two* ternary vectors to represent the union, rather than merging them together and losing information.

Example Consider the run of GSTE that is depicted in Figure 2.15. The aim is to verify the behaviour of a twice-delayed XNOR-gate shown in (i). We write ternary vectors in the form abcdefg. The first run, shown in Figure 2.15(ii), produces a false negative result, due to the over-abstraction that occurs at the second segment of the graph. During simulation, this edge is assigned XX10XXX $\sqcup$ XX01XXX, which is XXXXXXX. Therefore, all information about the required post-condition is lost due to the ternary abstract representation.

In order to overcome this false negative, a case-split transformation can be applied to the second time-slice of the assertion graph. The case-split distinguishes the two possible input cases occurring in the preceding time-frame. Following such a transformation, GSTE returns a positive result, as shown in Figure 2.15(iii).

2.5.4.2 Unrolling Loops

Another example of an abstraction refinement transformation is the *unrolling* of assertion graphs loops. Whenever a loop occurs in an assertion graph, the corresponding fixed-point calculation collects an approximation of the satisfying circuit states. Each step calculates the abstract union, $\sqcup$, of the previous state and the result of the next loop iteration. Since $\sqcup$ loses information, however, these types of fixed-points can over-approximate.

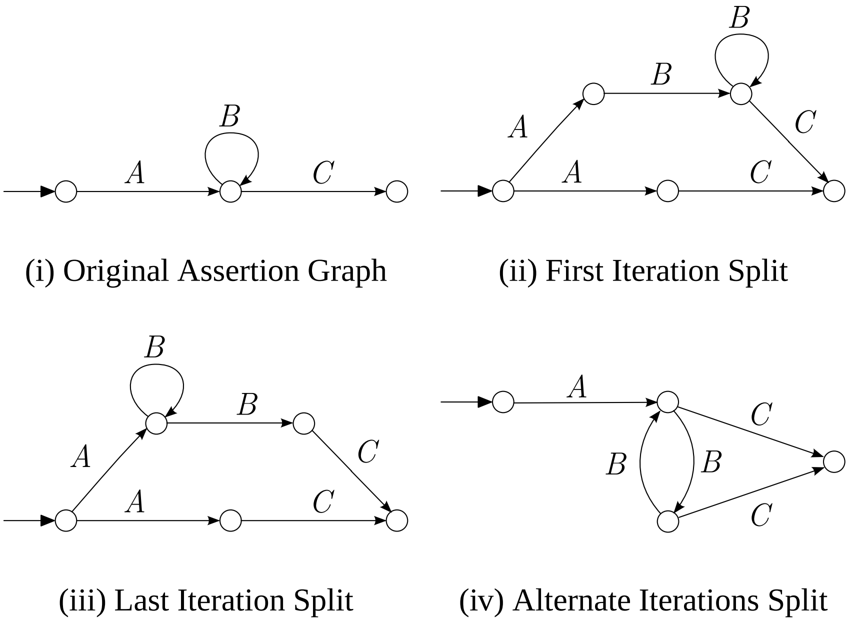


(i) Delayed XNOR Circuit

(ii) False Negative by Over-Abstraction

(iii) Successful Refinement

Figure 2.15: Refining an Assertion Graph



(i) Original Assertion Graph (ii) First Iteration Split

(iii) Last Iteration Split (iv) Alternate Iterations Split

Figure 2.16: Refinements by Unrolling

By unrolling an assertion graph loop, the states that correspond to different iterations in the fixed-point can be separated from each other and represented more precisely. A loop-unrolling therefore corresponds to a form of temporal case-split. Examples of loop-unrollings are shown in Figure 2.16.

2.5.4.3 Introducing Variables

As with STE, ternary vectors can also be case-split by introducing fresh symbolic variables, to capture dependencies between circuit nodes that are lost by the ternary vector representation. Unlike STE, a problem arises when such variables are introduced in assertion graph cycles. Since the variables are never quantified-out, their values are scoped over the entire simulation, and not just over each cycle iteration. But when variables are being used purely for the purpose of abstraction control, a fresh variable is required for each time-step, otherwise extra state dependencies can be unintentionally introduced. To cater for such cases, Yang and Seger define a special class of variables, for use in only antecedents, that are automatically removed via existential quantification at the end of each simulation step [YS02]. This quantification is sound, because the variables all occur negatively with respect to model checking satisfaction. The effect of this is that these variables are scoped only to the edge in which they appear.

2.5.4.4 Knots

In some cases, the precision needed requires that case-splits within fixed-point persist for longer than a single simulation step. This can be overcome via manual specification of variable quantification points to avoid name conflicts. These points are called *knots* [NHY04] in GSTE terminology, because they conceptually permit the tying together of an infinite line of case-splits into a loop.

2.5.4.5 Precise Nodes

If the precision introduced by case-splitting needs to persist even longer, then variable renaming with fresh names can be used to avoid conflicts between temporally overlapping scopes. If fresh variables are introduced at every iteration, however, then a fixed-point may never be attained.

This difficulty is resolved in GSTE through the use of what are termed *precise nodes* [YS02]. The general idea is as follows, but we will revisit it in some detail in Section 4.3.4. The set of precise nodes for a run of GSTE is a set of circuit nodes for which temporary anonymous variables are automatically introduced to maintain interdependencies. The use of such a set allows us to direct the model checker to increase the level of detail with which certain segments of the circuit are represented.

As an example, suppose a circuit has two 2-bit counters progressing in synchrony. The set of states in which the second counter is one ahead of the first can be described explicitly as $\{(00, 01), (01, 10), (10, 11), (11, 00)\}$. If the counters are marked as precise nodes, to avoid over-abstraction to $(\mathsf{XX}, \mathsf{XX})$, the state of the counters is automatically encoded using extra variables t_1 and t_2:

$$\begin{array}{lcl}
\neg t_1 \wedge \neg t_2 & \mapsto & (00, 01) \\
\neg t_1 \wedge t_2 & \mapsto & (01, 10) \\
t_1 \wedge \neg t_2 & \mapsto & (10, 11) \\
t_1 \wedge t_2 & \mapsto & (11, 00)
\end{array}$$

If we simulate such an encoding one step forward to find its post-image, then the result will be:

$$\begin{array}{lcl}
\neg t_1 \wedge \neg t_2 & \mapsto & (01, 10) \\
\neg t_1 \wedge t_2 & \mapsto & (10, 11) \\
t_1 \wedge \neg t_2 & \mapsto & (11, 00) \\
t_1 \wedge t_2 & \mapsto & (00, 01)
\end{array}$$

Although this representation describes the same set of states, it uses a different variable indexing. For fixed-point detection, these differences can be overcome using a canonical re-parameterization based on the algorithm in [AJS99]. Since the variables affected are anonymous, this re-parameterization does not affect the soundness of model checking.

2.6 Compositional GSTE

GSTE has also been further generalized to a form known as *compositional* (or *concurrent*) GSTE (CGSTE) [YS04, YGT05]. This extension allows different areas of a circuit to be independently simulated, and then have their simulations merged together. Performing these smaller simulations can greatly increase the capacity of the model checker since the maximum state space being explored at any one time is reduced. It does, however, require some insight into the design implementation or circuit structure.

Instead of using a single assertion graph, CGSTE specifications consist of multiple named assertion graphs, representing the sub-simulations. These assertion graphs can be seen as executing concurrently, effectively specifying a property as a product of individual machines. To allow for composition, the conditions on edge antecedents can contain terms called *compositional conditions*. Each composition condition references a given edge on any one of the group of assertion graphs. A trace satisfies a path containing such an antecedent condition if it satisfies the standard condition for normal assertion graphs, and all prefixes of the trace that match a path ending at a composition condition also satisfy some initial path of the graph referenced by that condition that ends at the referenced edge.

Example Suppose we have two memory cells whose outputs are attached to a comparator (Figure 2.17). In order to verify this circuit using a run of conventional GSTE, we must create a single assertion graph such as that shown in Figure 2.18. This specification must simulate both memory cells at the same time, considering all ways of

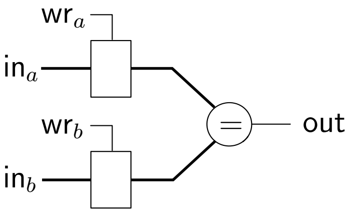


Figure 2.17: A Memory Cell Comparator

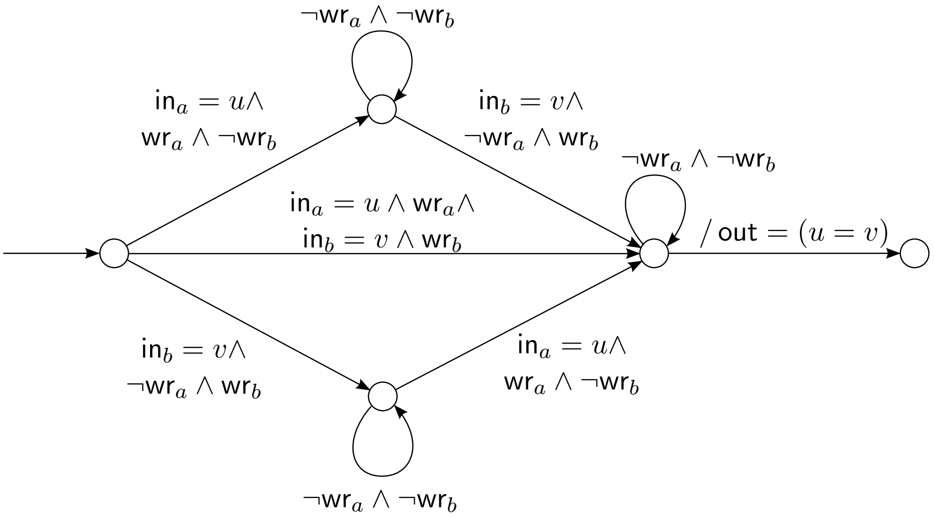

$\neg\mathsf{wr}_a \wedge \neg\mathsf{wr}_b$
$\mathsf{in}_a = u \wedge$
$\mathsf{wr}_a \wedge \neg\mathsf{wr}_b$
$\mathsf{in}_b = v \wedge$
$\neg\mathsf{wr}_a \wedge \mathsf{wr}_b$
$\neg\mathsf{wr}_a \wedge \neg\mathsf{wr}_b$
$\mathsf{in}_a = u \wedge \mathsf{wr}_a \wedge$
$\mathsf{in}_b = v \wedge \mathsf{wr}_b$
$/\,\mathsf{out} = (u = v)$
$\mathsf{in}_b = v \wedge$
$\neg\mathsf{wr}_a \wedge \mathsf{wr}_b$
$\mathsf{in}_a = u \wedge$
$\mathsf{wr}_a \wedge \neg\mathsf{wr}_b$
$\neg\mathsf{wr}_a \wedge \neg\mathsf{wr}_b$

Figure 2.18: Comparator Assertion Graph

interleaving writes of data u to the first memory cell and v to the second. Under these conditions it would then be expected that the circuit output matches the value of $u = v$.

By using compositional GSTE we can simulate each of the two memory cells independently. We create assertion graphs A and B for each of the cells, each with an edge named Done, to capture the conditions under which each of the memory cells should hold their respective symbolic values (Figure 2.19). We create a third graph, Main, to compose the two simulations and check the operation of the comparator. In compositional GSTE, edge antecedents may include predicates of the form “$\mathrm{G\ ON\ E}$”, which describe those states reachable from a path in the graph named G which ends at the edge named E. In our example, the first antecedent of the Main graph asserts the compositional condition $(\mathrm{A\ ON\ Done}) \wedge (\mathrm{B\ ON\ Done})$. Under this condition, we would expect the output of the comparator to be $u = v$ in the subsequent time-step.

During compositional model checking, each assertion graph is simulated in turn, and the compositional constraints from each are forwarded from one graph to another as symbolic ternary circuit states. Conjunction of the compositional conditions is interpreted using the meet on the lattice of ternary vectors. In the case where two graphs mutually refer to each other, the algorithm finds the corresponding fixed-point.

2.7 Other Variants

The framework for symbolic ternary simulation that we will present in the following chapters is sufficient to capture each of the simulation techniques that we have

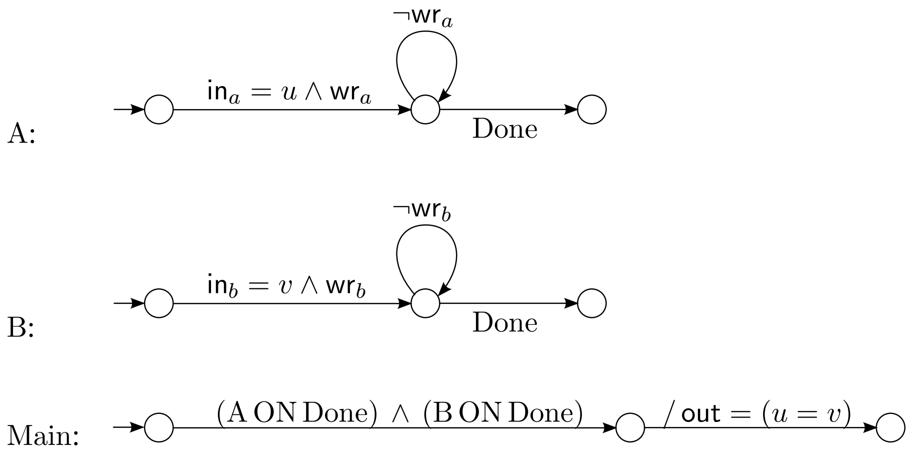

$\neg\mathsf{wr}_a$
A:
$\mathsf{in}_a = u \wedge \mathsf{wr}_a$
Done
$\neg\mathsf{wr}_b$
B:
$\mathsf{in}_b = v \wedge \mathsf{wr}_b$
Done
Main:
(A ON Done) $\wedge$ (B ON Done)
$/\,\mathsf{out} = (u = v)$

Figure 2.19: Comparator Compositional Assertion Graphs

introduced so far. There are several additional techniques that we will not cover. In particular, GSTE has been extended to liveness properties [YS00], specified using fairness constraints for assertion graphs. A form of backwards circuit simulation has also been proposed [YS02] that propagates circuit constraints bidirectionally. Although our framework will not cater for these approaches, we do not consider this to be a severe restriction, since compositional GSTE is sufficient for the majority of verification cases to which GSTE is applied in practice.

Chapter 3

Generalized Trajectory Logic

In this chapter, we address the problem of how runs of symbolic ternary simulation are best specified. A specification notation is required for two key roles. First, the notation needs to formally express the *property* of interest. This is necessary to communicate exactly *what* the verification demonstrates. Secondly, the notation must describe a simulation outline, giving information on *how* the verification takes place. Many model checking specification languages are concerned with the property alone, but manual guidance is paramount for symbolic ternary simulation, to allow for abstraction control and optimization strategies.

A successful specification notation achieves these roles in a way that makes all relevant aspects of the verification easy to manage. Most importantly, the notation needs to represent simulations directly and concisely with a clean semantics, so that their interpretations are as clear as possible. As well as this, it should provide good scope for formal reasoning, by existing in a form amenable to manipulation, and by preferably exhibiting compositional algebraic qualities that show promise for useful reasoning rules.

We introduce a specification language called *generalized trajectory logic* (GTL), which is a linear temporal logic that achieves these required qualities. We choose our notation to be expressive and detailed enough to express the common simulation patterns that we have explored in Chapter 2. An important quality of GTL is that each logical construct corresponds exactly with each of the atomic steps used within traditional symbolic ternary simulation techniques like STE and GSTE. By making the constructs of our logic in alignment with these steps, we allow our notation to describe both the simulation and the property in a single clean setting. By relating the constructs to propositional logic, we obtain a detailed and clear representation of simulations and their associated properties. As we will describe in Chapter 4, this fine level of simulation specification leads to clean reasoning rules.

To our knowledge, our logic is the first attempt at defining a formal property language capable of expressing all compositional GSTE runs. By providing a textual logical foundation, it greatly improves the outlook for formal reasoning about GSTE-style simulations. On top of this, we believe that GTL provides a good perspective from which to understand the model checking techniques that use symbolic ternary simulation, and in particular, GSTE.

3.1 Symbolic Indexing Notation

Before we define our specification, we introduce some notations for dealing with the symbolic mathematical structures used extensively by STE-based verification techniques. We will use the term *symbolically indexed representation* to describe a structure containing *index variable* symbols. These structures represent different values depending on the contextual valuation of the index variables that they contain. As we have seen in Chapter 2, STE-based techniques use symbolically indexed representations of sets of circuit states during verification. As a result of this, much of its underlying theory and notations are also built using these indexed representations.

We will assume that the indexing variables are Boolean-valued and drawn from a finite set Var. They therefore have associated set of valuations $\mathcal{V} = \mathrm{Var} \to \mathbb{B}$. We will model the symbolic representations of a set X using the set $X^{\mathcal{V}}$ of maps from valuations to X, and write $s\langle\nu\rangle$ for the value of s under valuation $\nu \in \mathcal{V}$.

Given a function $f : X \to Y$, its *indexed lifting* is the function $f^{\mathcal{V}} : X^{\mathcal{V}} \to Y^{\mathcal{V}}$ that applies f independently in each valuation:

$$(f^{\mathcal{V}}(x))\langle\nu\rangle \quad := \quad f(x\langle\nu\rangle) \tag{3.1}$$

We will lift an n-ary relation $R \subseteq X^n$ in three different ways. The *indexed lifting* of $R \subseteq X^n$ is the map $R^{\mathcal{V}} : (X^{\mathcal{V}})^n \to \mathbb{B}^{\mathcal{V}}$ that provides the Boolean predicate that is true in those valuations where the relation is satisfied. The *universal lifting* of R is the relation $R^{\forall\mathcal{V}} \subseteq (X^{\mathcal{V}})^n$ that is true for those symbolic representations where R holds in *every* index valuation. Dually, the *existential lifting* of R is the relation $R^{\exists\mathcal{V}} \subseteq (X^{\mathcal{V}})^n$ that is true of those representations in which R holds in *some* valuation. In summary:

$$\begin{aligned}
(R^{\mathcal{V}}(x))\langle\nu\rangle \quad &\text{iff} \quad R(x\langle\nu\rangle) \\
R^{\forall\mathcal{V}}(x) \quad &\text{iff} \quad \forall\nu \in \mathcal{V} \,.\, R(x\langle\nu\rangle) \\
R^{\exists\mathcal{V}}(x) \quad &\text{iff} \quad \exists\nu \in \mathcal{V} \,.\, R(x\langle\nu\rangle)
\end{aligned}$$

$$
\begin{array}{lcll}
f & ::= & \mathtt{tt} & \text{True} \\
 & | & \mathtt{ff} & \text{False} \\
 & | & \mathsf{n} & \text{Node is high} & \mathsf{n} \in \mathcal{N} \\
 & | & \neg\mathsf{n} & \text{Node is low} & \mathsf{n} \in \mathcal{N} \\
 & | & f \wedge f & \text{Conjunction} \\
 & | & f \vee f & \text{Disjunction} \\
 & | & \mathbf{Y} f & \text{Yesterday} \\
 & | & Z & \text{Recursion variable} & Z \in \mathcal{F} \\
 & | & \mu Z \,.\, f & \text{Least fixed-point} & Z \in \mathcal{F} \\
 & | & Q \;\rightarrow\; f \,|\, f & \text{Symbolic conditional} & Q \in \mathrm{pred}(\mathrm{Var}) \\
 & | & f(u := Q) & \text{Explicit substitution} & u \in \mathrm{Var}, Q \in \mathrm{pred}(\mathrm{Var})
\end{array}
$$

Figure 3.1: Syntax of GTL

Example Binary Decision Diagrams (BDDs) [Bry92] can be viewed as a form of symbolically indexed representation, commonly used to represent Boolean predicates over Var in model checking. Using our terminology, BDD conjunction is the symbolic lifting of Boolean conjunction. Similarly, implication on BDDs is the symbolic lifting of Boolean implication. The universal lifting of implication corresponds to Boolean implication followed by a validity check.

3.2 Formal Definition

This section provides a formal definition of generalized trajectory logic, and describes how it can be used to specify properties for symbolic ternary simulation.

3.2.1 Syntax

The syntax of GTL contains two different types of variables. First, we use Boolean-valued index variables $u \in \mathrm{Var}$ for the layer of symbolic representation that is maintained symbolic throughout the simulation process. Second, we use *recursion variables,* of the form $Z \in \mathcal{F}$, to capture fixed-point iteration in the style of the μ-calculi [BS01]. These variables correspond to intermediate simulation states.

Definition 3.2.1. *Formulas of GTL are those strings that satisfy the grammar shown in Figure 3.1, and that meet the syntactical requirement that within any fixed-point, $\mu Z \,.\, f$, every occurrence of the recursion variable Z in f is bound by an occurrence of the temporal Yesterday operator $\mathbf{Y}$.*

$$
\begin{array}{lcl}
\| \mathtt{tt} \|_\rho^\nu & = & S^+ \\
\| \mathtt{ff} \|_\rho^\nu & = & \emptyset \\
\| \mathsf{n} \|_\rho^\nu & = & \{ \, t \in S^+ \mid \mathrm{last}(t)(\mathsf{n}) = \mathsf{1} \, \} \\
\| \neg\mathsf{n} \|_\rho^\nu & = & \{ \, t \in S^+ \mid \mathrm{last}(t)(\mathsf{n}) = \mathsf{0} \, \} \\
\| f \wedge g \|_\rho^\nu & = & \| f \|_\rho^\nu \cap \| g \|_\rho^\nu \\
\| f \vee g \|_\rho^\nu & = & \| f \|_\rho^\nu \cup \| g \|_\rho^\nu \\
\| \mathbf{Y} f \|_\rho^\nu & = & \{ \, t.s \in S^+ \mid t \in \| f \|_\rho^\nu \, \} \\
\| Z \|_\rho^\nu & = & \rho(Z)\langle\nu\rangle \\
\| \mu Z \,.\, f \|_\rho^\nu & = & \bigcap \{ \, T\langle\nu\rangle \mid \| f \|_{\rho[Z\mapsto T]}^\nu \subseteq^{\forall\mathcal{V}} T \, \} \\
\| Q \rightarrow f \mid g \|_\rho^\nu & = & \text{if } Q\langle\nu\rangle \text{ then } \| f \|_\rho^\nu \text{ else } \| g \|_\rho^\nu \\
\| f(u := Q) \|_\rho^\nu & = & \| f \|_\rho^{\nu[u\mapsto Q\langle\nu\rangle]}
\end{array}
$$

Figure 3.2: Semantics of GTL

3.2.2 Semantics

The semantics for GTL is given in terms of non-empty finite words of states from the circuit model, which is assumed throughout to be the Kripke structure $\mathcal{K}_{\mathbf{C}} = (S, T)$, as described in Section 2.1.2. Recall that S is the finite set of bit-vectors that represent consistent circuit states, and that T is the model transition relation between these states. Each formula of GTL satisfies a particular set of words for each indexing variable valuation. This allows us to use indexing variables in formulas which correspond directly with the indexing variables in GSTE simulations. To model this, the semantic domain we use consists of maps from index valuations to sets of words, $(2^{S^+})^{\mathcal{V}}$.

To capture recursion, we use a *recursion context*, $\rho : \mathcal{F} \rightarrow (2^{S^+})^{\mathcal{V}}$, that provides semantic values for the free recursion variables in the formula being evaluated. We provide the semantics of GTL by giving a value to each formula of GTL in each possible fixed-point recursion context.

Definition 3.2.2. *The semantics of GTL are defined in Figure 3.2. The notation $\| f \|_\rho^\nu$ denotes the set of finite non-empty words from S^+ satisfied by formula f in symbolic indexing valuation $\nu \in \mathcal{V}$ and recursion context ρ.*

We describe each construct of the logic in turn. Every word satisfies truth, $\mathtt{tt}$, and no word satisfies falsity, $\mathtt{ff}$. A word satisfies the atomic proposition n, or $\neg\mathsf{n}$, if node n is high, or low, respectively, in the *final state* of the word. The connectives $\wedge$ and $\vee$ behave exactly as their equivalents in propositional logic, but notice that we do not allow for negation of arbitrary formulas.

The only temporal operator is the *Yesterday* operator, written $\mathbf{Y}$. Intuitively, $\mathbf{Y}f$ expresses that f held one time-step ago. A word $t.s$ ending in state s satisfies $\mathbf{Y}f$

if word t satisfies f. We handle fixed-point recursion in the standard manner of the μ-calculi. For example, $\mu Z \,.\, f \vee \mathbf{Y} Z$ expresses that f has held at some point in the past.

Finally, GTL has constructs for managing symbolically indexed representations. The *symbolic conditional* $Q \;\rightarrow\; f \mid g$ is equivalent to f in valuations where Q holds, and g otherwise. As an example, the formula $u \;\rightarrow\; \mathsf{n} \mid \neg\mathsf{n}$ describes the symbolic words ending in states where node n has value u. The *symbolic substitution* operator, written $f(u := Q)$, explicitly changes the current variable valuation context. Indexing variables used within these constructs remain symbolic during model checking, providing us with a handle to control the balance between the explicit and symbolic simulation. For example, $(\mathbf{Y}(u \;\rightarrow\; \mathsf{n} \mid \neg\mathsf{n}))(u := \mathrm{T})$ specifies the run that symbolically simulates the circuit with variable u, and subsequently uses this symbolic result to reference the state indexed by the case where u is true.

3.3 Semantic Characteristics

In this section, we explore some important characteristics of GTL that provide a basis for model checking. In particular, we show that GTL formulas are *continuous* with respect to their recursion context. This allows us to apply the Knaster-Tarski Theorem to deduce that the GTL fixed-point can be attained by a finite number of simulation iterations.

In order to reason about the effects of the recursion context, we will define the following map, which interprets the semantics of a formula as a function of a given recursion variable context:

Definition 3.3.1. *The function $\mathcal{R}_{f,\rho,Z}$ maps value $U \in (2^{S^+})^{\mathcal{V}}$ to the semantic value of f in formula context ρ extended by mapping variable Z to U:*

$$\mathcal{R}_{f,\rho,Z}(U) = \parallel f \parallel_{\rho[Z \mapsto U]}$$

This map then models the effect of passing once through a single iteration of a fixed-point computation.

3.3.1 Monotonicity

We show than any formula of GTL is monotonic with respect to the value of each recursion variable. As well as being a step on the way to demonstrating continuity, this property will also allow us to derive many useful reasoning rules about GTL in

Chapter 4. The property states that if the set of words satisfied by recursion variable Z increases, then we also increase the words satisfied by f.

Theorem 3.3.2 (Monotonicity)**.** *$\mathcal{R}_{f,\rho,Z}$ is monotonically increasing with respect to the symbolic containment relation, $\subseteq^{\forall\mathcal{V}}$.*

Proof. The proof is by structural induction over f and is given in Appendix A.1. □

3.3.2 Continuity

We demonstrate that the semantics of GTL are *finitary*, which is a sufficient condition for continuity. This corresponds to the statement that if a given word σ is satisfied by a formula recursion context ρ, there must be another recursion context ρ' such that the value of each recursion variable $\rho'(Z)$ is a finite subset of $\rho(Z)$, and in which σ is also satisfied.

In order to show that this property is true for GTL, we will use the following observation. Whether or not a word of length n satisfies a formula depends only on which words of length less than or equal to n are satisfied by the recursion variables in the evaluation context. Informally, this expresses that GTL only looks backwards in time a finite distance for each finite word. We formalize this argument, with a function that restricts a set of symbolic words to those words of length n or less:

Definition 3.3.3. *Let the nth* length restriction, *where $n \in \mathbb{Z} \cup \{+\infty, -\infty\}$, be the map $L^{\leq n} : (2^{S^+})^{\mathcal{V}} \to (2^{S^+})^{\mathcal{V}}$ defined by:*

$$L^{\leq n}(X)\langle\nu\rangle \quad := \quad \{\ \sigma \in X\langle\nu\rangle \ \mid\ |\sigma| \leq n\ \}$$

Notice that n need not be positive, and that $L^{\leq i}(X)$, for upwards of any i, is chain when ordered by $\subseteq^{\forall\mathcal{V}}$. It is simple to show that:

Lemma 3.3.4. *$L^{\leq n}$ is chain-continuous.*

Proof. Let X_i be a chain with respect to $\subseteq^{\forall\mathcal{V}}$. Then:

$$\begin{aligned} L^{\leq n}(\bigcup_i^{\mathcal{V}} X_i)\langle\nu\rangle &= \{x \in (2^{S^+})^{\mathcal{V}} \mid \exists\, i \,.\, x \in X_i\langle\nu\rangle \wedge |x| \leq n\} \\ &= \bigcup_i^{\mathcal{V}} L^{\leq n}(X_i)\langle\nu\rangle \end{aligned}$$

□

We will say that a function on symbolic sets is *consistently lengthening* when it is both monotonic, and the words of length n in its image are completely determined by the words of length less than or equal to n in its argument. Intuitively, this means that as we iterate through the fixed-point, the words that are covered are of ever-growing length.

Definition 3.3.5. *The map $F : (2^{S^+})^{\mathcal{V}} \to (2^{S^+})^{\mathcal{V}}$ is termed consistently lengthening iff F is both monotonic, and*

$$L^{\leq n}(F(X)) \ \subseteq^{\forall\mathcal{V}} \ F(L^{\leq n}(X))$$

for every $n \in \mathbb{Z}$ and $X \in (2^{S^+})^{\mathcal{V}}$.

We can now show that any consistently lengthening map is continuous, by demonstrating that each word in the image of the map is determined by a finite number of words in the argument. These are the those words that are bounded by the same length.

Lemma 3.3.6. *Every consistently lengthening map is chain-continuous.*

Proof. Let $F : (2^{S^+})^{\mathcal{V}} \to (2^{S^+})^{\mathcal{V}}$ be a consistently lengthening map and $X_i \in (2^{S^+})^{\mathcal{V}}$ be a chain with respect to $\subseteq^{\forall\mathcal{V}}$. Now suppose that word σ is in $F(\bigcup_i^{\mathcal{V}} X_i)$. It must be that σ is in the restriction of this set to words less than or equal to its own length: $\sigma \in L^{\leq|\sigma|}(F(\bigcup_i^{\mathcal{V}} X_i))$. It follows since F is consistently lengthening that $\sigma \in F(L^{\leq|\sigma|}(\bigcup_i^{\mathcal{V}} X_i))$. By Lemma 3.3.4, this equals $F(\bigcup_i^{\mathcal{V}} L^{\leq|\sigma|}(X_i))$. Now, since $\bigcup_i^{\mathcal{V}} L^{\leq|\sigma|}(X_i)$ is finite, there must exist some $j \in \mathbb{N}$ such that $\sigma \in F(L^{\leq|\sigma|}(X_j))$. Since F is monotonic, it follows that $\sigma \in F(X_j)$ and so $\sigma \in \bigcup_i^{\mathcal{V}} F(X_i)$. It is trivial to show that monotonicity also guarantees that $\bigcup_i^{\mathcal{V}} F(X_i) \subseteq^{\forall\mathcal{V}} F(\bigcup_i^{\mathcal{V}} X_i)$. Therefore $\bigcup_i^{\mathcal{V}} F(X_i) = F(\bigcup_i^{\mathcal{V}} X_i)$ and so F is chain-continuous. □

In order to demonstrate that $\mathcal{R}_{f,\rho,Z}$ is consistently lengthening, and therefore continuous, we use the concept of the *temporal depth* of a formula:

Definition 3.3.7. *Let the temporal depth of formula variable Z in formula f, written $\mathrm{depth}(Z, f)$, be the least number of $\mathbf{Y}$ operators binding any free occurrence of Z in f, and $+\infty$ when Z does not occur free in f.*

The temporal depth of a formula gives a measure of the difference in length between corresponding words that satisfy a formula f and Z. For example, since $\mathbf{Y}Z$ has temporal depth one, each word that satisfies Z corresponds to words *one* letter larger satisfying $\mathbf{Y}Z$. The temporal depth of a formula therefore gives a minimum measure of how much longer words will be after one recursive iteration through a formula.

We can show that $\mathcal{R}_{f,\rho,Z}$ is consistently lengthening by structural induction over f in the following theorem. We use a stronger inductive invariant than is strictly necessary, so that we can reuse the condition later, in Section 4.1.2.

Theorem 3.3.8 (Continuity)**.** *For every recursion context ρ and GTL formula f where every instance of recursion variable Z is bound by at least one instance of $\mathbf{Y}$, $\mathcal{R}_{f,\rho,Z}$ is consistently lengthening, and therefore continuous.*

Proof. The proof of this lemma is by structural induction over f and is given in Appendix A.2. For each formula it is shown in Appendix A.2 that the following condition holds:

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(R))) \ \subseteq^{\forall\mathcal{V}} \ \mathcal{R}_{f,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,f)}(R))$$

Due to the given requirement, $\mathrm{depth}(Z,f) \geq 1$. Therefore each formula is consistently lengthening map by virtue of this and Theorem 3.3.2. Hence by Lemma 3.3.6, it is chain-continuous. □

The importance of this theorem is in demonstrating that fixed-points can be attained through finite iteration, providing a basis for model checking. We will define the *approximants* of a GTL fixed-point as follows:

Definition 3.3.9 (Approximants)**.** *Let the nth approximant of $\mu Z \,.\, f$, written $\mu^n Z \,.\, f$, be defined inductively as:*

$$\begin{aligned} \mu^0 Z \,.\, f &:= \mathrm{ff} \\ \mu^{n+1} Z \,.\, f &:= f[(\mu^n Z \,.\, f)/Z] \end{aligned}$$

Continuity allows us to apply the Knaster-Tarski Theorem to show that these approximants converge to the corresponding fixed-point:

Corollary 3.3.10 (to Theorem 3.3.8)**.** *By the Knaster-Tarski Theorem on continuous maps, the GTL fixed-point $\mu Z \,.\, f$ is the least fixed-point of $\mathcal{R}_{f,\rho,Z}$, and the symbolically lifted union of the approximants:*

$$\begin{aligned} (\mathit{fix}(\mathcal{R}_{f,\rho,Z}))\langle\nu\rangle &= \parallel \mu Z \,.\, g \parallel_\rho\langle\nu\rangle \\ &= \bigcup_{n\geq 0}(\parallel \mu^n Z \,.\, f \parallel_\rho\langle\nu\rangle) \end{aligned}$$

3.4 Syntactic Sugar

To describe some common temporal patterns, we define the past-time equivalents of the Linear Temporal Logic [Pnu77] operators *Finally* and strong *Until*. *Previously* f, written $\mathbf{P}f$, asserts that f held at some point in the past. The formula f *Since* g, written $f\ \mathbf{S}\ g$, requires that f holds at every point backward in time until some point where g has held.

Definition 3.4.1. *The two temporal operators* $\mathbf{P}$ *and* $\mathbf{S}$ *are defined by:*

$$\begin{aligned} \mathbf{P}f &\quad:=\quad \mu Z\,.\,f \vee \mathbf{Y}Z \\ f\,\mathbf{S}\,g &\quad:=\quad \mu Z\,.\,g \vee (f \wedge \mathbf{Y}Z) \end{aligned}$$

Notice that we cannot define an equivalent to LTL's *Globally* operator, since our logic cannot express greatest fixed-points. This is in keeping with current applications of GSTE, which do not calculate any greatest fixed-points.

We define the following notations for symbolic quantification:

$$\begin{array}{lll} (\exists u\,.\,f) & \text{for} & f(u := \mathrm{T}) \vee f(u := \mathrm{F}) \\ (\forall u\,.\,f) & \text{for} & f(u := \mathrm{T}) \wedge f(u := \mathrm{F}) \end{array}$$

We will also write 'n **is** Q' as short-hand for $Q \ \rightarrow\ \mathsf{n} \,|\, \neg\mathsf{n}$. It is also useful to have a notation for describing the values on data buses. Data buses are commonly modeled as vectors of nodes, and conventionally written in the form $\mathsf{b}[n:0]$, which represents the group $\mathsf{b}[n], \mathsf{b}[n-1], \ldots, \mathsf{b}[0]$ of nodes. We will similarly write $v[n:0]$ for the vector of variables consisting of $v[n], v[n-1], \ldots, v[0]$. We can then use the following syntax for the values on buses:

$$\mathsf{b}[n:0]\ \textbf{is}\ v[n:0] \qquad \text{for} \qquad (\mathsf{b}[n]\ \textbf{is}\ v[n]) \wedge \ldots \wedge (\mathsf{b}[0]\ \textbf{is}\ v[0])$$

3.5 Expressing Properties

We will use GTL in order to describe the outlines of symbolic ternary simulations, as well as to provide a sound semantics for the properties being verified by those simulations. The top level of simulation must be described with formulas that are *closed* with respect to recursion variables, since the fixed-point simulation iterations must be closed for us to be able to simulate them.

Definition 3.5.1. *A GTL formula is closed if every occurrence of a recursion variable is bound by a corresponding* μ*-expression.*

To be able to use GTL to express runs of symbolic ternary simulation, it is necessary to introduce a construct to separate out the *antecedent* and *consequent* parts of the property. This allows us to differentiate those constraints that should be used to drive and define the circuit simulation, versus those constraints that are to be asserted about the simulation result.

Definition 3.5.2 (GTL Properties)**.** *A GTL property is of the form 'antecedent* A *leads to consequent* C*', written* $\mathrm{A} \Rightarrow \mathrm{C}$*, where* A *and* C *are closed formulas of GTL.*

We can now formally define what it means for a model to satisfy a GTL property. A model satisfies a property if every trace of it that satisfies the antecedent also satisfies the consequent.

Definition 3.5.3 (Trace Satisfaction)**.** *A model trace* $\sigma \in S^+$ *satisfies GTL property* $\mathrm{A} \Rightarrow \mathrm{C}$ *if under every indexing variable valuation, the antecedent being satisfied by* σ *implies that the consequent is satisfied by* σ*:*

$$\sigma \models \mathrm{A} \Rightarrow \mathrm{C} \quad \textit{iff} \quad \forall \nu \in \mathcal{V} \,.\, (\sigma \in \| \mathrm{A} \|^{\nu} \Rightarrow \sigma \in \| \mathrm{C} \|^{\nu})$$

Definition 3.5.4 (Property Satisfaction)**.** *A Kripke structure circuit model* $\mathcal{K}_{\mathbf{C}}$ *satisfies GTL property* $\mathcal{P}$ *if every trace of the model satisfies* $\mathcal{P}$*:*

$$\mathcal{K}_{\mathbf{C}} \models \mathcal{P} \quad \textit{iff} \quad \forall \sigma \in \mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \,.\, \sigma \models \mathcal{P}$$

Example The following property can be used to verify a delayed AND-gate with inputs a and b and output o:

$$\mathbf{Y}((\mathsf{a}\ \mathbf{is}\ u) \wedge (\mathsf{b}\ \mathbf{is}\ v)) \Rightarrow \mathsf{o}\ \mathbf{is}\ (u \wedge v)$$

The property can be read as 'whenever node a had value u and node b had value v in the previous time-step, node o should be $(u \wedge v)$'. Notice that the property uses two different forms of conjunction. The first is that of GTL, and expresses that two temporal events have both occurred. The second instance is that of Boolean conjunction, that defines the Boolean function describing the expected value on node o.

Example As another example, consider the following property that we might expect to hold of a simple memory cell, which has write enable input wr, data input node in and output out:

$$\mathbf{Y}((\neg \mathsf{wr})\ \ \mathbf{S}\ \ (\mathsf{wr} \wedge \mathsf{in}\ \mathbf{is}\ u)) \ \ \Rightarrow \ \ \mathsf{out}\ \mathbf{is}\ u$$

This property says that if from the previous time-step backwards no write has occurred since some time when a write occurred with input u, then the output should equal u. This property is equivalent to that of the assertion graph shown in Figure 2.13 (page 26).

3.6 Set-Based Model Checking

This section introduces set-based model checking for GTL properties, illustrating the algorithm behind our intended simulation approach, whilst keeping issues of abstraction aside. The algorithm will then be adapted in the subsequent section, for the ternary vector data structures required for abstract simulation.

The GTL property $\mathrm{A} \Rightarrow \mathrm{C}$ is verified by using the shape of antecedent formula A to control and drive the flow of simulation. The resulting set of states is then checked for containment against the consequent C. Since the logical constructs of GTL are in one-to-one correspondence with the atomic steps of symbolic ternary simulation, the model checking procedure follows the shape of the formula directly. Like each formula of GTL, each intermediate set of states produced is symbolically indexed.

3.6.1 Preliminaries

First of all, we formally define the *image* of a formula to describe what it is that the simulation of a formula actually calculates.

Definition 3.6.1 (Formula Image)**.** *The image of formula f, with respect to Kripke structure $\mathcal{K}_{\mathbf{C}}$ and context ρ, is the symbolic set of states that are the last states in those model traces of $\mathcal{K}_{\mathbf{C}}$ that satisfy f in ρ:*

$$\mathrm{im}_{\mathcal{K}_{\mathbf{C}},\rho}(f)\langle\nu\rangle \quad := \quad \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \parallel f \parallel^{\nu}_{\rho})$$

We will often omit $\mathcal{K}_{\mathbf{C}}$, since it is assumed to be constant throughout. The states of a ternary simulation consist of representations of the set of states satisfied by an antecedent up until a particular time point. Once simulation is complete, each of these sets can be checked for containment against some consequent condition. For this reason, the consequents that we can use must not be temporal. We will call this class of formulas *atemporal*.

Definition 3.6.2 (Atemporal)**.** *A GTL formula is atemporal if it does not contain any instances of* $\mathbf{Y}$.

Recall that every recursion variable used within a fixed-point expression must be bound by an instance of $\mathbf{Y}$ (Definition 3.2.1). Therefore any fixed-point variables used within a atemporal formula must not contain recursion variables, so each such fixed-point may therefore be replaced by its inner expression. For example, $\mu Z\,.\,f$ may be replaced by f if Z does not occur freely in f.

As would be expected, whether or not a word satisfies an atemporal formula is determined purely by the last state in the word:

Lemma 3.6.3. *For any atemporal formula f and word $t \in S^+$, t satisfies f if and only if $\mathrm{last}(t)$ satisfies f as a singleton word.*

Proof. The proof is in Appendix A.3. □

GTL verification is different from either STE or assertion graph verification because we only check for conditions using the *end-state* of the simulation. In STE the consequent can refer to any point along a fixed-length finite trace of simulation states. But this approach is not possible in general for GTL, because properties can be unbounded.

In GSTE assertion graphs, consequents can be applied at any point in the simulation, and not just at the end. We have avoided this for GTL properties because we believe that it simplifies the structure of properties and makes them more amenable to reasoning. Since consequents in GTL properties are only linked to the *end* of simulation, and not the intermediate states, we are freer to transform the simulation without being concerned about additional side-effects. If we still wish to verify multiple consequent conditions then we can instead do so by using multiple properties.

3.6.2 Set-Based Simulation

Similar to the standard model checking approach for calculating the set of states that satisfy a formula of Computation Tree Logic (CTL) [CES86], we define simulation for GTL by structural recursion on the syntax of the antecedent formula. The algorithm results in a symbolic set of states from $\mathcal{V} \to 2^S$ that is an upper-approximation of the image of a formula. We use a *simulation context* $\tau : \mathcal{F} \to (\mathcal{V} \to 2^S)$ to assign values to the free recursion variables in a formula and accumulate fixed-point values.

Definition 3.6.4 (GTL Simulation)**.** *Figure 3.3 defines simulation for GTL formulas. The result of simulation is written $[f]^{\nu}_{\tau}$, where f is the formula being simulated, ν is the indexing variables valuation, and τ is the simulation context. Fixed-points are calculated using the recursive function* fix*, defined by:*

$$\mathrm{fix}\ f\ x \ = \ \mathrm{if}\ ((f\ x) = x)\ \mathrm{then}\ x\ \mathrm{else}\ (\mathrm{fix}\ f\ (fx))$$

$$\begin{array}{lcl}
[\,\mathsf{tt}\,]^{\nu}_{\tau} & = & S \\
[\,\mathsf{ff}\,]^{\nu}_{\tau} & = & \emptyset \\
[\,\mathsf{n}\,]^{\nu}_{\tau} & = & \{s \in S \mid s(\mathsf{n}) = 1\} \\
[\,\neg\mathsf{n}\,]^{\nu}_{\tau} & = & \{s \in S \mid s(\mathsf{n}) = 0\} \\
[\,f \vee g\,]^{\nu}_{\tau} & = & [\,f\,]^{\nu}_{\tau} \cup [\,g\,]^{\nu}_{\tau} \\
[\,f \wedge g\,]^{\nu}_{\tau} & = & [\,f\,]^{\nu}_{\tau} \cap [\,g\,]^{\nu}_{\tau} \\
[\,\mathbf{Y}f\,]^{\nu}_{\tau} & = & \mathrm{post}([\,f\,]^{\nu}_{\tau}) \\
[\,Z\,]^{\nu}_{\tau} & = & \tau(Z) \\
[\,\mu Z\,.\,f\,]_{\tau} & = & \mathrm{fix}\,(\lambda S\,.\,[\,f\,]_{\tau[Z \mapsto S]})\,(\lambda \nu'.\emptyset) \\
[\,Q\,\rightarrow\,f \mid g\,]^{\nu}_{\tau} & = & \text{if } Q\langle\nu\rangle \text{ then } [\,f\,]^{\nu}_{\tau} \text{ else } [\,g\,]^{\nu}_{\tau} \\
[\,f(u := Q)\,]^{\nu}_{\tau} & = & [\,f\,]^{\nu[u \mapsto Q\langle\nu\rangle]}_{\tau}
\end{array}$$

Figure 3.3: Set-Based Simulation

3.6.2.1 Termination

For fixed-point $\mu Z\,.\,f$, we first simulate f in a context in which Z is assigned the empty set of states in every index valuation. We then use this result as a new assignment to Z to re-simulate f. This is repeated until equality is reached between iterations. Since our domain is finite, termination is ensured by the monotonicity of each simulation:

Lemma 3.6.5. *For every formula f, simulation terminates and is monotonic in the simulation context of each recursion variable.*

Proof. This is shown by induction, first on the number of fixed-points in a formula, and second by the length of the formula, and is given in full in Appendix A.3 □

3.6.3 Checking Properties

In order to use simulation to verify a GTL property, we simulate both the antecedent and the atemporal consequent of the property, and then assert containment:

Definition 3.6.6 (Set-Based Model Checking)**.** *Set-based model checking of a GTL property succeeds if and only if the antecedent simulation is contained within the consequent simulation:*

$$\mathbf{MC}_{set}(\mathcal{K}_{\mathbf{C}}, \mathrm{A} \Rightarrow \mathrm{C}) \quad \textit{iff} \quad [\,\mathrm{A}\,] \subseteq^{\forall\mathcal{V}} [\,\mathrm{C}\,]$$

We will now justify this model checking procedure. We do this by demonstrating relationships between the results of simulation and the simulated formula's image.

Lemma 3.6.7. *The simulation of any closed formula, f, is an upper-approximation of its image:*

$$\mathrm{im}(f) \ \subseteq^{\forall\mathcal{V}} \ [\, f\,]$$

Proof. In order to demonstrate this for closed formulas of GTL, we require the following inductive property of sub-formulas: for any formula f, trace recursion context ρ and simulation recursion context τ, if $\mathrm{im}_\rho(Z) \subseteq^{\forall\mathcal{V}} [\, Z\,]_\tau$ for every recursion variable Z, then $\mathrm{im}_\rho(f) \subseteq^{\forall\mathcal{V}} [\, f\,]_\tau$.

This property extends the lemma to formulas with free recursion variables, which requires the given simulation context to also contain upper-approximations of the image of the given corresponding semantic recursion context. A detailed proof of this property is in Appendix A.3, and proceeds by induction, ordering first by the length of a formula, and secondly by the number of μ-expressions in a formula. □

Lemma 3.6.8. *For any atemporal GTL formula f, the simulation and image are equal:*

$$\mathrm{im}(f) \ = \ [\, f\,]$$

Proof. The proof is very similar to that of Lemma 3.6.7, and is given in full in Appendix A.3.1. □

These two lemmas allow us to use a simulation containment check in order to verify that a circuit model satisfies a given GTL property:

Theorem 3.6.9. *If set-based model checking succeeds for property* $\mathrm{A} \Rightarrow \mathrm{C}$*, with atemporal consequent* C*, then the circuit satisfies* $\mathrm{A} \Rightarrow \mathrm{C}$.

$$\mathbf{MC}_{set}(\mathcal{K}_{\mathbf{C}}, \mathrm{A} \Rightarrow \mathrm{C}) \ \ \textit{implies} \ \ \mathcal{K}_{\mathbf{C}} \ \models \ \mathrm{A} \Rightarrow \mathrm{C}$$

Proof. Model checking succeeds when $[\,\mathrm{A}\,] \subseteq^{\forall\mathcal{V}} [\,\mathrm{C}\,]$. Lemma 3.6.7 ensures that $\mathrm{im}(\mathrm{A}) \subseteq^{\forall\mathcal{V}} [\,\mathrm{A}\,]$. Since C is atemporal then $\mathrm{im}(\mathrm{C}) \ = \ [\,\mathrm{C}\,]$ by Lemma 3.6.8. Therefore $\mathrm{im}(\mathrm{A}) \subseteq^{\forall\mathcal{V}} \mathrm{im}(\mathrm{C})$. Now suppose σ is a trace of the model that satisfies the antecedent in valuation ν: $\sigma \in \mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \|\ \mathrm{A}\ \|^\nu$. Then $\mathrm{last}(\sigma) \in \mathrm{im}(\mathrm{A})\langle\nu\rangle$, so $\mathrm{last}(\sigma) \in \mathrm{im}(\mathrm{C})\langle\nu\rangle$. Now since the consequent is atemporal, $\sigma \in \mathrm{im}(\mathrm{C})\langle\nu\rangle$ by Lemma 3.6.3. Hence every trace of the model satisfies the property in every valuation, so $\mathcal{K}_{\mathbf{C}} \models \mathrm{A} \Rightarrow \mathrm{C}$. □

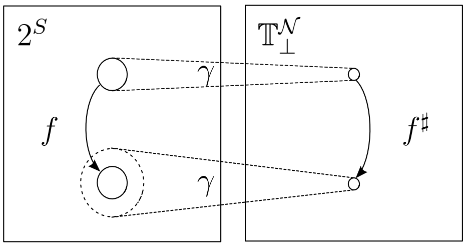


Figure 3.4: Soundness of an Abstract Operation

3.7 Abstract Model Checking

Abstract model checking is the modification of set-based model checking to make use of symbolic ternary simulation. These changes counter the prohibitive state-explosion problem that would accompany explicit set-based model checking of sizable designs. Using the ternary vector abstract state set representations requires abstract interpretations of $\cup, \cap$ and post. The next section describes how each of these operations are calculated in practice, and demonstrates their soundness, and, in some cases, completeness. This will allow us to justify abstract model checking.

3.7.1 Abstract Simulation Operations

The foundations for the abstract operations in ternary simulation have already been given in Section 2.5.3.2, using the terminology of abstract interpretation theory (Section 2.3.3). Recall that elements in the lattice of ternary vectors, $\mathbb{T}_\perp^\mathcal{N}$ are used to represent sets of circuit states, 2^S, and that we can reason about abstraction using the Galois connection (α, γ).

We will say that an abstract map $f^\sharp : \mathbb{T}_\perp^\mathcal{N} \rightarrow \mathbb{T}_\perp^\mathcal{N}$, on ternary vectors, is a *sound approximation* of a concrete map $f : 2^S \rightarrow 2^S$, on sets of circuit states, if, for every $a \in \mathbb{T}_\perp^\mathcal{N}$, the set represented by $f^\sharp(a)$ is a superset of the image under f of the set represented by a: $f(\gamma(a)) \subseteq \gamma(f^\sharp(a))$. This is illustrated in Figure 3.4. We will say that an abstract map is furthermore *complete* if, $f(\gamma(a)) = \gamma(f^\sharp(a))$.

Throughout this section, in order to describe ternary vectors succinctly, we chose to define them using a lambda-expression notation. We will write $\lambda \mathsf{n}.f(\mathsf{n})$ for the ternary vector a that satisfies $a(\mathsf{n}) = f(\mathsf{n})$ for all n in $\mathcal{N}$.

We will describe the calculation of each simulation operation in detail, with the exception of the propagation operator, $|\cdot|$ (see Section 2.3.1), which is beyond the scope of our work. We will merely make the assumption that this operation does not change the set of sets being represented: $\gamma(|s|) = \gamma(s)$. This is fair in practice, since propagation adds information to the ternary vectors that can be deduced directly from the circuit constraints. We will also describe in detail how over-constraint conditions interact with simulations—something which is not covered in detail by other STE literature.

3.7.1.1 True and False

We will use $\bot$ to represent the empty set of states. Since $\gamma(\bot) = \emptyset$, it is clear that this is a sound and complete representation. For the set of all consistent states, we will use the propagation of the ternary vector that consists of X at every node. Again, this representation is sound and complete since $\gamma(\mathsf{XX}\ldots\mathsf{X}) = S$.

3.7.1.2 Atomic Propositions

For the image of GTL's atomic propositions, we use the following abstract representation:

Definition 3.7.1. *Let* $\mathsf{n}\ \mathsf{is}^\sharp\ b$ *be the propagation of the ternary vector that has Boolean value* b *at node* n *and* X *everywhere else:*

$$\mathsf{n}\ \mathsf{is}^\sharp\ b \quad := \quad \left| \lambda\mathsf{m}. \begin{cases} b & \textit{if}\ \mathsf{n} = \mathsf{m} \\ \mathsf{X} & \textit{otherwise} \end{cases} \right|$$

Lemma 3.7.2. $\mathsf{n}\ \mathsf{is}^\sharp\ 1$ *and* $\mathsf{n}\ \mathsf{is}^\sharp\ 0$ *are sound and complete representations for* $[\,\mathsf{n}\,]^\nu$, *and* $[\,\neg\mathsf{n}\,]^\nu$*, respectively.*

Proof.

$$\begin{aligned}
\gamma(\mathsf{n}\ \mathsf{is}^\sharp\ 1) &= \gamma\left(\left| \lambda\mathsf{m}. \begin{cases} 1 & \text{if } \mathsf{n} = \mathsf{m} \\ \mathsf{X} & \text{otherwise} \end{cases} \right| \right) && \text{(Definition 3.7.1)} \\
&= \gamma\left(\lambda\mathsf{m}. \begin{cases} 1 & \text{if } \mathsf{n} = \mathsf{m} \\ \mathsf{X} & \text{otherwise} \end{cases} \right) && \text{(Assumption)} \\
&= \{s \in S \mid s(n) \sqsubseteq 1 \wedge \forall \mathsf{m} \neq \mathsf{n}\,.\,s(\mathsf{m}) \sqsubseteq \mathsf{X}\} && \text{(Definition 2.3.2)} \\
&= \{s \in S \mid s(n) = 1\} && \text{(Definition 2.3.1)} \\
&= [\,\mathsf{n}\,]^\nu && \text{(Definition 3.6.4)}
\end{aligned}$$

The same proof outline holds for 0. □

3.7.1.3 Union

To approximate union and intersection, we use the *join* $\sqcup$ and *meet* $\sqcap$, respectively, of the lattice of ternary vectors. These operations have already been defined for single quaternary values $\mathsf{X}, 0, 1$ and $\bot$ in Figure 2.7. For elements of the ternary lattice, the join for vectors can be calculated as the point-wise join of the ternary value assigned to each circuit node:

Definition 3.7.3. *Join on the ternary vector lattice for ternary vectors* $a, b \in \mathbb{T}_\bot^{\mathcal{N}}$ *is defined by:*

$$a \sqcup b \quad := \begin{cases} a & \textit{if } b = \bot \\ b & \textit{if } a = \bot \\ \lambda\mathsf{n}\ .\ a(\mathsf{n}) \sqcup b(\mathsf{n}) & \textit{otherwise} \end{cases}$$

Lemma 3.7.4. *Join is a sound, but not complete, abstract interpretation of set union.*

Proof. Let a and b be elements of the ternary vector lattice, $\mathbb{T}_\bot^{\mathcal{N}}$.

Case $a = \bot$. Then $\gamma(a) \cup \gamma(b) = \emptyset \cup \gamma(b) = \gamma(b) = \gamma(a \sqcup b)$.

Case $b = \bot$. Then $\gamma(a) \cup \gamma(b) = \gamma(a) \cup \emptyset = \gamma(a) = \gamma(a \sqcup b)$.

Case $a \neq \bot \wedge b \neq \bot$. Then:

$$\begin{aligned} s \in \gamma(a) \cup \gamma(b) \;&\Rightarrow\; (\forall \mathsf{n}\,.\, s(\mathsf{n}) \sqsubseteq a(\mathsf{n})) \vee (\forall \mathsf{n}\,.\, s(\mathsf{n}) \sqsubseteq b(\mathsf{n})) \\ &\Rightarrow\; (\forall \mathsf{n}\,.\, s(\mathsf{n}) \sqsubseteq a(\mathsf{n}) \sqcup b(\mathsf{n})) \\ &\Rightarrow\; s \in \gamma(a \sqcup b) \end{aligned}$$

Hence $\gamma(a) \cup \gamma(b) \subseteq \gamma(a \sqcup b)$, as required for soundness. The following case demonstrates that $\sqcup$ is not complete:

$$\gamma(\mathsf{10}) \cup \gamma(\mathsf{01}) \;=\; \{\mathsf{10}, \mathsf{01}\} \subset \{\mathsf{00}, \mathsf{10}, \mathsf{01}, \mathsf{11}\} \;=\; \gamma(\mathsf{XX}) \;=\; \gamma(\mathsf{10} \sqcup \mathsf{01})$$

□

3.7.1.4 Intersection

As a form of abstract set intersection, we use the meet of the lattice of propagations of ternary vectors. The meet is calculated by performing the point-wise meet on nodes, and then a propagation step. We use a propagation step at this stage, because we can deduce new information about the values of circuit nodes. This is because the knowledge that two conditions *both* hold allows us to increase the frontier of what we know about the circuit state. For example, consider the simple case of an AND-gate. If we know that *both* of its inputs are high, then a propagation step allows us to deduce the additional constraint that the output must be high. The meet is $\bot$ when either of its arguments is $\bot$, or if it introduces an inconsistency. This is summarized in the following definition:

Definition 3.7.5. *Meet on the ternary lattice is calculated by:*

$$a \sqcap b \ := \begin{cases} \bot & \textit{if} \quad a = \bot \vee b = \bot \\ & \qquad \vee \ \exists \mathsf{n} \,.\, a(\mathsf{n}) \sqcap b(\mathsf{n}) = \bot \\ & \qquad \vee \ (|\lambda \mathsf{n}.a(\mathsf{n}) \sqcap b(\mathsf{n})| = \bot) \\ |\lambda \mathsf{n} \ . \ a(\mathsf{n}) \sqcap b(\mathsf{n})| & \textit{otherwise} \end{cases}$$

Lemma 3.7.6. *Meet is a sound and complete abstract interpretation of set intersection.*

Proof. We show that $\gamma(a) \cap \gamma(b) = \gamma(a \sqcap b)$.

Case $a = \bot \vee b = \bot$. Then $\gamma(a) \cap \gamma(b) = \emptyset = \gamma(\bot) = \gamma(a \sqcap b)$.

Case $\exists \mathsf{n} \,.\, a(\mathsf{n}) \sqcap b(\mathsf{n}) = \bot$. Then since a and b disagree at node n, the sets of states represented do not overlap, so $\gamma(a) \cap \gamma(b) = \emptyset = \gamma(\bot) = \gamma(a \sqcap b)$.

Case $|\lambda \mathsf{n}.a(\mathsf{n}) \sqcap b(\mathsf{n})| = \bot$. Then by the assumption on $|\cdot|$, $\gamma(\lambda \mathsf{n}.a(\mathsf{n}) \sqcap b(\mathsf{n})) = \emptyset$, so no consistent states satisfy the conditions of both a and b. Therefore, $\gamma(a) \cap \gamma(b) = \emptyset = \gamma(\bot) = \gamma(a \sqcap b)$.

Case Otherwise:

$$\begin{aligned} s \in \gamma(a) \cap \gamma(b) \quad & \iff \quad (\forall \mathsf{n} \,.\, s(\mathsf{n}) \sqsubseteq a(\mathsf{n})) \wedge (\forall \mathsf{n} \,.\, s(\mathsf{n}) \sqsubseteq b(\mathsf{n})) \\ & \iff \quad \forall \mathsf{n} \,.\, (s(\mathsf{n}) \sqsubseteq a(\mathsf{n}) \wedge s(\mathsf{n}) \sqsubseteq b(\mathsf{n})) \\ & \iff \quad \forall \mathsf{n} \,.\, (s(\mathsf{n}) \sqsubseteq a(\mathsf{n}) \sqcap b(\mathsf{n})) \\ & \iff \quad s \in \gamma(a \sqcap b) \end{aligned}$$ □

3.7.1.5 Post-Image

The post-image function post is interpreted using the abstract post-image function $\mathrm{post}^\sharp$ that applies one step of forward symbolic ternary simulation. Recall from Section 2.3.1 that this calculation consists of propagating information through the circuit using the ternary logic, and then simulating a time-step by transferring values across delay elements. We will not formally demonstrate that $\mathrm{post}^\sharp$ is a sound abstract interpretation of post, due to the additional work required to formally model gate-level

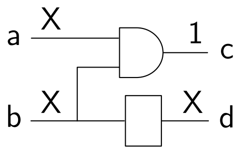


Figure 3.5: Example of Post-Image Over-Approximation

simulation. The thrust of the argument, however, is that $\gamma(\mathrm{post}^\sharp(a))$ includes all the successor states of $\gamma(a)$, since the abstract post-image calculation transfers constraints across from the input of delay nodes to their output.

We note that $\mathrm{post}^\sharp$ is not a complete interpretation of post, since it is possible to lose information due to the lack of backwards constraint propagation. For example, consider the ternary vector a illustrated in Figure 3.5. Since the output of the AND gate is 1, it follows that the second input, b, must also be 1. Therefore, in every state of $\mathrm{post}(\gamma(a))$, it must be that the output d is 1. Because simulation only propagates values forward, however, this constraint is not maintained.

3.7.1.6 Containment

Finally, the set inclusion test for the consequent is replaced by a point-wise application of the 'is less approximate than' order on the lattice of ternary values, $\sqsubseteq$. We interpret this relation into the abstract Boolean domain where false is more approximate than true, since model checking will be sound, but not complete.

Definition 3.7.7. *The abstract containment operation, $\subseteq^\sharp$, is defined by:*

$$a \subseteq^\sharp c \quad := \begin{cases} \mathrm{true} & \textit{if } a = \bot \\ \mathrm{false} & \textit{if } a \neq \bot \wedge c = \bot \\ \forall \mathsf{n} \in \mathcal{N} \ . \ a(\mathsf{n}) \sqsubseteq c(\mathsf{n}) & \textit{otherwise} \end{cases}$$

Lemma 3.7.8. *The abstract containment operation is sound.*

$$a \subseteq^\sharp c \quad \textit{implies} \quad \gamma(a) \subseteq \gamma(c)$$

Proof. We examine the cases in which $a \subseteq^\sharp c$ holds.

Case $a = \bot$: Then $\gamma(a) = \emptyset$, so the result holds trivially.

Case $a \neq \bot \wedge c \neq \bot \wedge (\forall \mathsf{n} \in \mathcal{N} \ . \ a(\mathsf{n}) \sqsubseteq c(\mathsf{n}))$:

$$\begin{aligned} & \forall \mathsf{n} \in \mathcal{N} \, . \, a(\mathsf{n}) \sqsubseteq c(\mathsf{n}) \\ \Rightarrow \; & \forall s \in S \, . \, (\forall \mathsf{n} \in \mathcal{N} \, . \, s(\mathsf{n}) \sqsubseteq a(\mathsf{n})) \Rightarrow (\forall \mathsf{n} \in \mathcal{N} \, . \, s(\mathsf{n}) \sqsubseteq c(\mathsf{n})) \\ \Longleftrightarrow \; & \forall s \in S \, . \, s \in \gamma(a) \Rightarrow s \in \gamma(c) \\ \Longleftrightarrow \; & \gamma(a) \subseteq \gamma(c) \end{aligned}$$

□

Set Construct	Abstract Interpretation	Sound	Complete
S	$\lambda \mathsf{n}.\mathsf{X}$	✓	✓
$\emptyset$	$\bot$	✓	✓
$[\,\mathsf{n}\,]^{\nu}$	$\mathsf{n}\ \mathsf{is}^{\sharp}\ 1$	✓	✓
$[\,\neg\mathsf{n}\,]^{\nu}$	$\mathsf{n}\ \mathsf{is}^{\sharp}\ 0$	✓	✓
$\cup$	$\sqcup$	✓	✗
$\cap$	$\sqcap$	✓	✓
post	$\mathrm{post}^{\sharp}$	✓	✗
$\subseteq$	$\subseteq^{\sharp}$	✓	✗

Table 3.1: Abstract Interpretation of Set Operations

Lemma 3.7.9. *The abstract containment operation is not complete.*

Proof. Consider the case where $S = \{\mathsf{00}\}$, $a = \mathsf{01}$ and $c = \bot$. Here the ternary vector a is inconsistent with the circuit constraints described by S, so $\gamma(a) = \emptyset$. But $\gamma(c) = \emptyset$. Therefore $\gamma(a) \subseteq \gamma(c)$, even though $a \subseteq^{\sharp} b$ does not hold. □

For STE, containment for ternary traces can be made complete by making use of bounded model checking instead of point-wise ternary checks [TG06]. We expect that containment can also be made complete in our set-based view of GTL simulation, by introducing similar forms of reasoning.

3.7.2 Abstract Simulation

We now have all the abstract operations necessary in order to perform abstract model checking. Table 3.1 shows a summary of the atomic set operations used, together with their abstract interpretations and characteristics. We adapt the set-based algorithm to use these abstract interpretations, resulting in *abstract simulation*, defined in Fig. 3.6.

3.7.2.1 Termination

As with set-based simulation, we demonstrate the termination of fixed-point calculations by showing that abstract simulation is monotonic over a finite domain.

Lemma 3.7.10. *Abstract simulation terminates and is monotonic in each recursion variable, i.e.* $\lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto U]}$ *is monotonic with respect to* $\sqsubseteq^{\forall\mathcal{V}}$ *as a function of* U.

Proof. Proof is by induction, ordering first by the number of fixed-points in f, and secondly by the length of f, and is given in Appendix A.4. □

$$
\begin{array}{lcl}
\lfloor \mathsf{tt} \rfloor_\sigma^\nu & = & \lambda \mathsf{m}.\mathsf{X} \\
\lfloor \mathsf{ff} \rfloor_\sigma^\nu & = & \bot \\
\lfloor \mathsf{n} \rfloor_\sigma^\nu & = & \mathsf{n}\ \mathsf{is}^\sharp\ 1 \\
\lfloor \neg\mathsf{n} \rfloor_\sigma^\nu & = & \mathsf{n}\ \mathsf{is}^\sharp\ 0 \\
\lfloor f \vee g \rfloor_\sigma^\nu & = & \lfloor f \rfloor_\sigma^\nu \sqcup \lfloor g \rfloor_\sigma^\nu \\
\lfloor f \wedge g \rfloor_\sigma^\nu & = & \lfloor f \rfloor_\sigma^\nu \sqcap \lfloor g \rfloor_\sigma^\nu \\
\lfloor \mathbf{Y} f \rfloor_\sigma^\nu & = & \mathrm{post}^\sharp(\lfloor f \rfloor_\sigma^\nu) \\
\lfloor \mu Z \,.\, f \rfloor_\sigma^\nu & = & \mathrm{fix}\ (\lambda S \,.\, \lfloor f \rfloor_{\tau[Z \mapsto S]})\ (\lambda \nu'.\bot) \\
\lfloor Q \ \rightarrow\ f \mid g \rfloor_\sigma^\nu & = & \text{if } Q\langle \nu \rangle \text{ then } \lfloor f \rfloor_\sigma^\nu \text{ else } \lfloor g \rfloor_\sigma^\nu \\
\lfloor f(u := Q) \rfloor_\sigma^\nu & = & \lfloor f \rfloor_\sigma^{\nu[u \mapsto Q]}
\end{array}
$$

Figure 3.6: Abstract Simulation

3.7.3 Checking Properties

We can now define what it means for abstract model checking to succeed:

Definition 3.7.11. *Abstract model checking succeeds when the abstract simulation of the antecedent is contained within the abstract simulation of the consequent for every variable valuation:*

$$\mathbf{MC}(\mathcal{K}_{\mathbf{C}}, \mathrm{A} \Rightarrow \mathrm{C}) \quad \textit{iff} \quad \lfloor \mathrm{A} \rfloor^\nu \subseteq^\sharp \lfloor \mathrm{C} \rfloor^\nu \quad \textit{for every } \nu \in \mathcal{V}$$

Using our results on the soundness of our abstract operations, we can show that abstract simulation consistently over-approximates:

Lemma 3.7.12. *If the value assigned to each recursion variable $Z \in \mathcal{F}$ in abstract simulation context σ is an upper-approximation of its value in concrete simulation context τ, then abstract simulation of any formula f in this context will result in an upper-approximation of the concrete simulation of the same formula:*

$$(\forall Z \in \mathcal{F} \,.\, [\, Z \,]_\tau \ \subseteq^{\forall\mathcal{V}}\ \gamma(\lfloor Z \rfloor_\sigma)) \quad \textit{implies} \quad [\, f \,]_\tau \ \subseteq^{\forall\mathcal{V}}\ \gamma(\lfloor f \rfloor_\sigma)$$

Proof. The previous section has shown that every operation of abstract simulation is a sound approximation of set-based simulation. This lemma is a direct result of applying this observation inductively over the length of the simulation. □

Model checking verifies that every trace which satisfies the antecendent must also satisfy the consequent. Therefore the consequent restricts the accepted set of behaviours, so the simulation of the abstract consequent image must not approximate. Since $\sqcup$ is not a complete interpretation of $\vee$, it follows that abstract model checking is sound only when the consequent under consideration does not contain disjunction.

This requirement rules out properties that express non-deterministic outcomes using disjunction. Since we are performing verification using simulation over a deterministic model, these types of properties typically require us to 'determinize' them via case-splitting. For example, consider a reverse OR-gate property such as:

$$\mathsf{input} \quad \Rightarrow \quad (\mathsf{output}_1) \vee (\mathsf{output}_2)$$

Simulating the consequent condition $(\mathsf{output}_1) \vee (\mathsf{output}_2)$ for the two outputs involved will result in X being assigned to both of them, so that model checking succeeds no matter what the input. Unlike antecedent simulation, this form of consequent over-approximation is unsound, because it weakens the verification. Such a property might instead be soundly verified by

$$\mathbf{MC}(\mathcal{K}_{\mathbf{C}}, \mathsf{input} \wedge A \Rightarrow \mathsf{output}_1) \quad \vee \quad \mathbf{MC}(\mathcal{K}_{\mathbf{C}}, \mathsf{input} \wedge \neg A \Rightarrow \mathsf{output}_2)$$

where A is the case-split condition required to determine a concrete outcome for the circuit at-hand.

Lemma 3.7.13. *If the value assigned to each recursion variable $Z \in \mathcal{F}$ in abstract simulation context σ is an exact representation of its value in concrete simulation context τ, then abstract simulation of any formula f that does not contain disjunction or Yesterday will result in an exact representation of the concrete simulation of the same formula:*

$$(\forall Z \in \mathcal{F} \,.\, [\,Z\,]_\tau = \gamma(\lfloor\,Z\,\rfloor_\sigma)) \quad \textit{implies} \quad [\,f\,]_\tau = \gamma(\lfloor\,f\,\rfloor_\sigma)$$

Proof. Every operation of abstraction simulation is complete, with the exception of those for disjunction and post-image calculation. This lemma is a direct result of applying this observation inductively over the length of the simulation. □

Finally, we can use this to demonstrate that abstract model checking is sound.

Theorem 3.7.14. *If* C *is atemporal and does not contain disjunction then*

$$\mathbf{MC}(\mathcal{K}_{\mathbf{C}}, \mathrm{A} \Rightarrow \mathrm{C}) \quad \textit{implies} \quad \mathcal{K}_{\mathbf{C}} \models \mathrm{A} \Rightarrow \mathrm{C}$$

Proof. By Lemma 3.7.12, $[\,\mathrm{A}\,]^\nu \subseteq \gamma(\lfloor\,\mathrm{A}\,\rfloor^\nu)$ for any formula A. For atemporal consequents without disjunction, Lemma 3.7.13 ensures $[\,\mathrm{C}\,]^\nu = \gamma(\lfloor\,\mathrm{C}\,\rfloor^\nu)$. Now, if model checking succeeds, then $\lfloor\,\mathrm{A}\,\rfloor^\nu \subseteq^\sharp \lfloor\,\mathrm{C}\,\rfloor^\nu$, which implies $\gamma(\lfloor\,\mathrm{A}\,\rfloor^\nu) \subseteq \gamma(\lfloor\,\mathrm{C}\,\rfloor^\nu)$ by Lemma 3.7.8. Combining these, we have that $[\,\mathrm{A}\,]^\nu \subseteq \gamma(\lfloor\,\mathrm{A}\,\rfloor^\nu) \subseteq \gamma(\lfloor\,\mathrm{C}\,\rfloor^\nu) = [\,\mathrm{C}\,]^\nu$. Thus set-based model checking also succeeds, which in turn implies $\mathcal{K}_{\mathbf{C}} \models \mathrm{A} \Rightarrow \mathrm{C}$ by Theorem 3.6.9. □

As described for STE in [TG06], it is possible to obtain more than justs 'pass' or 'fail' from the ternary containment check if required. If a failure is due to over-approximation, then the failed comparisons will be of the form $\mathsf{X} \sqsubseteq 0$ or $\mathsf{X} \sqsubseteq 1$. But if we have demonstrated a genuine error in the design then there will be a check of the form $1 \sqsubseteq 0$ or $0 \sqsubseteq 1$.

3.8 Related Work

We have presented a specification notation and model checking algorithm for symbolic ternary simulation, using a pure-past linear temporal logic with constructs for symbolic indexing. This section explores how GTL relates to existing notations, both inside and outside the domain of STE-based simulation.

3.8.1 STE Specifications

The most commonly used specification notation for traditional STE is Trajectory Evaluation Logic (TEL) [BS91, SB95]. Like GTL, TEL takes the form of a restricted linear temporal logic, with formulas of the form TRUE, $\mathsf{n\ is\ 0}$, $\mathsf{n\ is\ 1}$, f and f, $Q \rightarrow f$, and *next* f, written $\mathsf{N}f$. Assertions are written in the form $[\mathit{antecedent} \Rightarrow \mathit{consequent}]$. The considerable success of TEL for describing and managing STE verifications [SJO^{+}05, PRBA97] was one factor that inspired us to use similar foundations for GTL.

Unlike GTL, TEL does not use past time. This does not pose the same problems with composition as exist for GSTE, since past and future time operands are easily interchangeable in a bounded setting. The logic also differs in its use of an *if-then* construct, rather than the *if-then-else* construct of GTL. One reason for this is that TEL specifications intuitively feel more like assertions about sets of traces, whereas GTL formulas feel more like the sets of states that they describe. In practice we believe that our approach better represents common simulation cases such as symbolic case-splitting. The semantics of TEL also differs significantly from GTL, as the nodes in TEL circuits models themselves themselves have ternary states. This model structure derives from early use of STE for switch-level circuitry, where charge is not modeled as binary, but can include an extra logically neutral physical state corresponding to a node that is driven neither high nor low. This was part of the origin of the use of the value X. But binary models suffice at the gate-level targeted by GSTE, and there are significant benefits to staying within in a binary model.

As STE evolved to deal with richer classes of properties, there were several intermediate specification notations proposed. Hazelhurst describes a variant of TEL with arbitrary negation and an additional *until* operator, implemented via a fixed-point calculation. This goes some way toward the expressiveness of GSTE and our own approach. But like TEL, the semantics of Hazelhurst's language are also based on partially ordered model states, leading to a four-valued logic semantics that complicates reasoning.

STE was also adapted to include fixed-points by Seger and Bryant in [SB95]. In their notation, TEL assertions are sequentially composed with a '$;$' operator, and a Kleene star syntax is used for iteration. For example, the property $[\mathrm{A}_1 \Rightarrow \mathrm{C}_1]^*; [\mathrm{A}_2 \Rightarrow \mathrm{C}_2]$, is interpreted as "A trace which initially satisfies any number of iterations of A_1 must satisfy the same number of iterations as C_2 and then subsequently satisfy $[\mathrm{A}_2 \Rightarrow \mathrm{C}_2]$". In order to model check such properties directly, fixed-points were introduced into the STE simulation flow.

Regular expressions are not so suitable for GSTE. First, there is no equivalent for GTL conjunction, which is used to describe the branching simulations of compositional GSTE. This does not limit the expressibility of properties, but does limit the description of how a simulation can take place. Second, the representation of fixed-points using the Kleene star approach is less compact. This is because it does not allow shared references to fixed-points as GTL μ-expressions can. This means that regular expressions require nested fixed-points to be repeated explicitly in-line.

The main remaining difference is that regular expressions use concatenation to represent the progression of time, whereas GTL uses the $\mathbf{Y}$ operator. For simple properties the two notations are easily interchangeable, and this largely comes down to a question of aesthetics. The use of the concatenation operator enforces temporal conditions to follow from left to right as they pass from past to future, and is likely to require reduced use of parenthesis. But when we consider what happens during substitution, concatenation and $\mathbf{Y}$ behave quite differently.

For example, consider the condition $(\mathbf{Y} \mathit{reset}) \wedge \mathit{init}$, which, we might express as the regular expression $\mathit{reset}.\mathit{init}$. Suppose we would now like to substitute the condition init with the initialization condition given by $(\mathbf{Y} \mathit{init1}) \wedge \mathit{init2}$, or $\mathit{init1}.\mathit{init2}$. Direct substitution will give us $(\mathbf{Y} \mathit{reset}) \wedge (\mathbf{Y} \mathit{init1}) \wedge \mathit{init2}$ and $\mathit{reset}.\mathit{init1}.\mathit{init2}$, respectively, which are quite different conditions. This shows that GTL formulas are better suited to independent backwards-looking temporal conditions, whereas regular expression concatenation is better suited to describing temporal events with defined start and end points. We believe that the former is more relevant for hardware verification because circuits exhibit a high-degree of branching parallel behaviour with relatively little synchronization between components.

3.8.2 GSTE Specifications

Work by Beatty [BB94] laid the foundations for GSTE by relating circuits to arbitrary state transition graphs and using an individual run of STE to verify each transition in turn. In this work, each transition is specified as an STE assertion containing abstract state that is manually mapped into circuit state. Nelson and Jain [NJB97] introduced labeled transition graphs to connect these assertions, and proposed a generalized STE algorithm to model check multiple transitions in a single model checking run. In [Cho99], Chou shows that this algorithm is incomplete, and proposes an alternative graph satisfaction criterion that better fits the implementation. These graph structures were further formalized and extended to form GSTE *assertion graphs* (see Section 2.5.3.1) by Yang and Seger [YS02, YS03].

Although assertion graphs can be useful for displaying simple properties in a visual manner, the notation suffers from several drawbacks in comparison to GTL. It places strict limitations on the amount of formal reasoning that can be achieved, due to its graphical nature. Some progress has been made on formal reasoning with assertion graphs, but the resulting rules, and their proofs, tend to be complicated, because they deal with unstructured flat representation of what is naturally a graphical concept. In contrast, as we will see in the subsequent chapter, there exist simple reasoning rules for GTL that are clear and intuitive.

Assertion graphs also introduce problems associated with indexing variables. In particular, the variable scoping conventions are difficult to visually parse because the notations used do not positionally enclose the defined boundaries of scope. There are two areas where this is especially true: the use of variable classes with different scoping rules (see Section 2.5.4.3), and the use of knots (see Section 2.5.4.4). In both of these cases, the scoping rules are, in practice, dislocated from the assertion graph structure itself. In contrast, GTL variable scoping is more natural because the notation directly encloses a variable's lexicographical scope. Since it follows conventional logical syntax, the notation is also more familiar.

The forall-semantics of assertion graphs can also cause confusion, because it differs from the typical semantics of standard automata. Some considerable effort is sometimes required to understand the interplay between the conditions asserted by different concurrent paths in a graph. Another common misinterpretation of the assertion graph semantics is that the initial assertion graph state must align with some form of initial hardware state.

3.8.3 CGSTE Specifications

As well as the problems with formal reasoning, the notation of assertion graphs cannot describe the *compositional extensions* of GSTE that have been discussed in Section 2.6. These extensions are currently expressed with one of two different notations. The first is by using the language of *compositional specification* [YS04]. This language was primarily introduced as a theoretical notation, in the form of a process algebra, to explain the compositional GSTE technique. Because of this, it does not cater for certain aspects of GSTE, such as its symbolic nature. Unlike GTL, antecedents and consequents are mixed together in this language within the same temporal structure. We believe this complicates the resulting semantics, since each formula corresponds to both a verification check as well as a simulation description. As a result, the logical constructs are complex and unfamiliar. In contrast, GTL separates antecedent and consequent concerns to maintain the familiar form of propositional logic.

A second specification notation for compositional GSTE is provided by the compositional assertion graph data-types for the properties accepted by Intel's compositional GSTE model checker, provided with the Forte verification platform [SJO^{+}05]. This notation suffers from the opposite problem to the *specification language*—it is suitably expressive for practical use, but it does not have a formal semantics beyond that of the model checker's implementation. Furthermore, our initial experiments have shown that the notation is not closed with respect to certain fundamental forms of composition, and suffers from some counter-intuitive irregularities.

3.8.4 Temporal Logics

Outside the realm of STE, temporal logics [Eme90, MP92] are widely used for the specification of properties for formal verification, as well as for controlling some forms of simulation [BC96]. Of these logics, GTL is most similar to the linear-time mu-calculus [Sti92]. In particular, when GTL's two symbolic indexing operators are removed, the remaining constructs form the negation-free pure-past fragment of the linear-time mu-calculus over finite traces. GTL's particular qualities are born out of the need to align the logic closely with the existing simulation methodology of GSTE. We will comment on each of the logic's aspects in turn.

The most striking feature of our logic is that we only refer to *past* time. The use of both past and future temporal operators is relatively common in other logics, such as in LTL+Past [Eme90] and the ForSpec temporal property specification language [AFF$^+$02]. It has been shown that the addition of these past time operators does not affect expressibility [Gab87], although logics with past time can be exponentially more succinct [LMS02]. There are also arguments that the use of past time operators make specifications more natural to write [LPZ85]. Logics that reference only the past are more rare, although some pure-past temporal logics on finite traces have been used recently for characterizing attacks on security protocols [RLSC05]. Despite it being more conventional to think of properties defined using the future, we believe that past time is the natural way to express properties for GSTE. Since simulations proceed *forward* in time, the simulation state is a reflection on the events that have occurred in the past, not those that will occur in the future. This means that by using a past operator, simulation patterns become compositional. For example, $\mathbf{Y}f$ can be simulated first by simulating f and then calculating the post-image of this simulation step. In contrast, a future time operator would introduce a requirement for us to temporally invert properties before we are able to model check them. Furthermore, since we cannot simulate backwards, a future time operator would introduce predicates that be cannot simulated. The correspondence between past-time logics and executable specifications has been explored by Gabby [Gab87], and work in other fields has resulted in similar observations [FW93].

We have chosen to use a finite trace semantics for GTL. An infinite semantics is often more appropriate for reasoning about ongoing, nonterminating behaviour. There are, however, two reasons why an infinite semantics is not appropriate for GTL. First, since we are reasoning with respect to the past, we are obliged to consider finite words, otherwise traces that start with a state with no predecessor are not included in our analysis. Second, since we are only considering forms of GSTE that describe safety properties, there is no need for us to consider infinite words, and by restricting ourselves to finite words, we obtain a simpler and cleaner semantics, which is one of our main aims. One example of this is that GTL fixed-points are unique, as we will show in detail in the next chapter. This is because the finite lengths of traces enforce bounds on the depth of fixed-point recursion. If we instead consider a logic equivalent to GTL but defined over infinite traces then fixed-point are no longer unique. For example, the equation $Z = \mathsf{n} \wedge \mathbf{Y}Z$ would have the empty set as a least fixed-point, as well as the greatest fixed-point $\{ \sigma \mid \forall i \in \mathbb{N} \,.\, \sigma_i(\mathsf{n}) \}$. With finite traces, the least and greatest fixed-points are equal. The result of this is that we can use uniqueness to reason about

GTL formula. Such reasoning is used by various rules in the subsequent chapter, as well as for the verification methodology that we propose in Chapter 6.

Finite forms of linear-time temporal logic have been used elsewhere [GP02], including for the verification of sizable industrial systems where the abstractions required for liveness reasoning are infeasible [HR01]. Generally these logics raise questions about how temporal operators should be interpreted once they reach the end of a trace. The most common way to deal with this is to introduce weak and strong operators that are true and false, respectively, when a trace is exhausted. Alternatively, an extra undetermined value can be used for the end of traces [RHKR00], but this requires the use of a three-valued meta-logic. In GTL, the absence of negation above temporal operators prevents us from requiring these extra operators.

Another property of GTL is that it is negation free. This aligns the logic with the use of upper-approximation in ternary simulation, where the best approximation of a negated ternary vector is truth itself. By excluding negation, we also limit the expressibility of our logic to those states that we can approximate using simulation. For instance, if we were to adapt our logic and semantics to include arbitrary negation, then the hypothetical formula $\neg\mathbf{Y}\mathrm{true}$, would ideally correspond to the set of states without a predecessor. It is not clear how forward abstract simulation could be used to find such states. Furthermore, our avoidance of negation brings about monotonicity rules in the subsequent chapter that are useful for reasoning about property decomposition and abstraction refinement. We note that similar lack of negation occurs in other logics deriving from abstract interpretations, such as in Schmidt's approach to reasoning about abstraction-interpretation-based static analysis [Sch07]. The constructive aspect of our logic also provides a similar feel to many process algebras [Hoa85, Bar93, Mil82].

Chapter 4

Reasoning With GTL

In this chapter, we use the GTL specification language to develop and categorize *reasoning rules* for managing symbolic ternary simulation. Such rules are of vital importance for enabling large-scale verifications, serving as the glue between different model checking runs and ensuring property reductions are sound. As well as being a contribution in its own right, the development of these rules also validates the work of Chapter 3, by demonstrating that GTL can express the types of transformations that are useful for refining typical GSTE verification approaches.

The first section considers simple rules for *property equivalence* in GTL. These rules form the foundations for simulation optimization, since they express how a simulation can be transformed without changing the property being represented. The second section considers rules for *decomposition*. By splitting a run into several smaller runs that together imply the first, verification can become practically possible. The third and final section explores rules for *abstraction refinement*. Such rules transform a simulation to change the precision with which states are represented.

4.1 Equivalence Rules

In this section, we describe some fundamental rules for GTL that can be used to demonstrate property equivalence. Since formulas of GTL determine both the shape of simulation flow as well as the property being checked, these equivalence rules effectively describe property-preserving simulation transformations. Simulation transformations are useful in practice because they can affect both the efficiency and abstraction level with which the simulation is carried out.

4.1.1 Boolean Connectives

First, we show that GTL follows the normal rules of logic that we would expect from a propositional temporal logic. GTL formulas form a distributive lattice with respect to $\wedge, \vee, \mathtt{tt}$ and $\mathtt{ff}$:

$$
\begin{array}{rclrcll}
\mathtt{tt} \wedge f & = & f & \mathtt{ff} \vee f & = & f & \text{Ones} \\
\mathtt{ff} \wedge f & = & \mathtt{ff} & \mathtt{tt} \vee f & = & \mathtt{tt} & \text{Zeros} \\
f \wedge (g \wedge h) & = & (f \wedge g) \wedge h & f \vee (g \vee h) & = & (f \vee g) \vee h & \text{Associativity} \\
f \wedge g & = & g \wedge f & f \vee g & = & g \vee f & \text{Commutativity} \\
f \vee (f \wedge g) & = & f & f \wedge (f \vee g) & = & f & \text{Absorption}
\end{array}
$$

$$
\left.\begin{array}{rcl}
f \vee (g \wedge h) & = & (f \vee g) \wedge (f \vee h) \\
f \wedge (g \vee h) & = & (f \wedge g) \vee (f \wedge h)
\end{array}\right. \qquad \text{Distributivity}
$$

These rules all follow directly from the trace-set semantics of GTL, and the corresponding rules for set arithmetic. For example,

$$
\begin{array}{rcl}
\| f \vee (g \wedge h) \|_{\rho}^{\nu} & = & \| f \|_{\rho}^{\nu} \cup (\| g \|_{\rho}^{\nu} \cap \| h \|_{\rho}^{\nu}) \\
 & = & (\| f \|_{\rho}^{\nu} \cup \| g \|_{\rho}^{\nu}) \cap (\| f \|_{\rho}^{\nu} \cup \| h \|_{\rho}^{\nu}) \\
 & = & \| (f \vee g) \wedge (f \vee h) \|_{\rho}^{\nu}
\end{array}
$$

The built-in semantics for negated propositions behaves as we would expect negation to, since the value of a node in the final state of a trace is Boolean:

$$
\mathsf{n} \wedge \neg \mathsf{n} \;=\; \mathtt{ff} \qquad\qquad \mathsf{n} \vee \neg \mathsf{n} \;=\; \mathtt{tt}
$$

Although these logical rules are simple, they are useful for simulation simplification, as well as for describing the abstraction refinement transformations to be covered in Section 4.3. Furthermore, these rules are not evident for the existing specification notations of assertion graphs.

4.1.2 Fixed-Points

As we have already shown, GTL fixed-points are the limit of their approximants from below (Corollary 3.3.10). Hence fixed-points can be unrolled with the rule:

$$
\mu Z \,.\, f \;=\; f[(\mu Z \,.\, f)/Z] \qquad (\mu\text{-unroll})
$$

Applying this rule does not directly change the result of simulating the fixed-point. It does, however, mean that last iteration of the fixed-point can be distinguished from the other iterations during the application of subsequent rules, since it has been taken outside of the μ-expression.

Example Suppose we are trying to demonstrate the property $\neg \mathsf{n} \wedge (\mathsf{m}\ \mathbf{S}\ \mathsf{n}) \Rightarrow \mathsf{m}$. Using the definition of *Since* and the fixed-point unrolling rule, we can rewrite the antecedent as follows:

$$\begin{aligned}
\neg \mathsf{n} \wedge (\mathsf{m}\ \mathbf{S}\ \mathsf{n}) &= \neg \mathsf{n} \wedge (\mu Z \,.\, \mathsf{n} \vee (\mathsf{m} \wedge \mathbf{Y} Z)) && \text{(Def. 3.4.1)}\\
&= \neg \mathsf{n} \wedge (\mathsf{n} \vee (\mathsf{m} \wedge \mathbf{Y}(\mu Z \,.\, \mathsf{n} \vee (\mathsf{m} \wedge \mathbf{Y} Z)))) && (\mu\text{-unroll})\\
&= (\neg \mathsf{n} \wedge \mathsf{n}) \vee (\neg \mathsf{n} \wedge \mathsf{m} \wedge \mathbf{Y}(\mu Z \,.\, \mathsf{n} \vee (\mathsf{m} \wedge \mathbf{Y} Z))))\\
&= \mathrm{ff} \vee (\neg \mathsf{n} \wedge \mathsf{m} \wedge \mathbf{Y}(\mu Z \,.\, \mathsf{n} \vee (\mathsf{m} \wedge \mathbf{Y} Z))))\\
&= \mathsf{m} \wedge \neg \mathsf{n} \wedge \mathbf{Y}(\mu Z \,.\, \mathsf{n} \vee (\mathsf{m} \wedge \mathbf{Y} Z)))
\end{aligned}$$

It is quite clear that $\mathsf{m} \wedge \neg \mathsf{n} \wedge \mathbf{Y}(\mu Z \,.\, \mathsf{n} \vee (\mathsf{m} \wedge \mathbf{Y} Z))) \Rightarrow \mathsf{m}$ holds, so the original property must also hold.

By splitting off the last iteration in this way, the unroll rule will also allow us to selectively apply the rules that we will present for abstraction refinement and property decomposition.

GTL fixed points also have the property that they are *unique*, so if $Z = f(\mathbf{Y}Z)$ and $W = f(\mathbf{Y}W)$ then it follows that $Z = W$:

Theorem 4.1.1 (Unique Fixed-Point)**.** *If every free occurrence of Z in f occurs within a $\mathbf{Y}$ operator, then $\mathcal{R}_{f,\rho,Z}$ has a unique fixed-point.*

Proof. Since every instance of Z is bound by $\mathbf{Y}$, the temporal depth of Z in f must be greater than or equal to one. Therefore, by the intermediate result of Theorem 3.3.8:

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(R)) \ \subseteq^{\mathcal{V}}\ \mathcal{R}_{f,\rho,Z}(L^{\leq n-1}(R))$$

Since $L^{\leq n}$ is a lower-closure,

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(R)) \ \subseteq^{\mathcal{V}}\ L^{\leq n}(\mathcal{R}_{f,\rho,Z}(L^{\leq n-1}(R))) \tag{4.1}$$

Now, $\mathcal{R}_{f,\rho,Z}$ is monotonic, so:

$$\mathcal{R}_{f,\rho,Z}(R) \ \supseteq^{\mathcal{V}}\ \mathcal{R}_{f,\rho,Z}(L^{\leq n-1}(R))$$

so, by applying the monotonic function $L^{\leq n}$ to both sides,

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(R)) \ \supseteq^{\mathcal{V}}\ L^{\leq n}(\mathcal{R}_{f,\rho,Z}(L^{\leq n-1}(R))) \tag{4.2}$$

Therefore by Equations 4.1 and 4.2,

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(R)) \ =\ L^{\leq n}(\mathcal{R}_{f,\rho,Z}(L^{\leq n-1}(R))) \tag{4.3}$$

Now suppose R and Q are fixed-points of $\mathcal{R}_{f,\rho,Z}$ and $R \neq Q$. Then w.l.o.g. there is some word σ and valuation $\nu \in \mathcal{V}$ such that $\sigma \in R\langle\nu\rangle \setminus Q\langle\nu\rangle$. Then it must be that $\sigma \in (L^{\leq|\sigma|}(R))\langle\nu\rangle \setminus (L^{\leq|\sigma|}(Q))\langle\nu\rangle$, so $\{m \in \mathbb{N} \mid L^{\leq m}(Q) \neq L^{\leq m}(R)\}$ is non-empty. Let $n = \min\{m \in \mathbb{N} \mid L^{\leq m}(R) \neq L^{\leq m}(Q)\}$. n is therefore the shortest length trace for which R and Q differ.

Now $L^{\leq n}(Q) \neq L^{\leq n}(R)$ implies $L^{\leq n}(\| f \|_{\rho[Z\mapsto Q]}) \neq L^{\leq n}(\| f \|_{\rho[Z\mapsto R]})$ since both Q and R are fixed-points. Therefore

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(L^{\leq n-1}(Q))) \neq L^{\leq n}(\mathcal{R}_{f,\rho,Z}(L^{\leq n-1}(R)))$$

by Equation 4.3. This in turn implies $L^{\leq n-1}(Q) \neq L^{\leq n-1}(R)$, contradicting minimality of n. □

This theorem is useful for demonstrating equivalence between two fixed-point formulas, as we will show in the next section.

4.1.3 Temporal Operators

In this section we will consider rules that apply to the temporal operators $\mathbf{Y}$, $\mathbf{P}$ and $\mathbf{S}$. As a consequence of GTL being a linear logic, the Yesterday operator distributes over the other connectives. For example:

$$\mathbf{Y}f \wedge \mathbf{Y}g = \mathbf{Y}(f \wedge g) \qquad (\mathbf{Y}\text{-dist-}\wedge)$$

$$\mathbf{Y}f \vee \mathbf{Y}g = \mathbf{Y}(f \vee g) \qquad (\mathbf{Y}\text{-dist-}\vee)$$

Although $\mathbf{Y}\mathbb{ff} = \mathbb{ff}$, $\mathbf{Y}\mathbb{tt}$ is not equivalent to $\mathbb{tt}$, since $\mathbb{tt}$ satisfies all traces of length one, but $\mathbf{Y}\mathbb{tt}$ does not. This may seem a counter-intuitive rule, but it is in fact useful, because it accurately matches the characteristics of ternary simulation. Since not every state has a pre-image in our circuit model, the set of all states may differ from its own post-image.

Because *Previous* and *Since* are defined in terms of fixed-points, fixed-point unrolling directly induces the following rules for them:

$$\mathbf{P}f = f \vee \mathbf{YP}f \qquad (\mathbf{P}\text{-unroll})$$

$$f\,\mathbf{S}\,g = g \vee (f \wedge \mathbf{Y}(f\,\mathbf{S}\,g)) \qquad (\,\mathbf{S}\,\text{-unroll})$$

The first of these rules might, for example, be used to case-split based on whether the event described by f occurred in the most recent time-step or not. These also allow us to instantly derive that:

$$\begin{aligned} \mathbf{P}\mathtt{tt} &= \mathtt{tt} && (\mathbf{P}\text{-}\mathtt{tt}) \\ f\,\mathbf{S}\,\mathtt{tt} &= \mathtt{tt} && (\,\mathbf{S}\text{ -}\mathtt{tt}\text{-2}) \\ \mathtt{ff}\,\mathbf{S}\,f &= f && (\,\mathbf{S}\text{ -}\mathtt{ff}\text{-1}) \end{aligned}$$

The distributive law for disjunction, together with the uniqueness of fixed-points, allows us to demonstrate some interesting properties of $\mathbf{P}$ and $\mathbf{S}$. For example, the following two equations for $\mathbf{YP}f$ and $\mathbf{PY}f$ are obtained by simple unrolling and distribution:

$$\mathbf{PY}f = \mathbf{Y}f \vee \mathbf{Y}(\mathbf{PY}f) \qquad (\mathbf{P}\text{-unroll})$$

$$\begin{aligned} \mathbf{YP}f &= \mathbf{Y}(f \vee \mathbf{YP}f) && (\mathbf{P}\text{-unroll}) \\ &= \mathbf{Y}f \vee \mathbf{Y}(\mathbf{YP}f) && (\mathbf{Y}\text{-dist-}\vee) \end{aligned}$$

Therefore both $\mathbf{YP}f$ and $\mathbf{PY}f$ satisfy the equation $Z = \mathbf{Y}f \vee \mathbf{Y}Z$. Applying the Unique Fixed-Point Theorem, this demonstrates, as we would expect, that

$$\mathbf{PY}f = \mathbf{YP}f \qquad (\mathbf{Y}\text{-dist-}\mathbf{P})$$

This result can be used, for example, for us to case-split and independently simulate the *first* step of an iteration, since $\mathbf{P}f = f \vee \mathbf{YP}g = f \vee \mathbf{PY}f$.

We can also use the uniqueness of fixed-points to derive the following similar results:

$$\begin{aligned} \mathbf{P}\mathtt{ff} &= \mathtt{ff} && (\mathbf{P}\text{-}\mathtt{ff}) \\ \mathbf{P}(f \vee g) &= \mathbf{P}f \vee \mathbf{P}g && (\mathbf{P}\text{-dist-}\vee) \\ f\,\mathbf{S}\,\mathtt{ff} &= \mathtt{ff} && (\,\mathbf{S}\text{ -}\mathtt{ff}\text{-2}) \\ \mathtt{tt}\,\mathbf{S}\,g &= \mathbf{P}g && (\,\mathbf{S}\text{ -}\mathtt{tt}\text{-1}) \\ \mathbf{Y}(f\,\mathbf{S}\,g) &= (\mathbf{Y}f)\,\mathbf{S}\,(\mathbf{Y}g) && (\mathbf{Y}\text{-dist-}\,\mathbf{S}\,) \\ f\,\mathbf{S}\,(g \vee h) &= (f\,\mathbf{S}\,g) \vee (f\,\mathbf{S}\,h) && (\,\mathbf{S}\text{ -dist-}\vee) \\ (f \wedge g)\,\mathbf{S}\,h &= (f\,\mathbf{S}\,h) \wedge (g\,\mathbf{S}\,h) && (\,\mathbf{S}\text{ -dist-}\wedge) \end{aligned}$$

4.1.4 Symbolic Constructs

The symbolic constructs of GTL are orthogonal to the other types of constructs, since they are the only constructs that affect the differences between each of the symbolic indices. As a result, the other logical operations distribute over symbolic ones. For example,

$$(Q \rightarrow f \mid g) \wedge h = Q \rightarrow (g \wedge h) \mid (f \wedge h) \qquad (\wedge\text{-dist-}\rightarrow)$$

$$(Q \rightarrow f \mid g) \vee h = Q \rightarrow (g \vee h) \mid (f \vee h) \qquad (\vee\text{-dist-}\rightarrow)$$

$$\mathbf{Y}(u \rightarrow f \mid g) = u \rightarrow \mathbf{Y}f \mid \mathbf{Y}g \qquad (\mathbf{Y}\text{-dist-}\rightarrow)$$

$$(f \wedge g)(u := Q) = (f(u := Q)) \wedge (g(u := Q)) \qquad (:=\text{-dist-}\wedge)$$

$$(f \vee g)(u := Q) = (f(u := Q)) \vee (g(u := Q)) \qquad (:=\text{-dist-}\vee)$$

$$\mathbf{Y}(f(u := Q)) = (\mathbf{Y}f)(u := Q) \qquad (\mathbf{Y}\text{-dist-}:=)$$

The conditional follows a few simple rules that we would expect it to:

$$(\text{true} \rightarrow f \mid g) = f \qquad (\rightarrow\text{-true})$$

$$(\text{false} \rightarrow f \mid g) = g \qquad (\rightarrow\text{-false})$$

$$(\neg Q \rightarrow f \mid g) = Q \rightarrow g \mid f \qquad (\rightarrow\text{-neg})$$

$$(Q \rightarrow f \mid f) = f \qquad (\rightarrow\text{-equal})$$

Rules for introducing and removing variables are useful because they allow control over whether properties are model checked explicitly or symbolically. This is important because some property aspects, such as values on datapaths, can be more efficiently represented symbolically. For example, we have the option of representing disjunction either explicitly or symbolically by introducing a fresh variable u:

$$f \vee g = (\exists u \,.\, u \rightarrow f \mid g) \qquad (\text{sym-disj})$$

This rule can be combined with the distributive laws for disjunction in order to affect the model checking process. For example, the rule allows us to change the two explicit post-image calculations represented by $\mathbf{Y}f \vee \mathbf{Y}g$ into the single symbolic post-image calculation given by $(\exists u \,.\, \mathbf{Y}(u \rightarrow f \mid g))$.

We can also go in the reverse direction and make model checking more explicit. The following rule states that symbolic substitution is equivalent to textual substitution on its operand.

$$f(u := Q) = f[Q/u] \qquad (\text{subst})$$

We can use this rule to flatten symbolic model checking into explicit model checking, as the following example demonstrates.

Example Suppose we would like to verify that a 4-step clock generator signals clk when reset with signal r, and every fourth time interval afterward. We can model check this behaviour explicitly with the following formula:

$$\mu\mathrm{Count}_0 \,.\, \mathsf{r} \vee (\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}_0))))$$

Model checking calculates the set of states where either a reset occurs, or has last occurred a multiple of four time-steps ago. Now suppose that we know that the timer is implemented using a two-bit counter, and we would like to use a more efficient symbolic model checking approach. The following property finds the symbolic set of states in which the value of the counter is $\mathbf{u}$:

$$(\mu\mathrm{Count} \,.\, ((\mathbf{u} = 0) \;\rightarrow\; \mathsf{r} \,|\, \mathsf{ff}) \vee (\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}(\mathbf{u} := \mathbf{u} - 1)))(\mathbf{u} := 0)$$

It is not immediately obvious that the two properties are equivalent. We can repeatedly use the substitution rule, however, to flatten-out the symbolic states and demonstrate equivalence:

$$\begin{aligned}
\mathrm{Count}[0/\mathbf{u}] =& (((\mathbf{u} = 0) \;\rightarrow\; \mathsf{r} \,|\, \mathsf{ff}) \vee (\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}(\mathbf{u} := \mathbf{u} - 1)))(\mathbf{u} := 0)\\
=& (((\mathbf{u} = 0) \;\rightarrow\; \mathsf{r} \,|\, \mathsf{ff}) \vee (\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}(\mathbf{u} := \mathbf{u} - 1)))[0/\mathbf{u}]\\
=& ((0 = 0) \;\rightarrow\; \mathsf{r} \,|\, \mathsf{ff}) \vee (\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}(\mathbf{u} := 0 - 1))\\
=& \mathsf{r} \vee (\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}[3/\mathbf{u}])\\
=& \mathsf{r} \vee (\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}[2/\mathbf{u}]))\\
=& \mathsf{r} \vee (\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}[1/\mathbf{u}])))\\
=& \mathsf{r} \vee (\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}(\neg\mathsf{r} \wedge \mathbf{Y}\mathrm{Count}[0/\mathbf{u}]))))
\end{aligned}$$

We can now use the unique fixed-point theorem to deduce that $\mathrm{Count}_0 = \mathrm{Count}[0/\mathbf{u}]$.

4.2 Decomposition Rules

We now consider rules that enable decomposition of a property into multiple model checking runs. Such rules can be employed in cases where the original property requires excess resources, but splitting it up allows each piece to be verified successfully.

4.2.1 Logical Decomposition

There are some straightforward rules for decomposing GTL properties by splitting antecedents or consequents. For example, if an antecedent can be case-split into conditions A_1 and A_2, then we can verify each of these conditions in turn:

$$\frac{\mathrm{A}_1 \Rightarrow \mathrm{C} \quad \mathrm{A}_2 \Rightarrow \mathrm{C}}{\mathrm{A}_1 \vee \mathrm{A}_2 \Rightarrow \mathrm{C}} \tag{$\vee$-split}$$

This rule follows directly from the semantics of GTL. Dually, if the consequent of a property requires us to verify two independent responses, then we can verify each of these separately:

$$\frac{\mathrm{A} \Rightarrow \mathrm{C}_1 \quad \mathrm{A} \Rightarrow \mathrm{C}_2}{\mathrm{A} \Rightarrow \mathrm{C}_1 \wedge \mathrm{C}_2} \tag{$\wedge$-split}$$

Recall that the GTL property $\mathrm{A} \Rightarrow \mathrm{C}$ is satisfied if and only if every model trace that satisfies A also satisfies C. Therefore, for any circuit model, the leads-to relation is reflexive and transitive:

$$\frac{}{A \Rightarrow A} \tag{$\Rightarrow$-refl}$$

$$\frac{A \Rightarrow B \quad B \Rightarrow C}{A \Rightarrow C} \tag{$\Rightarrow$-trans}$$

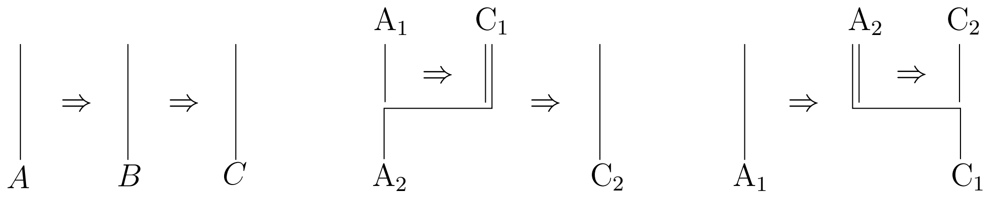
Transitivity allows us to split a simulation in half by introducing an intermediate state, B. The new verification approach first shows that simulating from A results in B. Secondly it verifies that simulating from B results in C. These two properties will typically simulate different adjacent segments of the circuit, with the formula B corresponding to a ‘cut-point’ between them. This is illustrated in Figure 4.1(i). In logical terms, it can also be seen as expressing the soundness of either weakening an antecedent, or strengthening a consequent.

The transitivity rule is linear in nature, but we can extend the approach to branching simulations, given that the GTL semantics is monotonic. We can express monotonicity (Theorem 3.3.2) with the rule

$$\frac{A \Rightarrow C}{f[A/X] \Rightarrow f[C/X]} \tag{mono}$$

Or equivalently as

$$\frac{\mathrm{A}_1 \Rightarrow \mathrm{C}_1}{f[\mathrm{A}_1/\mathrm{C}_1] \Rightarrow f} \qquad \frac{\mathrm{A}_1 \Rightarrow \mathrm{C}_1}{f \Rightarrow f[\mathrm{C}_1/\mathrm{A}_1]}$$



(i) Leads-to Transitivity (ii) Pre-composition (iii) Post-composition

Figure 4.1: Decomposing a Simulation

Using these rules combined with transitivity, we can show that it is sound to weaken, or strengthen, any *sub-formula* of an antecedent, or consequent, respectively:

$$\frac{\mathrm{A}_1 \Rightarrow \mathrm{C}_1 \quad \mathrm{A}_2 \Rightarrow \mathrm{C}_2}{\mathrm{A}_2[\mathrm{A}_1/\mathrm{C}_1] \Rightarrow \mathrm{C}_2} \qquad \text{(pre-composition)}$$

$$\frac{\mathrm{A}_1 \Rightarrow \mathrm{C}_1 \quad \mathrm{A}_2 \Rightarrow \mathrm{C}_2}{\mathrm{A}_1 \Rightarrow \mathrm{C}_1[\mathrm{C}_2/\mathrm{A}_2]} \qquad \text{(post-composition)}$$

Because these substitutions can take place at any place within a formula, these rules express that branching simulations can be split apart on any branch. The first rule is termed *pre-composition*, because it effectively adds a simulation before some part of the antecedent condition. Similarly, the *post-composition* effectively places an extra simulation after one part of the consequent of another. Figures 4.1 (ii) and (iii) illustrate the possible layout of the segments of a circuit corresponding to such decompositions.

Monotonicity and transitivity allow us to derive other rules for decomposition. For example, the following derived rule can be used to compose simulations conjunctively:

$$\frac{\dfrac{\mathrm{A}_1 \Rightarrow \mathrm{C}_1}{\mathrm{A}_1 \wedge \mathrm{A}_2 \Rightarrow \mathrm{C}_1 \wedge \mathrm{A}_2}\,\text{(mono)} \qquad \dfrac{\mathrm{A}_2 \Rightarrow \mathrm{C}_2}{\mathrm{C}_1 \wedge \mathrm{A}_2 \Rightarrow \mathrm{C}_1 \wedge \mathrm{C}_2}\,\text{(mono)}}{\mathrm{A}_1 \wedge \mathrm{A}_2 \Rightarrow \mathrm{C}_1 \wedge \mathrm{C}_2}\,(\Rightarrow\text{-trans})$$

This can be useful if two parallel independent circuit segments can be simulated independently. Since monotonicity holds for every formula, we can equally well derive this result for any function of two recursion variables. For example, we can show that $\mathrm{A}_1 \Rightarrow \mathrm{C}_1$ and $\mathrm{A}_2 \Rightarrow \mathrm{C}_2$ implies

$$(\mathbf{Y}\mathrm{A}_1)\,\mathbf{S}\,\mathrm{A}_2 \;\Rightarrow\; (\mathbf{Y}\mathrm{C}_1)\,\mathbf{S}\,\mathrm{C}_2$$

Properties can also be split based on the value of a symbolic variable. For example, if we can verify a property for both valuations of a particular variable, then the entire property must hold:

$$\frac{(\mathrm{A} \Rightarrow \mathrm{C})[\mathrm{true}/u] \quad (\mathrm{A} \Rightarrow \mathrm{C})[\mathrm{false}/u]}{\mathrm{A} \Rightarrow \mathrm{C}}$$

4.2.2 Temporal Decomposition

In this section, we examine decomposition relating to temporal aspects of a property.

4.2.2.1 Temporal Shifting

Since GTL properties have no notion of a starting state, we can temporally shift properties without affecting their meaning, i.e. for a given model:

$$(A \Rightarrow C) \quad \text{iff} \quad (\mathbf{Y}A \Rightarrow \mathbf{Y}C) \tag{$\mathbf{Y}$-shift}$$

Proof. By monotonicity of $\mathbf{Y}Z$, it is clear that $A \Rightarrow C$ implies $\mathbf{Y}A \Rightarrow \mathbf{Y}C$.

Now suppose $\mathcal{K} \models \mathbf{Y}A \Rightarrow \mathbf{Y}C$ and pick some trace $\sigma \in \mathrm{tr}(\mathcal{K})$ that satisfies A. Since every Kripke structure has a total transition relation, there is some trace $\sigma.s \in \mathrm{tr}(\mathcal{K})$, where s is some state of the system. By the semantics of Yesterday, $\sigma.s$ satisfies $\mathbf{Y}A$, and so, by our assumption, also satisfies $\mathbf{Y}C$. Now by the semantics of Yesterday, it must be that σ satisfies C. Hence $\mathcal{K} \models A \Rightarrow C$. □

Section 3.7.3 introduced a model checking algorithm for GTL properties with atemporal consequent (containing no $\mathbf{Y}$). Using the temporal shifting rule allows us extend this algorithm to those properties that may contain $\mathbf{Y}$ in their consequent. Properties that are bounded and linear, like those of STE, can be re-written into a form where both the antecedent and consequent are a conjunction of formulas that refer to distinct time-slices, such as:

$$\mathbf{Y}f \wedge g \Rightarrow \mathbf{Y}h \wedge i$$

The consequent conjuncts can then be split up. For this example there is one to check the final time-step, i, and one which checks the preceding condition, h:

$$\dfrac{\mathbf{Y}f \wedge g \Rightarrow i \qquad \dfrac{\dfrac{}{\mathbf{Y}f \wedge g \Rightarrow \mathbf{Y}f} \qquad \dfrac{f \Rightarrow h}{\mathbf{Y}f \Rightarrow \mathbf{Y}h}\,\mathbf{Y}\text{-shift}}{\mathbf{Y}f \wedge g \Rightarrow \mathbf{Y}h}\,\Rightarrow\text{-trans}}{\mathbf{Y}f \wedge g \Rightarrow \mathbf{Y}h \wedge i}\,\wedge\text{-split}$$

Example Suppose a 16-bit adder should deliver the first 8-bits of its output in one time-step, and the remainder in the subsequent time step. This can be captured by the property:

$$\begin{gathered}\mathbf{Y}(\mathbf{Y}(\mathsf{opA}\ \text{is}\ a \wedge \mathsf{opB}\ \text{is}\ b)) \\ \Rightarrow \\ \mathbf{Y}(\mathsf{out1}\ \text{is}\ (a+b)[7:0]) \wedge \mathsf{out2}\ \text{is}\ (a+b)[15:8]\end{gathered}$$

We can verify this property by showing independently that both outputs are correct, using the two properties

$$\begin{aligned} \mathbf{Y}(\mathsf{opA} \text{ is } a \wedge \mathsf{opB} \text{ is } b) &\Rightarrow \mathsf{out1} \text{ is } (a+b)[7:0] \\ \mathbf{Y}(\mathbf{Y}(\mathsf{opA} \text{ is } a \wedge \mathsf{opB} \text{ is } b)) &\Rightarrow \mathsf{out2} \text{ is } (a+b)[15:8] \end{aligned}$$

In practical implementations, the model checking process for such simulations can be shared, since the simulation of the second property will include that of the first.

4.2.2.2 Induction

Since GTL fixed-points represent finite iteration, we can use an induction rule to decompose them. For formulas f where every free instance of Z is bound by $\mathbf{Y}$, such a rule is captured by:

$$\frac{f[\mathrm{ff}/X] \Rightarrow C \quad f[C/Z] \Rightarrow C}{\mu Z\,.\,f \Rightarrow C} \qquad (\mu\text{-induct})$$

In effect, this rule states that fixed-points satisfy their inductive invariants.

Proof. We show by induction that the assumptions are sufficient to conclude that each approximant ‘leads to’ C.

Case $n = 0$: $\mu^0 Z\,.\,f = \mathrm{ff}$ and $\mathrm{ff} \Rightarrow C$ is vacuously true.

Case $n = 1$: $\mu^1 Z\,.\,f = f[\mathrm{ff}/X]$, which is covered by the first assumption.

Case $n+1$ given case n: Suppose $\mu^n Z\,.\,f \Rightarrow C$. Then since $f[C/Z] \Rightarrow C$, by monotonicity we have that $f[C/Z][\mu^n Z\,.\,f/C] \Rightarrow C$. But $f[C/Z][\mu^n Z\,.\,f/C] = f[\mu^n Z\,.\,f/Z] = \mu^{n+1} Z\,.\,f$. Hence $\mu^{n+1} Z\,.\,f \Rightarrow C$.

Since the traces that satisfy each approximant also satisfy C, and the fixed-point is the union of these approximants, it must be that traces satisfying the fixed-point also satisfy C, as required. □

This induction principle allows us to derive some powerful high-level temporal patterns. For example, we can verify that some invariant I holds perpetually after reset R, by first showing that the reset establishes the invariant, and then that the invariant inductively holds:

$$\frac{R \Rightarrow I \quad \mathbf{Y}I \Rightarrow I}{\mathbf{P}R \Rightarrow I}$$

Proof.

$$\dfrac{\dfrac{\dfrac{R \Rightarrow I}{R \vee \mathbf{Y}\mathrm{ff} \Rightarrow I} \qquad \dfrac{\mathbf{Y}I \Rightarrow I \quad R \Rightarrow I}{R \vee \mathbf{Y}I \Rightarrow I}\ \vee\text{-split}}{\mu Z\,.\,(R \vee \mathbf{Y}Z) \Rightarrow I}\ \mu\text{-induct}}{\mathbf{P}R \Rightarrow I}\ \mathbf{P}\text{-def}$$

□

Example To illustrate the use of one of these rules, we will examine the decomposition of the verification of an industrial memory design previously described in [HCY03b]. The design consists of two blocks: a memory block for storing incoming data, and a processing block that performs selection, alignment and masking on the data being read.

Verification aims to show that if data D has been written to address A, and not overwritten since, then if address A is accessed with appropriate options, the correct selection, alignment and mask of D is returned. The selection and alignment options must be provided with the read request, and the mask options provided one cycle later, when the read completes. Introducing extra names to describe the simple input predicates required, the property can be specified with GTL as:

$$\begin{aligned}\mathbf{Y}(\mathbf{Y}(\text{no\_overwrite}\,\mathbf{S}\,\text{write}) \wedge \text{read} \wedge \text{sel\_align})\\ \wedge\,\text{mask} \Rightarrow \text{data\_correct}\end{aligned} \tag{4.4}$$

The decomposition approximately halves verification time [HCY03b] by introducing an internal predicate, read_result, to assert that the data is correctly transmitted on the bus between memory and processing blocks. The first stage of verification checks that the memory correctly stores the data and sends it on this bus:

$$\mathbf{Y}(\text{no\_overwrite}\,\mathbf{S}\,\text{write}) \wedge \text{read} \Rightarrow \text{read\_result} \tag{4.5}$$

The second stage verifies that if the processing block correctly receives the data then it is processed correctly:

$$\mathbf{Y}(\text{read\_result} \wedge \text{sel\_align}) \wedge \text{mask} \Rightarrow \text{data\_correct} \tag{4.6}$$

To justify such decomposition it is necessary to show that Equation 4.4 is implied by Equation 4.5 and Equation 4.6 together. This condition is exactly captured by the branching pre-composition rule, since substituting the antecedent of Equation 4.5 in for read_result in the antecedent of Equation 4.6 results in the original property of Equation 4.4.

4.3 Abstraction Refinement Rules

As well as expressing *what* is being checked, GTL specifications also express the model checking approach that will be used to check it. This means that the shape of a formula can be used to control the degree of precision employed. Abstraction refinement can therefore be described using property-preserving rewrite rules that change the model checking direction. Since every atomic step of simulation can be described precisely using GTL, we have complete control over model checking. We will formalize abstraction refinement as follows:

Definition 4.3.1. *The transformation of GTL formula f to GTL formula g is an* abstraction refinement, *written $f \succsim g$, if f and g are semantically equivalent, and the abstract simulation of g is more precise than the abstract simulation of f, $\lfloor f \rfloor_\sigma \sqsupseteq^{\forall\mathcal{V}} \lfloor g \rfloor_\sigma$, for every circuit model.*

Any abstraction refinement rule can be soundly applied to any sub-formula in order to create another abstraction refinement rule:

$$f \succsim g \quad \text{implies} \quad h[f/Z] \succsim h[g/Z]$$

This is a result of the monotonicity of abstract simulation, given by Lemma 3.7.10. In practical terms, this allows us to refine the abstraction of any *intermediate* simulation state.

There are two ways in which a ternary simulation can over-approximate the image of an antecedent. First, set-based simulation itself is not complete for certain antecedents. Second, the ternary representation introduces information loss due to its approximation of disjunction and post-image calculation. We will consider rules for each of these effects in turn.

4.3.1 Refining Disjunction

Suppose f does not contain any fixed-points, and model checking $f(g \vee h) \Rightarrow \mathrm{C}$ fails due to over-abstraction. If the loss of required information is caused by this disjunction alone, then it must be that both $f(g) \Rightarrow \mathrm{C}$ and $f(h) \Rightarrow \mathrm{C}$ would succeed individually.

One way of avoiding such information loss is to *repeat* the simulation f for both disjuncts independently. By doing this, we effectively make model checking more explicit. This refinement is captured by distributing f, where possible, over disjunction:

$$f(g \vee h) \quad \succsim \quad f(g) \vee f(h) \qquad (\vee\text{-dist})$$

Proof. By definition of $\sqcup$,

$$\begin{aligned}\lfloor g \rfloor^{\nu}_{\sigma} \sqcup \lfloor h \rfloor^{\nu}_{\sigma} &\sqsupseteq \lfloor g \rfloor^{\nu}_{\sigma} \\ \lfloor g \rfloor^{\nu}_{\sigma} \sqcup \lfloor h \rfloor^{\nu}_{\sigma} &\sqsupseteq \lfloor h \rfloor^{\nu}_{\sigma}\end{aligned}$$

so by monotonicity of abstract simulation of $f(Z)$,

$$\begin{aligned}\lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto \lfloor g \rfloor^{\nu}_{\sigma} \sqcup \lfloor h \rfloor^{\nu}_{\sigma}]} &\sqsupseteq \lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto \lfloor g \rfloor^{\nu}_{\sigma}]} \\ \lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto \lfloor g \rfloor^{\nu}_{\sigma} \sqcup \lfloor h \rfloor^{\nu}_{\sigma}]} &\sqsupseteq \lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto \lfloor h \rfloor^{\nu}_{\sigma}]}\end{aligned}$$

Hence

$$\lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto \lfloor g \rfloor^{\nu}_{\sigma} \sqcup \lfloor h \rfloor^{\nu}_{\sigma}]} \sqsupseteq \lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto \lfloor h \rfloor^{\nu}_{\sigma}]} \sqcup \lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto \lfloor g \rfloor^{\nu}_{\sigma}]}$$

or, equivalently

$$f(g \vee h) \succsim f(g) \vee f(h) \qquad \square$$

Intuitively, this rule is an abstraction refinement because it delays the stage at which information is lost until later in the simulation. This allows more to be deduced from this information, increasing the precision with which model checking runs. Unfortunately, this increases the number of simulation steps that occur during model checking, giving a performance penalty.

Symbolic representation can allow us to reduce this penalty by *sharing* the common elements between two repeated steps. We will term this technique *symbolic disjunctive completion* due to the correspondence with disjunctive completion in abstract interpretation theory [CC79]. Symbolic disjunctive completion uses $u \;\rightarrow\; g \,|\, h$ to represent $g \vee h$, where u is existentially quantified at the top level of simulation. The symbolic states capture circuit node dependencies that would otherwise have been lost by the abstract disjunction. This induces the following form of abstraction refinement:

$$f(g \vee h) \succsim \exists u \,.\, f(u \;\rightarrow\; g \,|\, h) \qquad (\vee\text{-sym-dist})$$

In abstract interpretation theory, using the set of downwards closed abstract elements to express disjunction is known as *disjunctive completion* [GRS00]. When computing over *sets* of representatives of *sets* of states, union is a complete abstract interpretation of disjunction. The symbolic representation can be viewed as indexing such sets. Under this interpretation, the symbolic indexing encodes the disjunctive completion of ternary propagations.

Example Consider the XOR-gate with delayed inputs shown in Figure 4.2. We would like to verify that if the inputs were mutually exclusive in the preceding time step, then the output should now be high. The most obvious option to simulate this, $\mathbf{Y}((\mathsf{a} \wedge \neg \mathsf{b}) \vee (\neg \mathsf{a} \wedge \mathsf{b}))$, loses all information in the first time-step, as shown in Figure 4.2(i). This is because we are specifying a dependency between circuit nodes that ternary vectors cannot capture. Applying the ($\vee$-dist) rule distributes the disjunction so that the information about the mutual exclusion case is retained in the post-image, as shown in Figure 4.2(ii). If we apply the rule for symbolic disjunctive completion, ($\vee$-sym-dist), then we achieve the same effect with a single simulation (Figure 4.2(iii)). The variable u captures the required dependency, and the output can be shown to be high.

4.3.2 Product Reduction

It is often possible to simplify a model checking run by forcing separate parts of the circuit to be exercised independently. This is the motivation behind compositional GSTE, described in Section 2.6. The result of such a transformation is that, rather than simulating an entire system $Q_1 \times Q_2 \times \ldots \times Q_n$, simulation takes place independently within each component $Q_1, Q_2, \ldots, Q_n$ in turn. For this reason, we will term this type of transformation *product reduction*.

In terms of GTL, the degree of product reduction is determined by the point in the simulation where the result of two sub-simulations are conjoined. Since GTL model checking keeps track of only the final states, or images, of each formula, a product reduction effectively allows for *more* traces than required, since they are not limited by the constraining interaction between the system components. Product reduction can be expressed as the abstraction refinement rule:

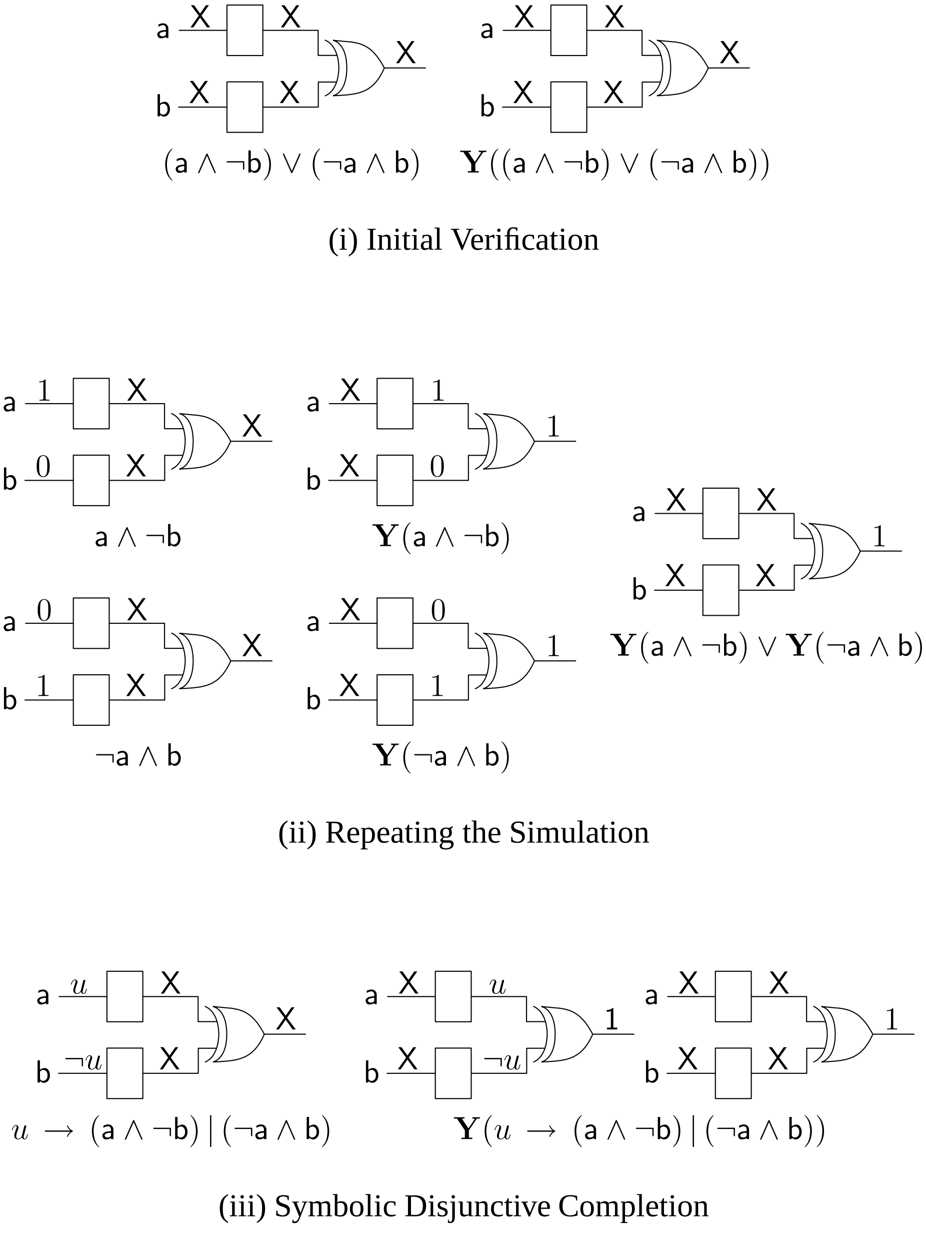
$$\mathbf{Y}f \wedge \mathbf{Y}g \succsim \mathbf{Y}(f \wedge g) \qquad \text{(prod. red.)}$$

Proof. By definition of $\sqcap$,

$$\begin{aligned} \lfloor f \rfloor_\sigma^\nu \sqcap \lfloor g \rfloor_\sigma^\nu &\sqsubseteq \lfloor f \rfloor_\sigma^\nu \\ \lfloor f \rfloor_\sigma^\nu \sqcap \lfloor g \rfloor_\sigma^\nu &\sqsubseteq \lfloor g \rfloor_\sigma^\nu \end{aligned}$$

so by monotonicity of $\mathrm{post}^\sharp$,

$$\begin{aligned} \mathrm{post}^\sharp(\lfloor f \rfloor_\sigma^\nu \sqcap \lfloor g \rfloor_\sigma^\nu) &\sqsubseteq \mathrm{post}^\sharp(\lfloor f \rfloor_\sigma^\nu) \\ \mathrm{post}^\sharp(\lfloor f \rfloor_\sigma^\nu \sqcap \lfloor g \rfloor_\sigma^\nu) &\sqsubseteq \mathrm{post}^\sharp(\lfloor g \rfloor_\sigma^\nu) \end{aligned}$$


a X X
b X X
X
$(a \wedge \neg b) \vee (\neg a \wedge b)$
a X X
b X X
X
$\mathbf{Y}((a \wedge \neg b) \vee (\neg a \wedge b))$
a 1 X
b 0 X
X
$a \wedge \neg b$
a X 1
b X 0
1
$\mathbf{Y}(a \wedge \neg b)$
a X X
b X X
1
$\mathbf{Y}(a \wedge \neg b) \vee \mathbf{Y}(\neg a \wedge b)$
a 0 X
b 1 X
X
$\neg a \wedge b$
a X 0
b X 1
1
$\mathbf{Y}(\neg a \wedge b)$
a u X
b $\neg u$ X
X
$u \rightarrow (a \wedge \neg b) \mid (\neg a \wedge b)$
a X u
b X $\neg u$
1
a X X
b X X
1
$\mathbf{Y}(u \rightarrow (a \wedge \neg b) \mid (\neg a \wedge b))$

(i) Initial Verification

(ii) Repeating the Simulation

(iii) Symbolic Disjunctive Completion

Figure 4.2: Two Methods of Refining Abstract Disjunction

Hence

$$\mathrm{post}^\sharp(\lfloor f \rfloor_\sigma^\nu \sqcap \lfloor g \rfloor_\sigma^\nu) \;\sqsubseteq\; \mathrm{post}^\sharp(\lfloor f \rfloor_\sigma^\nu) \sqcap \mathrm{post}^\sharp(\lfloor g \rfloor_\sigma^\nu)$$

which exactly corresponds to

$$\mathbf{Y}(f \wedge g) \precsim \mathbf{Y}f \wedge \mathbf{Y}g \qquad \square$$

Example Consider verifying a three-stage pipeline that independently decrements its two binary inputs, a and b, then adds them together. We can try to simulate the two inputs independently, and combine the results at the final stage using: $(\mathbf{YY}(\mathsf{a}\ \mathbf{is}\ u)) \wedge (\mathbf{YY}(\mathsf{b}\ \mathbf{is}\ v))$. Suppose this run fails due to over-abstraction and we discover that, in fact, the two inputs start to interact at the second stage. We refine the simulation to $\mathbf{Y}(\mathbf{Y}(\mathsf{a}\ \mathbf{is}\ u) \wedge \mathbf{Y}(\mathsf{b}\ \mathbf{is}\ v))$ (or even $\mathbf{YY}(\mathsf{a}\ \mathbf{is}\ u \wedge \mathsf{b}\ \mathbf{is}\ v)$). This simulation may have greater space requirements, but may now maintain enough information for verification to succeed.

4.3.2.1 Partial Order Reduction

When simulation with fixed-points is involved, product reduction can correspond to partial order reduction. Partial order reduction helps improve the performance of model checking by eliminating the interleaving of independent actions.

For example, suppose that we wish to simulate the condition under which event f and g have both occurred in the past. The most obvious way to simulate this condition is to use the formula $\mathbf{P}f \wedge \mathbf{P}g$. When this is given to the model checker, it simulates the results of f and g independently. This is perfectly acceptable if f and g simulate different parts of the circuit, but if there is any dependency between the effects of the two formulas, then they will not be captured.

An alternative way to simulate the condition is to case-split on which event occurred first. This is captured by the simulation

$$\mathbf{P}(f \wedge \mathbf{P}g) \vee \mathbf{P}(\mathbf{P}f \wedge g)$$

that is capable of capturing the interaction between the two events.

4.3.3 Refinement by Case-Splitting

Due to the nature of abstract simulation, there are several other ways in which information can be lost, leading to over-abstraction. First, propagation is calculated

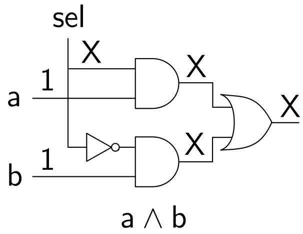


Figure 4.3: Multiplexer

locally, on a node-by-node basis. Because of this, the algorithm does not take account of global patterns that affect the dependencies between nodes. For example, consider the multiplexer in Figure 4.3. The output should be the same as a when sel is 1, and the same as b when sel is 0. It follows that if both a and b are high then the output should also be high. The propagation shown in this figure does not derive this global outcome, however, as it simulates each gate locally.

Second, propagation only takes place in a forward direction, although backwards propagation is sometimes required to attain new constraints. An example of this has already been illustrated in Figure 3.5.

In both such cases, the required precision can be retained by suitably case-splitting the simulation. This is captured by the abstraction refinement rule:

$$f \quad \gtrsim \quad (f \wedge \mathsf{n}) \vee (f \wedge \neg \mathsf{n}) \qquad \text{(case-split)}$$

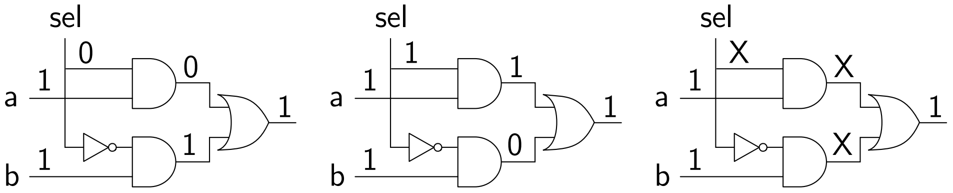
It is easy to show that this is a refinement rule, since f is equivalent to $f \wedge (\mathsf{n} \vee \neg \mathsf{n})$, and we can then apply the rule for refinement via distribution of disjunction. Typically, n is some circuit node whose simulation directly affects the segment of interest.

Example Figure 4.4 shows the result of applying the rule case-split on a multiplexer by splitting on the sel input node. The two scalar inputs of 1 and 0 are each considered in turn (Figures 4.4 (i) and (ii)) and the simulator maintains that the output is always high when the two cases are merged in (iii).

As with disjunctive completion, we also have the option of using a variable to index the two cases symbolically:

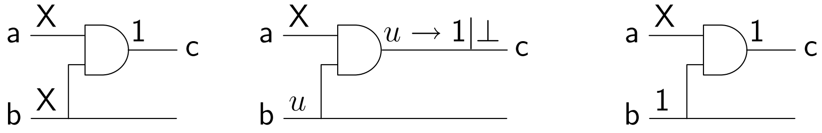
$$f \quad \gtrsim \quad \exists u \,.\, (f \wedge \mathsf{n} \ \textbf{is} \ u) \qquad \text{(sym-case-split)}$$

This powerful transformation effectively allows us to pepper the simulation with extra detail, by introducing pieces of complete symbolic simulation within a predominantly ternary run.



(i) $\mathsf{a} \wedge \mathsf{b} \wedge \mathsf{sel}$ (ii) $\mathsf{a} \wedge \mathsf{b} \wedge \neg\mathsf{sel}$ (iii) $(\mathsf{a} \wedge \mathsf{b} \wedge \mathsf{sel}) \vee (\mathsf{a} \wedge \mathsf{b} \wedge \neg\mathsf{sel})$

Figure 4.4: Case-Split Multiplexer Verification



(i) c (ii) $\mathsf{c} \wedge (\mathsf{b}\ \mathbf{is}\ u)$ (iii) $\exists u\ .\ \mathsf{c} \wedge (\mathsf{b}\ \mathbf{is}\ u)$

Figure 4.5: Case-Splitting to Avoid Backwards Simulation

Example Figure 4.5 (i) shows an example where information is lost because constraints are not propagated backwards. Since the AND-gate output c is 1, it must be that both its inputs are also high. By introducing the variable u on node b in Figure 4.5 (ii), this forces an over-constraint in the case that u is false. When u is quantified out in Figure 4.5 (iii), we maintain the constraint that node b must be high.

4.3.4 Emulating Precise Nodes

As described in Section 2.5.4.5, the GSTE simulator for assertion graphs allows a set of *precise nodes* to be specified. Ghost variables are created to mirror the value of these nodes, thus maintaining their dependencies precisely. This allows us to select aspects of the circuit that we believe require a more concrete representation. In this section, we describe a GTL transformation that achieves the same effect.

The transformation works by using a set of ghost variables, p_i to encode the values on precise nodes n_i. Similar to symbolic model checking [McM92], the simulator keeps a Boolean predicate, containing these ghost variables, that precisely characterizes the set of possible node states associated with each simulation state. This is encoded using the *over-constraint* predicate (see 2.4.1). The connection between the values on the nodes and the constraints on the ghost variables is maintained by asserting that the two match in each simulation step.

In symbolic model checking, the variables used to encode state are effectively scoped locally to each time-step. Typically the transition relation, T, is a predicate over the current-state variables, p_i, and the next-state variables, p'_i. The image of a state predicate S is then calculated as $(\exists_i\, p_i \,.\, T \wedge S)[p_i/p'_i]$. This use of substitution and quantification eliminates the need to use one set of variables for each time-step. Our precise nodes transformation uses substitution and quantification in a similar way, so that variables can be used by each property state independently.

Let us use the example from Section 2.5.4.5, of a circuit with two 2-bit counters, c_1 and c_2, that increment in synchrony. When a reset occurs, by signaling node r, both counters are reset to zero. Suppose that both counters are attached to a comparator, and we would like to verify that this comparator always returns true.

If we simulate the circuit with the formula $\mu Z \,.\, \mathsf{r} \vee \neg\mathsf{r} \wedge \mathbf{Y} Z$ then the values of both counters are quickly abstracted to XX. In the first iteration, $(a_1a_0, b_1b_0) = (00, 00)$. In the second iteration $(a_1a_0, b_1b_0) = (00, 00) \sqcup (01, 01) = (0\mathsf{X}, 0\mathsf{X})$. In the third iteration, $(a_1a_0, b_1b_0) = (0\mathsf{X}, 0\mathsf{X}) \sqcup (\mathsf{XX}, \mathsf{XX}) = (\mathsf{XX}, \mathsf{XX})$. The final output of the comparator is X and so verification fails.

Now we will aim to use variables to index these different cases. The situation is different from symbolic model checking because we will use current-state variables, p_i, and *previous*-state variables, p_i^- . Given any simulation state, we can add variables to the state to encode the precise nodes, n_i, with the formula

$$D \quad = \quad (\bigwedge_i n_i \textbf{ is } p_i)$$

In our example, the formula r resets the counters to zero. Therefore $D \wedge \mathsf{r}$ sets the counters to zero, and asserts that the ghost variables match these zero counts. Now suppose that state Z is already indexed in terms of the counters' ghost variables for the current state. Then we can capture the relationship between the states by substituting the current state variables for previous state variables during the post-image calculation, then asserting the new current state variables and quantifying out the previous state variables:

$$\mu Z \,.\, \exists_i\, p_i^- \,.\, ((D \wedge \mathsf{r}) \vee (D \wedge \neg\mathsf{r} \wedge \mathbf{Y}(Z(p_i := p_i^-)_i)))$$

Let us step though this simulation for the dual-counter example with set of precise nodes $\{a_1, a_0, b_1, b_0\}$. In the first iteration, simulating r drives (a_1a_0, b_1b_0) to $(00, 00)$, and then D asserts that $(p_{a_1}p_{a_0}, p_{b_1}p_{b_0}) = (00, 00)$. Therefore the ternary vector associated with each symbolic valuation is $\perp$ (over-constrained) except for the state where

$$(p_{a_1}p_{a_0}, p_{b_1}p_{b_0}) = (00, 00)$$

for which it is $(00, 00)$.

We then substitute and take the post-image, giving us a simulation state where every symbolic valuation is $\bot$ except for

$$(p^-_{a_1} p^-_{a_0}, p^-_{b_1} p^-_{b_0}) = (00, 00)$$

which is assigned ternary vector $(01, 01)$. Now D again asserts that the current-state variables encode the post-image,

$$(p_{a_1} p_{a_0}, p_{b_1} p_{b_0}) = (01, 01)$$

Following quantification of the previous-state variables, every symbolic valuation is assigned $\bot$ except for when

$$(p_{a_1} p_{a_0}, p_{b_1} p_{b_0}) = (00, 00) \quad \vee \quad (p_{a_1} p_{a_0}, p_{b_1} p_{b_0}) = (01, 01)$$

which are assigned $(00, 00)$ and $(01, 01)$ respectively. This process continues until a fixed-point is reached where the only symbolic values that are not $\bot$ satisfy $(p_{a_1} = p_{b_1}) \wedge (p_{a_0} = p_{b_0})$ and have assigned ternary vector $(p_{a_1} p_{a_0}, p_{b_1} p_{b_0})$. In each of these consistent states, the output of the comparator is true. Therefore the verification succeeds.

This method generalizes to arbitrary GTL formulas, although the matter becomes more complicated when the formula is made up from more than one recursion variable. One way around this is to create a family of ghost variables, so that there is one for each of the occurrences of the variable being simulated.

4.4 Related Work

Although limited reasoning techniques currently exist for GSTE, the development of reasoning rules and associated theorem proving infrastructure has in the past played an important role within the more restricted setting of STE verification.

4.4.1 STE Reasoning

Seger and Joyce [SJ92] first formally reason about STE runs by embedding their semantics in Higher-Order Logic (HOL), and using HOL's associated theorem proving environment [GM93] to manipulate properties. They describe STE assertions as a set of tuples, each tuple of which encodes the timing and value information about a particular event in the circuit. The connection to STE simulation is then performed using a decision procedure tool called Voss. Manual reasoning is then used to connect STE assertion tuples to custom higher-level specification definitions in HOL.

Seger and Hazelhurst [Haz96] structure and simplify this approach, by describing runs in terms of the simple domain specific logic, Trajectory Evaluation Logic (TEL). They develop a simple set of rules for this logic that can be applied directly within their lightweight proof tool, VossProver. This makes reasoning more accessible, since it no longer requires either low-level reasoning at the level of assertion tuples, or an understanding of reasoning with HOL. The rule-base [SC95] contains rules for equivalence and decomposition, and have shown to be useful in industrial verification [AS95]. A range of extra reasoning rules for TEL have also been presented to complement particular advances in STE verification techniques. For instance, Aagaard et al. [AJS98] describe extra rules for temporal induction and case-splitting. Kaivola and Aagaard describe a range of rules used in an industrial divider circuit in [KA00]. TEL is also used to express symbolic indexing transformations [AJS99, MJ02], which can dramatically reduce memory verifications [PRBA97], as well as play a part in automatic abstraction refinement [RC06b, ABMS07, TG06, CHXY07].

Since TEL is close to being a sub-language of GTL, these rules are similar to some of our own. For instance, some are closely related to our rules for transitivity, temporal shifts, weakening and strengthening, and some aspects of symbolic reasoning. One significant difference is that TEL is defined in terms of a partial model structure, and so does not allow a law of the excluded middle. When using TEL in a binary setting, its partially ordered state-space models must be connected with additional Boolean relational models [AMO99] in order to derive such rules. In contrast, the semantic approach of GTL models the circuit as Boolean from the start, thus simplifying both our specification notation semantics as well as our theory. Of course, the main advantage of our reasoning system is that the rules are generalized to an unbounded setting.

4.4.2 GSTE Reasoning

In contrast to STE, reasoning techniques for GSTE are fewer in number, and are less formally specified. We suspect that this is primarily due to the difficulties that accompany reasoning with assertion graphs. Systematic transformations for abstraction control in GSTE were first described by Yang and Seger [YS02], where it is noted that case-splitting of assertion graph edges, and unrolling of edge loops, are often sufficient for abstraction refinement. In GTL, equivalent transformations to these are captured cleanly by a combination of case-splits, fixed-point unrolling and distributive operations. Our abstraction rules are, to our knowledge, the first general-purpose formalization of abstraction refinement applicable to GSTE simulation approaches.

Jain [Jai97] describes some similar forms of transformation and composition for his graphical simulation specification notations—the fore-runners of assertion graphs. Like assertion graphs, reasoning with and manipulating these forms requires lengthly proofs and algorithms. Hu et al. [HCY03b] are the first to formally reason with the currently accepted specification notation of assertion graphs, by providing decision procedures for both implication and verification under assumption. The methods rely on constructing monitor circuits [HCY03a] from assertion graphs, which signal when an assertion on their inputs fails. GSTE itself is then used on such monitor circuits to verify the required characteristics. An alternative approach to checking implication is provided by in [YYSX06], via an algorithm for explicit calculation of maximal models for assertion graphs. Although both these approaches are sound, they are long-winded when compared to some of the options available with GTL. For instance, at the end of Section 4.2, we demonstrate a GTL decomposition that is shown to be sound via a simple syntactic proof rule in our logic. The same decomposition verified using monitor circuits [HCY03b] requires an entirely separate GSTE verification effort.

As well as approaches that use decision procedures to reason formally about assertion graphs, some progress has also been made with exhaustive manual reasoning. A series of results regarding implication between assertion graphs is provided in [YYHS05]. These rules and proofs are long and involved, because they deal directly with flat mathematical representations of assertion graphs. Furthermore, the rules have a very limited range of applicability, with comparisons typically requiring equal assertion graph structures. When translated into GTL specifications, many of these rules become trivial instances of monotonicity.

Our product reduction abstraction refinement rule characterizes the motivation behind compositional GSTE [YS04]. This rule amounts to a universal abstraction of the dependencies between the parallel processes of which the circuit is composed. Similar abstractions between parallel processes have also been explored in [KDG95].

Chapter 5

Assertion Programs

This chapter introduces the language of *assertion programs*, which can be used to describe GSTE specifications at a higher level of abstraction than generalized trajectory logic. This enables more succinct specifications that can connect and make sense of the often numerous and complex simulation patterns required to fully verify a non-trivial hardware component.

There are several reasons why a language such as this is required above-and-beyond GTL. Most importantly, GTL formulas operate at the bit-level, so operations like arithmetic are generally obscured beyond the point of easy recognition. Assertion programs get around this problem by providing standard data types to capture these patterns directly. Another problem with GTL is the low limit on nested μ-expressions before a simulation becomes unintelligible to the human eye. In contrast, recursion in assertion programs is captured in a more scalable style, based on familiar programming notations. Furthermore, GTL specifications must sometimes be written in unintuitive forms to enforce a particular model checking strategy. In contrast, assertion programs are independent of model checking, so can be written in the best way to represent the properties most clearly.

Traditionally, GSTE specifications are complete forms of component specification. A single GTL property, however, expresses a single cause-of-effect relation, so a multiplicity of properties are typically required to verify the complete functionality of a hardware component. For example, several distinct GSTE simulations may be used for each different output signal of a particular circuit. Although these simulations are distinct, they often share some of their structure, since they are intended to drive the same device. It therefore makes sense to collate these common factors into a unified component specification. For this reason, we choose to use assertion programs to describe high-level model specifications that capture the required circuit

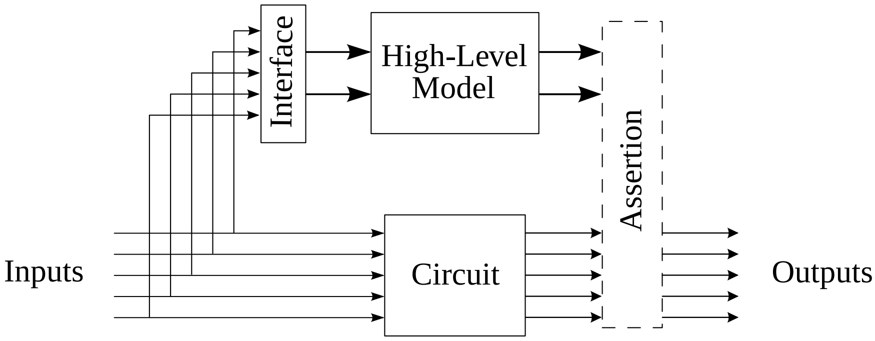


Figure 5.1: Conceptual Framework

behaviour. GSTE simulations for different aspects of the specification can then be generated directly from these specifications.

5.1 Specification Approach

In the conceptual view of our framework, the high-level model and the circuit design receive the same input data from the environment at each time-step. Both execute independently with these inputs, and produce their own responses. A set of assertions are then used to verify that the outputs of the circuit match the expectations from the specification, meaning that the circuit is considered a refinement of the high-level model. This is illustrated in Figure 5.1.

One apparent disadvantage of our approach is that the high-level models will be susceptible to the same human error as the circuit designs themselves. But verification does plug a significant abstraction gap, since factors other than pure functionality drive practical circuit designs to higher levels of complexity. For instance, nondeterminism is often introduced by design optimizations such as resource sharing, power reduction features and pipelines with additional control state. Symmetries in the functional specification are often not mirrored in the design, for reasons of layout placement or even electrical interference with neighbouring components.

5.2 Language Overview

Assertion programs are cycle-accurate executable representations of circuit specifications, so the language of assertion programs subscribes to the same temporal characteristics as the hardware itself. Time is modeled as a series of discrete steps, each of which contains some amount of synchronous parallel computation. So that we can describe any timing characteristics that might be possible in hardware, we follow its reactive nature. Therefore the current output values and the subsequent circuit state in our high-level model are determined by the current state and current input values.

5.2.1 Structure

Each program is split into four different blocks:

- The *variable declaration block* defines the types and variables used by the program.
- The *model block* describes the transition system of the specification in terms of statements that assign values to variables.
- The *interface block* describes how the inputs to the circuit relate to those of the high-level model.
- Finally, the *assertion block* describes properties to verify, regarding how the circuit output relates to that of the high-level model.

5.2.2 Types

Assertion programs make use of higher-level data types, and their associated functional operators, in order to represent complex patterns of specification succinctly. The language is strongly typed, to support the semantics, as well as for type-checking and aiding reasoning. In principle, any relevant range of data types and operators may be used within our framework, matching the specific requirements of the design at hand. For the purposes of this chapter, however, we will limit our data types to the Boolean type, $\mathbf{bool}$, and the family of bounded non-negative integer types $\mathbf{int}(n)$ for $n \in \mathbb{N}$, where $\mathbf{int}(n)$ describes those non-negative integers that are strictly less than n. As operators on these types, we will allow the standard Boolean operators, together with conventional operators for arithmetic modulo n.

5.2.3 Variables and Declarations

Assertion programs contain two classes of variables: program variables and indexing variables. *Program variables* capture the state of the high-level model in a given time-step. *Indexing variables* are used for defining locally scoped generalizations, such as generalized parallel composition and Boolean quantification. Only program variables are evaluated with respect to the current temporal context. For instance, it makes sense to refer to the value that a program variable held in the last time-step, but not to the previous value of an indexing variable.

For clarity and typing, program variables, constants and type aliases are declared at the beginning of a program, in the *declaration block.* Program variables are declared with the **var** keyword, as such:

$$\textbf{var}\ \ \textit{variable\_name}\ :\ \textit{variable\_type}$$

Constant declarations provide a simple and useful means of easily parameterizing specifications. In practice, parameterization would be better serviced by a complete module system, but this is beyond the scope of our work. Constants are defined with the **const** keyword. For example:

$$\textbf{const}\ \ \mathrm{SIZE}\ =\ 10$$

It is also useful to be able to define type constants, or aliases, via declarations of the form

$$\textbf{type}\ \ \mathrm{index}\ =\ \textbf{int}(\mathrm{SIZE})$$

5.2.4 Expressions

Assertion program expressions allow us to describe how to compute the values with which to update the program state. Expressions can consist of literal constant values, program or indexing variables, or be constructed using Boolean operators, equality or other data-specific functions. This allows us to build expressions such as

$$((1 + \mathrm{count}) = \mathrm{max}) \wedge \mathrm{done}$$

We will also allow conditional expressions, using the same syntax as the GTL conditional construct:

$$\textit{guard}\ \rightarrow\ \textit{if\_expression} \mid \textit{else\_expression}$$

We introduce the special operator **last** which modifies an expression to refer to its value in the preceding time-frame. For example, $\textbf{last}(\mathrm{input} + 1)$ refers to the value assigned to program variable input in the preceding synchronous time-step, incremented by one.

This form of temporal expression is useful for several reasons. First, since our output may be functionally dependent either the current or past values of other signals, we require a means of specifying temporal references. Second, being able to temporally shift signal specifications is useful, since hardware interfaces are frequently affected by pipelining adjustments. Third, the use of $\mathbf{last}$ expressions conveniently mirrors the *Yesterday* operator of GTL. This correspondence is vital in the translation which we will present from programs to GTL properties.

5.2.5 Statements

The state transitions for our high-level models are described using assignment statements that set the values of program variables at each synchronous time-step. Statements can be viewed either imperatively or declaratively. Interpreted imperatively, the statements provide instructions that describe how to update the state of the model. This gives the specifications an intuitive feeling, and mimics many common programming languages. In contrast, assignments can also be viewed declaratively, as defining constraints on the model's possible state transitions. This alternative viewpoint also forms a logical basis for our translation to GTL properties, presented in the subsequent chapter.

Setting a program variable is described by an *assignment* statement of the form:

$$identifier := expr$$

where $identifier$ is the variable to be set and $expr$ is the expression to evaluate for the update value. For example, $\mathrm{count} := \mathbf{last}(\mathrm{count}) + 1$ stipulates that the program variable count is be incremented at every time-step. From the declarative viewpoint, the statement can be viewed as the constraint given by the corresponding equality: $\mathrm{count} = \mathbf{last}(\mathrm{count}) + 1$. Such an equation says that in any model transition, the value of count is one greater in the successor state than in the original state.

Statements are composed via *parallel composition*, written using the infix symbol ‘ $\|$ ’, or, alternatively, as a line-break. For example, the statement

$$\mathrm{count}' := \mathbf{last}(\mathrm{count}) \;\|\; \mathrm{count} := \mathrm{count}' + 1$$

says that count' should be set to the previous value of count and count should be set to the current value of count' plus one. In terms of logical specification, parallel composition can be seen as the conjunction of the logical constraints of its two component statements.

Since parallel updates take place simultaneously, two potential problems are raised by this form of composition. First, it is possible to construct inconsistent programs

```
var reset : bool
var count : int(256)

model
  if reset then
    count := 0
  else
    count := last(count) + 1
```

Figure 5.2: Program for an 8-bit Counter

that assign different values to the same variable. For example, $\mathrm{a} := 1 \parallel \mathrm{a} := 2$ does not have a clear interpretation. We will judge such multiple assignments to be ill-formed statements, which it is the responsibility of the user to avoid. Second, mutually recursive dependency loops such as $\mathrm{a} := \mathrm{b} \parallel \mathrm{b} := \mathrm{a}$ can be created, which do not have a well-founded evaluation strategy. We will avoid this problem by requiring of the user that there are no dependency loops in the programs.

Programs also include conditional statements for specifying control, using the **if**, **then** and **else** keywords. Such statement follow the standard expectations of a conditional. An example of this is shown in the assertion program in Figure 5.2, which models a simple 8-bit counter.

5.2.6 Arrays

We include arrays as built-in composite data types in assertion programs, as they are frequently useful for the describing common hardware specification patterns that mirror structural circuit duplication and memories. Array types consist of an element type and a index type, written in the form

$$\mathit{element\_type}[\mathit{index\_type}]$$

Array-lookup expressions are written in the form

$$\mathit{array\_name}[\mathit{index\_expression}]$$

Notice that we allow arrays to be indexed over any type, and not just the integers. In this sense, they resemble arbitrary value maps. For example, we might describe an address to be an array of 8-bit values, and describe a memory to be an array of data

```
type data = int(256)
type addr = int(8)

model
  var mem : data[addr]
  var data_in, data_out : data
  var write_addr, read_addr : addr
  var write_enable, read_enable : bool

  for $i$ : addr do
    if write_enable $\wedge$ ($i$ = write_addr) then
      mem[$i$] := data_in
    else
      mem[$i$] := last(mem)[$i$]
  if read_enable then
    data_out := last(mem)[read_addr]
```

Figure 5.3: Program Segment for an 8-Place 32-bit Memory

indexed by addresses:

```
type addr = bool[int(8)]
type mem  = data[addr]
```

Modeling often requires simultaneous assignment to array indices within a single time-step. To express this, we use **for** statements to describe indexed parallel composition, generalizing parallel composition over typed indexing variables. As an example Figure 5.3 shows a program that models an 8-place memory for 32-bit integers, using a **for** statement to update each index of the memory array model at every time-step.

In order to achieve a clean semantics, we will place the restriction that assignment statements can only assign the primitive types of Boolean and integers. This enforces the degree of atomicity with which variables can be set. As a result, an array cannot be copied directly with a single assignment of the form $\mathrm{a} := \mathrm{b}$. We can, however, always introduce **for** statements to handle such cases:

```
for i : type do a[i] := b[i]
```

5.2.7 Circuit Interface

The *interface block* describes how the inputs to the high-level model relate to the input nodes of the circuit implementation. Circuit input nodes are represented by special

Boolean program variables, declared with lines of the form:

$\mathbf{node}$ reset, write_en

The names of these variables correspond with the string identifiers used by circuit netlist models, and necessarily have type $\mathbf{bool}$. Since they represent environmental input, they may not be assigned to. Buses of nodes may be declared using the conventional square bracket hardware notation for defining vector index ranges. For example, the declaration

$\mathbf{node}$ data_in$[7:0]$

declares the array of the nodes named data_in$[7]$,data_in$[6]$, ..., data_in$[0]$ in the circuit model. For such a bus, data_in is assigned the array type $\mathbf{bool}[\mathbf{int}(8)]$.

By introducing these variables, we can now express the mapping between these variables and the high-level model inputs using the same types of statements that are used to define the model. At each time-step, the inputs to the high-level model are set using assignments evaluated from circuit input expressions. This follows the approach set out in Figure 5.1.

Since these statements follow the same semantics as statements in high-level models, the interface block can be seen as merely a parallel extension to the high-level model. By enforcing the separation between the two, however, we can greatly enhance the clarity of the high-level model by separating it from the messy details of the interface. It is also common for there to be many different hardware designs and interfaces for different circumstances even though they implement the same functional purpose. Therefore our separation of model and interface allows for the model to be reused and only the interface part need be rewritten for different circuit implementations. The use of a separate interface ensures that the high-level model does not contain details such as:

- The names of circuit nodes, which, following synthesis, many be long and obscure.

- The binary implementation encodings for high-level data types.

- The timing details about when circuit signals are stable.

- Small aspects of control (termed ‘pre-logic’) that conditionally select, route and decode parameters in an implementation specific manner.

```
interface
  node force_bypass_rd_L_335
  node res_op_FH_334[35 : 0]
  node bypass_H_333[31 : 0]
  node res_op_out_H_338[31 : 0]

  write_addr := bvn2int_raw(res_op_FH_334[4 : 1])
  read_addr := bvn2int_raw(res_op_FH_334[4 : 1])
  write_enable := res_op_FH_334[0]
  read_enable := ¬res_op_FH_334[0]
  if last(force_bypass_rd_L_335) then
    data_in := last(last(bypass_H_333))
  else
    data_in := res_op_FH_334[35 : 4]
```

Figure 5.4: Example Interface for an 8-Place 32-bit Memory

The methodology surrounding the Forte verification platform has demonstrated the benefits of separating such hardware interfaces from the core functional specification, using what it terms *circuit APIs* [SJO$^+$05].

5.2.7.1 Encoding Mappings

Our programs make use of the library of functions provided by Forte [SJO$^+$05] for mapping between high-level data types and bit-vector representations. Forte includes functions for bit-vector arithmetic, conversion, extension, contraction and signing. An example is the useful map int2bvn, which converts non-negative integers to their unsigned bit vector encoding, and bvn2int, which converts back again.

Example Suppose we are to use the program in Figure 5.3 to verify a small memory. In the implementation at hand, the memory receives either a read or write operation encoded on the bus res_op_FH_334$[35 : 0]$. The node res_op_FH_334$[0]$ is set true if this operation is a write, otherwise the operation is a read. The address is encoded as an unsigned integer in res_op_FH_334$[4 : 1]$. If this is a write then the data is in res_op_FH_334$[35 : 4]$, unless a special flag force_bypass_rd_L_335 was set in the previous time-step, in which case the data should come from the values on bus bypass_H_333$[31 : 0]$ two time-steps previously. The assertion program interface for this is shown in Figure 5.4, and characterizes typical interfaces found during microprocessor verification.

5.2.8 Assertions

The final block of an assertion program specifies properties that link the output of the circuit to the behaviour of the high-level model. Such properties can be seen as an output interface mapping between the two. Assertions are of the form

$$antecedent \quad \Rightarrow \quad consequent$$

where the antecedent and consequent are Boolean expressions, and the consequent is an expression involving only circuit nodes and indexing variables. Each line asserts that if the circuit is operating in parallel with the interface mapping and high-level model, within the same environmental input trace, then at every step, if the antecedent condition is true, then the consequent condition must also evaluate to true. Typically, the antecedent corresponds to a particular high-level model state, and the consequent describes the resulting signals that we would expect to see in the circuit implementation under such conditions. Often this is used to assert simple output equivalence.

For example, suppose we have a high-level model of a counter, consisting of program variable count. We can assert that under the conditions in which the count is zero, the circuit under verification should signal empty:

$$(\mathrm{count} \ = \ 0) \quad \Rightarrow \quad \mathsf{empty}$$

When programs are compiled into symbolic ternary simulations, as described in the subsequent chapter, the antecedents of these assertions are used to calculate those circuit execution traces that need to be simulated, and the consequents are used to assert that the right values occur in these traces.

In order to bridge the gap between antecedent and consequent, it is often necessary to introduce indexing variables that remain symbolic in the corresponding simulation run. This is achieved via indexed **forall** assertions. For example, we might make the following assertion about the memory model of Figure 5.3:

$$\textbf{forall}\, i : \mathrm{data} \,.\, ((\mathrm{data\_out} = i) \Rightarrow (\mathsf{res\_op\_out\_H\_338}[31:0] = i))$$

5.3 Formal Semantics

This section introduces the formal semantics for assertion programs. A formal grammar for the programs can also be found in Appendix B.

Definition 5.3.1. *Assertion programs satisfy the grammar in Appendix B together with the conditions that there are no atemporal cyclic dependencies between the values assigned to the program variables and conditional guards cannot depend on the variables set within either of their branches (i.e. we cannot set some element of state conditional upon its future value).*

These sanity conditions are used later in order to ensure that the expressions assigned to variables can be calculated directly from the inputs available.

We provide a semantics for programs by viewing statements as declarative constraints on execution traces. A circuit then satisfies each program trace if each of the assertions holds in every circuit trace with matching inputs.

As we have already described, assertion programs make use of *program variables*, which we will denote with the finite set $\mathsf{ProgVars}$, and *indexing variables*, for which we will use the finite set $\mathsf{IndexVars}$. The circuit nodes $\mathcal{N} \subseteq \mathsf{ProgVars}$ are described using special program variables that cannot be assigned to. Let Types be the set of types in use, containing at least a Boolean type, $\mathbf{bool}$.

We designate Values to be the set of primitive values, which, for illustrative purposes will contain true, false and the non-negative integers. We will define values to map from the set of types to the sets of values that a particular type represents. For example, we might have that $\mathrm{values}(\mathbf{int}(4)) = \{0, 1, 2, 3\}$. For our purposes, such sets are assumed to be finite.

The state of a program is modeled as a finite words of stores, corresponding to execution histories, where a store is a map from locations to values. Since an expression may incorporate arbitrarily many $\mathbf{last}$ expressions, we must keep track of the complete past execution history. A store location consists of a program variable, together with a list of lookup indices, giving us the set:

$$\mathsf{Locations} = \mathsf{ProgVars} \times \mathsf{Values}^*$$

For example, the location associated with address $\mathrm{a}[3][4]$ is $(\mathrm{a}, [3, 4])$, and the address for v is $(\mathrm{v}, [])$. A store is then a map from locations to values:

$$\mathsf{Stores} = \mathsf{Locations} \rightarrow \mathsf{Values}$$

The possible transitions between such stores are determined by the assignment statements that a program contains.

5.3.1 Evaluating Expressions

Expression evaluation is performed using two different variable contexts, which cater for program and indexing variables respectively. Program variables are evaluated using the store history, $\sigma \in \mathsf{Stores}^+$. The index store, $\nu \in \mathsf{IndexVars} \to \mathsf{Values}$, is the local context in which to evaluate indexing variables.

We will define a natural semantics [Kah87] for expression evaluation using the judgment

$$E \xrightarrow[\nu]{\sigma} e$$

which means that expression E evaluates to value e in store history σ and index store ν. To define expression evaluation, we first define how to evaluate locations from assignable expressions, such as program variables and array indices. These expressions will be evaluated by first evaluating their store location, and then looking up this location in the current store. The judgment

$$E \xrightarrow[\mathsf{loc}]{\sigma,\nu} l$$

will be used to denote that an expression evaluates to location $l \in \mathsf{Locations}$. Program variables evaluate to the location consisting of themselves with the associated empty index. Array indices are evaluated by creating a location from the name of the array and the evaluations of each index expression:

$$\frac{\mathrm{x} \in \mathsf{ProgVars}}{\mathrm{x} \xrightarrow[\mathsf{loc}]{\sigma,\nu} (\mathrm{x}, [])} \qquad \frac{A \xrightarrow[\mathsf{loc}]{\sigma,\nu} (\mathrm{x}, \pi) \quad E \xrightarrow[\nu]{\sigma} e}{A[E] \xrightarrow[\mathsf{loc}]{\sigma,\nu} (\mathrm{x}, \pi e)}$$

An expression that can be evaluated to a location can then be evaluated to a value by being looked up in the current store:

$$\frac{E \xrightarrow[\mathsf{loc}]{\sigma s,\nu} l}{E \xrightarrow[\nu]{\sigma s} s(l)}$$

Indexing variables are simply evaluated by looking up their value in the index store:

$$\frac{u \in \mathsf{IndexVars}}{u \xrightarrow[\nu]{\sigma} \nu(u)}$$

Functional application is evaluated by first evaluating the arguments in the same context, and then applying the function:

$$\frac{E_i \xrightarrow[\nu]{\sigma} e_i}{f(E_0, \ldots, E_n) \xrightarrow[\nu]{\sigma} f(e_0, \ldots, e_n)}$$

We will assume that equality and Boolean negation correspond to specific types of functional application, with their standard definition. We will also allow for the other Boolean operators, as well as conditional expressions, although these are to be interpreted lazily. For example, if the first argument of Boolean conjunction evaluates to false, then the second argument need not be evaluated. Universal quantification is defined lazily by:

$$\frac{\forall e \in \text{values}(t)\ .\ (E \xrightarrow[\nu[\mathrm{x}\mapsto e]]{\sigma} \text{true})}{(\textbf{forall}\ \mathrm{x} : t\ .\ E) \xrightarrow[\nu]{\sigma} \text{true}} \qquad \frac{\exists e \in \text{values}(t)\ .\ (E \xrightarrow[\nu[\mathrm{x}\mapsto e]]{\sigma} \text{false})}{(\textbf{forall}\ \mathrm{x} : t\ .\ E) \xrightarrow[\nu]{\sigma} \text{false}}$$

and existential quantification is defined dually. Finally, last-expressions are evaluated by shifting the program variable store context by one time-step backwards:

$$\frac{E \xrightarrow[\nu]{\sigma} e}{\textbf{last}(E) \xrightarrow[\nu]{\sigma s} e}$$

Notice that as a result, there is no guarantee that an expression can be evaluated at all, since the current execution history may not be sufficiently lengthly.

5.3.2 Statements

We will write the judgment

$$\sigma \xrightarrow[\nu]{T} \sigma s$$

to indicate that statement T is consistent with the transition from state history σ to σs in indexing variable context ν. Transitions always corresponds to the addition of one time-step to the store history. The skip statement poses no constraints on execution:

$$\frac{}{\sigma \xrightarrow[\nu]{\textbf{skip}} \sigma s}$$

Assignment statements assert that current store value for a program variable matches the value of the assigned expression:

$$\frac{E \xrightarrow[\nu]{\sigma s} e \quad s(\mathrm{x}) = e}{\sigma \xrightarrow[\nu]{\mathrm{x} := E} \sigma s}$$

A transition respects a parallel composition of statements if it respects both statements at once:

$$\frac{\sigma \xrightarrow[\nu]{T_1} \sigma s \quad \sigma \xrightarrow[\nu]{T_2} \sigma s}{\sigma \xrightarrow[\nu]{T_1 \parallel T_2} \sigma s}$$

The branches of conditional statements must only constrain the state when the guard evaluates correspondingly:

$$\frac{E \xrightarrow[\nu]{\sigma s} \text{true} \quad \sigma \xrightarrow[\nu]{T_1} \sigma s}{\sigma \xrightarrow[\nu]{\textbf{if } E \textbf{ then } T_1 \textbf{ else } T_2} \sigma s} \qquad \frac{E \xrightarrow[\nu]{\sigma s} \text{false} \quad \sigma \xrightarrow[\nu]{T_2} \sigma s}{\sigma \xrightarrow[\nu]{\textbf{if } E \textbf{ then } T_1 \textbf{ else } T_2} \sigma s}$$

The statements provided by a **for** expression apply under every valuation of the corresponding indexing variable:

$$\frac{e \in \text{values}(t) \quad \sigma \xrightarrow[\nu[u \mapsto e]]{T} \sigma s}{\sigma \xrightarrow[\nu]{\textbf{for } u : t \textbf{ do } T} \sigma s}$$

Notice that these conditions can result in a nondeterministic transition system. In particular, if a particular program variable is not constrained by an assignment statement, then it may take on any value at that time-step.

5.3.3 Transition System

We will say that a statement is deadlock-free if every state history has a successor. Statements that introduce deadlock are those that assert inconsistent state updates, such as $\mathrm{x} := \text{true} \parallel \mathrm{x} := \text{false}$.

Definition 5.3.2. *A statement T is deadlock-free if for every $\sigma \in \mathsf{Stores}^+$, there exists some $s \in \mathsf{Stores}$ such that*

$$\sigma \xrightarrow[\nu]{T} \sigma s$$

Assuming that a given statement T is deadlock-free, then it has associated transition system given by $(\mathsf{Stores}^+, \xrightarrow[\{\}]{T})$. If the statement has a bounded nesting depth n of **last** expressions, we can reduce this infinite state structure to the finite Kripke Structure $(\bigcup_{i \leq n+1} \mathsf{Stores}^i, \xrightarrow[\{\}]{T})$, since the state is not dependent on the entire execution history. Given a program $\mathbf{AP}$, with model block $\mathbf{model}(\mathbf{AP})$ and interface block $\mathbf{interface}(\mathbf{AP})$, let its Kripke Structure be given by:

$$\mathcal{K}_{\mathbf{AP}} = (\bigcup_{i \leq n+1} \mathsf{Stores}^i, \xrightarrow[\{\}]{\mathbf{model}(\mathbf{AP}) \parallel \mathbf{interface}(\mathbf{AP})})$$

5.3.4 Satisfaction

Now that we have defined this transition system, we can describe what it means for a circuit to satisfy the assertions laid down by an assertion program. Recall that we intend to subject the high-level model and circuit to the same input traces, and then assert that every output guaranteed by the high-level model is also guaranteed by the circuit. First, we will define the traces that can be performed when a program runs in parallel with a circuit:

Definition 5.3.3 (Monitor Traces)**.** *Given a trace* $\sigma \in \mathrm{tr}(\mathcal{K}_{\mathbf{C}})$ *of circuit model* $\mathcal{K}_{\mathbf{C}}$*, then the set* $\mathrm{monitor}_{\mathbf{AP}}(\sigma)$ *of corresponding monitor traces for program* $\mathbf{AP}$ *is defined by those traces of the same length as* σ *that are in lock-step agreement over the circuit nodes:* $\mathrm{monitor}_{\mathbf{AP}}(\sigma) =$

$$\{\ \mathrm{last}(\pi)\ \mid\ \pi \in \mathrm{tr}(\mathcal{K}_{\mathbf{AP}})\ \ \wedge\ \ |\sigma| = |\pi|\ \ \wedge\ \ \forall i < |\sigma| \,.\, (\sigma_i = (\mathrm{last}(\pi)_i)_{|\mathcal{N}})\ \}$$

Definition 5.3.4 (Satisfying Traces)**.** *For circuit* $\mathcal{K}_{\mathbf{C}}$*, a given circuit trace* $\sigma \in \mathrm{tr}(\mathcal{K}_{\mathbf{C}})$ *satisfies assertion* $\mathrm{A} \Rightarrow \mathrm{C}$ *of program* $\mathbf{AP}$ *when for every indexing context* $\nu \in \mathsf{IndexVars} \rightarrow \mathsf{Values}$*,* σ *satisfies* C *whenever every monitor trace of* σ *satisfies* A*:*

$$\Big(\forall \pi \in \mathrm{monitor}_{\mathbf{AP}}(\sigma)\ .\ (\mathrm{A} \xrightarrow[\nu]{\pi} \mathrm{true})\Big) \quad \textit{implies} \quad (\mathrm{C} \xrightarrow[\nu]{\sigma} \mathrm{true})$$

Definition 5.3.5 (Satisfying Circuits)**.** *A circuit* $\mathcal{K}_{\mathbf{C}}$ *satisfies program* $\mathbf{AP}$ *when every trace of the circuit satisfies every assertion* $\mathrm{A} \Rightarrow \mathrm{C}$ *from* $\mathbf{AP}$.

Notice that our assertions require that the consequent hold only if the antecedent holds for *every* possible monitor trace of a given circuit (input) trace. Therefore, if the antecedent nondeterministically holds for a given input trace then no assertion is made at all. For example, suppose we are modeling a memory cell, that has a nondeterministic starting state. We assert that if the contents of the memory cell matches some constant $\mathbf{d}$, then the circuit should output this value. Since the initial state is not specified, there will be some execution traces where this antecedent holds initially. Under our definition of satisfaction, however, the assertion only applies after those input traces that *guarantee* the antecedent under all possible execution traces. Therefore, the specification will only assert that the memory outputs the correct value after an input trace occurs that first writes to it.

```
model
  var count : int(5)
  var reset : bool

  if last(reset) then
    count := 0
  else
    count := last(count) + 1

interface
  node resetnode, signalnode

  reset := resetnode

assert
  (count = 0) ⇒ signalnode
```

Figure 5.5: Counter Assertion Program

5.3.5 Example

Consider the program in Figure 5.5, which describes a simple counter that operates modulo 5. The circuit under test contains at least a reset node, resetnode, and a node signalnode that is supposed to signal true whenever the counter is zero. The reset node should set the counter to zero after a delay of one cycle. We consider exactly what it means for a circuit to satisfy this specification.

Satisfaction requires that the assertion $(\mathrm{count} = 0) \Rightarrow$ signalnode holds for any input trace given both to the program and the circuit. First, we will consider input traces of length one. The input consists solely of whether the reset node is high or not, so there are only two possible cases. In either case, the input trace does not force the value of the count to zero in the current time-step, so the antecedent fails, and the trace is satisfied.

Now let us consider traces of length two. A similar scenario now holds, with the exception of the cases where resetnode is high in the first time-step. Under this condition, the constraints of the program transition force count to be zero in the subsequent time-step, so that the antecedent condition evaluates to true. We therefore assert that if resetnode is high then signalnode must be high in the subsequent time-step.

By the time we reach traces that are longer than length five, another option presents itself. The antecedent now evaluates to true either if resetnode is high in the penulti-

...10
...11
...1000000
...1000001
...100000000000
...100000000001
⋮

Table 5.1: Antecedent Satisfying Counter Input Traces

mate step, or if it was high six steps previously, and has been low for the following five steps. Clearly, as traces lengthen further, we will assert that signalnode must be high $1 + 5n$ steps after the most recent reset. The traces for which we assert a signal are illustrated in Table 5.1.

5.4 State Variables

When a program variable is not assigned an update in a given time-frame, its value is effectively nondeterministic. This quality is useful for modeling hardware, since it allows us to write partial specifications. It is often easier, however, to specify transitions using variables with persistent state, that retain their value if an update is not explicitly given. Of course, we can specify such persistence using assignments of the form $\mathrm{a} := \mathbf{last}(\mathrm{a})$, but including a large number of such assignments can cloud the overall view of the specification. We therefore provide an alternative mechanism for this, by allowing individual program variables to be marked as *state variables*, meaning that their values persist by default. Figure 5.6 shows a memory cell program that has been simplified through the use of a state variable.

A program that uses state variables can be changed into an equivalent program that does not include state variables by adding extra state preservation assignments of the form $\mathrm{a} := \mathbf{last}(\mathrm{a})$ at various points in the program. For example, the memory cell specification in Figure 5.6 (i) can be obtained automatically from the state variables used in (ii). We do not describe this procedure in detail, but note that it involves calculating a symbolic condition under which no assignments update the value of a state variable, and inserting a top-level state preservation assignment guarded by this condition.

```
var cell : bool
var write, data : bool

if write then
  cell := data
else
  cell := last(cell)
```

(i) Using Standard Variables

```
state cell : bool
var write, data : bool

if write then
  cell := data
```

(ii) Using a State Variable

Figure 5.6: Memory Cell with Persistent State

5.5 Related Work

One of the most common methods of property specification for hardware verification is to use assertion languages like Property Specification Language (PSL) [PSL05], ForSpec [AFF$^+$02] or System Verilog Assertions (SVA) included in-line within the circuit description itself. These linear assertion languages tend to use a mix of temporal logic and regular expressions, making them useful for checking individual aspects of a design's functionality. GSTE verification, however, tends to be based around complete component specifications, and therefore resembles *refinement checking* rather more than property verification. For example, the GSTE FIFO property in [YS02] describes *all* the requirements of the device, in a single assertion graph, by essentially describing a complete abstract reference model. Although such abstract states can be captured using μ-expressions in GTL, this approach does not scale well, especially in terms of human legibility. Therefore, because they can succinctly describe these arbitrary abstract state transition systems, assertion programs are more suitable for GSTE-style verifications. We therefore use assertion programs and relate the implementation to the specification via trace equivalence. Similar approaches using reference models and trace equivalence for processor verification can be found in [Cyr03, Kai05].

There is a large range of existing languages that may also be suitable for describing such high-level models. The most important characteristic required is the ability to describe the synchronous parallel behaviour of *reactive* systems [Hal98]. Broadly speaking, these languages can be divided into: synthesizable hardware description languages, hardware modeling languages, and languages used purely for specification.

5.5.1 Hardware Description Languages

Hardware description languages have been widely used to describe systems at all levels of abstraction, and have the benefit of being familiar to many engineers. Some of the most commonly used include Verilog [TM91], VHDL [Nav97], and SystemC [GLMS02]. One main drawback of these languages is that they are not defined with a formal semantics. Formalization attempts have proved difficult [Gor95], although limited progress has been made [Gor97, PLC94]. These languages also contain significantly more complexity than required for our purposes, for example by including support for sequentiality, timing, synchronization conditions and recursion. Such complexity would significantly affect the cleanliness and reasoning ease of our specifications. Furthermore, since these languages are designed for providing implementations rather than reference models, they can also tend to over-specify systems, not allowing for partial specification.

Aside from the most popular hardware description languages, there are other options built around solid formal semantics. The synchronous programming languages [Hal98], such as Esterel [BC85], Signal [RMC94] and Lustre [HLR92] are particularly relevant for our requirements, due to their alignment with the temporal models of hardware execution. Of these, assertion programs most closely resemble the dataflow language Lustre [HLR92], which also uses assignment-based constraints built from expressions with a last-time operator. Lustre models include a concept of initial state, but ternary simulation models do not, so relating the two via refinement could prove troublesome. The mutually recursive equation definitions described by both Lustre and assertion programs can also be embedded as mutually recursive functional definitions in a functional programming languages, leading to languages such as Lava [CS00] and the original formulation of Bluespec [HA00].

5.5.2 Modeling Languages

There is also a class of similar languages used to describe design models for verification. Like assertion programs, these languages generally declare some form of model state, and then describe behaviour as a series of transition constraints, based around this state.

One example is the modeling language for the Symbolic Model Verifier (SMV) [McM99]. As with assertion graphs, assignment in the SMV language can be interpreted as a logical equality, allowing reasoning in the flavor of Lamport's Temporal Logic of Actions [Lam94]. Unlike assertion programs, temporal aspects in SMV are slightly more rigid. Instead of allowing arbitrary $\mathbf{last}$ expressions, there are two forms of assignment: instantaneous assignment, written $\mathtt{x} := \mathtt{E}$, and delayed assignment, written $\mathtt{next(x)} := \mathtt{E}$. Our use of the last operator is not only more succinct in certain cases, but also provides a closer fit when we come to translate to GTL properties, as we will see in the subsequent chapter.

Assertion programs also share much in common with other transition-based modeling languages. Murϕ [DDHY92] is one such language, developed for protocol verification. As with assertion programs, transitions are specified as updates to abstract state, and assignments can be guarded or composed in indexed parallel. One crucial difference, however, is that updates are grouped into guarded blocks called *rules* (or *transitions*), *at most one* of which is nondeterministically chosen to execute in each time-step. Although such nondeterminism in the order of update execution is useful for simplifying protocol models, it does not help for GSTE verifications, where specifications are required to be cycle-accurate. Similar languages, such as synchronized transitions [GS90], CIRCAL [Mil85] and UNITY [CM88] are also structured in this form. The language of the Symbolic Analysis Library (SAL) [dMOS03] is an example that includes support for a mixture of both interleaved and truly synchronous updates.

Other hardware modeling languages used to model hardware deviate significantly from the semantics of assertion programs. In SML [BC86], for example, timed sequential composition is permitted by associating each statement with a given discrete time length, allowing the modeling of algorithmic state machines that are more prone to changing mode sequentially.

5.5.3 Specification Languages

There are also languages similar to assertion programs that exist purely for the abstract *specification* of systems. The Abstract State Machine Language (AsmL) [GRS05] is an executable specification language for formally specifying and dynamically exploring hardware and software models. As with assertion programs, AsmL programs have a procedural flavor, modeling systems with sets of synchronous updates to some abstract program state. The main assertion program constructs, such as indexed parallel composition, are also present. AsmL is primarily intended for software, so it is also object-oriented, and supports sequential composition. Although these constructs operate at a higher level of abstraction than we are aiming for with assertion programs, work by Hanna and Melham proposes to link the two levels.

The e specification language [HMN01] was developed to aid automatic hardware test-case generation in a richer programming environment than those of traditional hardware description languages. Assertions are specified with an assertion language similar to PSL [PSL05]. The language defines extensive rules for test-case selection heuristics, but allows C-like procedures for writing assertion-based monitors. The Tempura language [Mos86] is another example of a logic-based specification language with similar sequential imperative features. Whilst both of these languages permit high-level reference models similar to assertion programs, we believe that their use of sequentiality is unnecessary for most GSTE properties, and makes them less amenable to direct reasoning. Furthermore, neither contains constructs such as indexing variables, which are of integral importance when we come to translate assertion programs into runs of symbolic ternary simulation.

5.5.4 STE-Based Specifications

There are several existing formalisms to express specifications for symbolic ternary simulation at higher-levels of abstraction. Joyce and Seger [SJ92] connect the HOL [GM93] theorem prover to an STE model checker so that functional definitions in HOL can be used to provide more expressive user-defined abstractions. At the same time, they proposed a basic specification language and a library of pre-defined bit-vector arithmetic functions to standardize these abstractions. Whilst the original proposals were quite limited in scope, this work has provided a basis for the specification libraries contained within the Forte verification platform [SJO$^+$05], the use of which we have proposed for assertion programs.

Beatty [BB94] describes specifications of considerable complexity by using sets of actions defined in terms of their pre- and post-conditions. Jain and Nelson [NJB97] extend this specification methodology to unbounded properties, introducing the graphical specifications on which assertion graphs are based. Relatively little work has attempted to abstract above the detail of these assertion graphs. Yang and Goel [YG02] describe how graphs can be automatically generated using functional programing scripts. Whilst this does allow the user to progressively build-up parameterizable abstract concepts of assertion patterns, the situation is not much improved, since these scripts are not compositional or standardized, and are often as unstructured as the graphs themselves.

Another approach is described by Kaivola [Kai05], based on creating STE runs from high-level models embedded in a functional language. The high-level state is given as a data type in the function language, and the model is described using Boolean functions on pairs of states that characterize the model transition relation. This gives the specifications a feel somewhat similar to assertion programs, since it encodes constraints on the current model state in terms of the current and previous state.

Chapter 6

Verifying Assertion Programs

This chapter describes an approach for generating the simulations structures that can be used to verify that a circuit satisfies a given assertion program. Simulation generation is described by defining a collection of rules within an interactive framework, allowing us to combine the formalisms and techniques that we have introduced so far.

Within this framework, we use our low-level logic, generalized trajectory logic (GTL), to progressively build up an exact representation of the simulation being built. We extend this logic, first by allowing a means of embedding program propositions, and second by creating a vector form that helps with the management of sizable simulations.

The core contribution is a set of rules that can be used to create concise representations of the patterns of input sequences that bring about a given antecedent condition for a particular program. This is done by using weakest precondition calculations to progressively rewrite antecedent conditions as formulas of GTL, until a point is reached where the conditions are expressed solely in terms of the circuit inputs. We also describe how the techniques for abstraction control, simplification and decomposition from Chapter 4 can be added as rules in our framework. These can then be applied throughout the generation process to affect the resulting simulation.

6.1 Extending GTL

For simulation generation it is necessary to extend GTL to be able to capture the relationship between input traces and the program states they induce. In this section, we describe how we can unify the semantics of GTL and assertion programs, to allow Boolean program expressions as GTL propositions. We then go on to describe *vector GTL*, which uses a set of equations to capture a simulation, rather than nested μ-expressions.

6.1.1 Adding Assertion Program Expressions

Recall from Section 5.3.4 that a circuit satisfies an assertion program if the two agree on outputs when they are running in parallel on the same input traces. During simulation generation, we must therefore reason about the parallel composition of the program and circuit in question.

We create a connection between input traces and program states by extending GTL to include formulas that express that given program expressions evaluate to truth. The connection is natural, owing to the similarities in the temporal and symbolic foundations of the two formalisms.

We first allow GTL's indexing variables to use the same types that are found in assertion programs. We can then use the same context for both GTL and program index variables, since both classes of variables are semantically independent of time and used solely for quantification. As a result, the index context in extended GTL is a map from the set of all index variables, $\mathsf{IndexVars}$, to the set of program values, Values.

We include Boolean-typed program expressions as propositions in extended GTL. These are written in the form $\mathrm{eval}(E)$, where E an expression.

Definition 6.1.1. *The proposition* $\mathrm{eval}(E)$ *is true only in those combined circuit and program states where* E *evaluates to* true*:*

$$\pi \in \|\ \mathrm{eval}(E)\ \|_{\rho}^{\nu} \quad \textit{if and only if} \quad E \xrightarrow[\nu]{\pi} \mathrm{true}$$

Recall that a program expression is not guaranteed to evaluate at all, for example, if there is not enough past store data to evaluate the required depth of $\mathbf{last}$ expressions. Therefore, $\mathrm{eval}(E)$ is false *either* if E evaluates to false, or if E does not evaluate at all. The eval operator plugs the gap between the three-valued and Boolean models, at the same time allowing GTL formulas to express the requirements of assertion program satisfaction (Section 5.3.4).

6.1.1.1 Example

As an example, we will consider the GTL formula

$$\exists i\ :\ \mathbf{int}(256)\ .\ \mathbf{Y}(\mathrm{eval}((i < 64) \wedge (\mathrm{int2bvn}(i) = \mathsf{in}[7:0])))$$

that contains the program expression $(i < 64) \wedge (\mathrm{int2bvn}(i) = \mathsf{in}[7:0])$. This formula says that in the previous time-step, the integer encoded by the 8-bit bus $\mathsf{in}[7:0]$ held a value less than 64. It is semantically equivalent to the formula $\neg\mathsf{in}[7] \wedge \neg\mathsf{in}[6]$.

6.1.1.2 Transforming Expressions into GTL

We progressively generate our simulation structures, by transforming program expressions into simulation steps expressed by the constructs of GTL. This is achieved through a series simple of rules that relate the two. For example, conjunction in an assertion program is equivalent to GTL conjunction: $\mathrm{eval}(E \wedge E') = \mathrm{eval}(E) \wedge \mathrm{eval}(E')$. This step allows us to simulate the condition represented by this conjunction by simulating the conditions represented by each conjunct in turn, and then taking the greatest lower bound, as defined by GTL model checking.

6.1.2 Vector GTL

Because of their flat syntax, μ-expressions are particularly useful simulation representations for theoretical reasoning and automated manipulation. But nested fixed-point simulation descriptions can quickly become incomprehensible to the human reader, making simulations difficult to manually manipulate and debug. Assertion programs have addressed this problem for high-level specifications, but an alternative representation is also desirable for the low-level. For this we introduce *vector GTL*, which has the same syntax and semantics as GTL except that fixed-points are defined using mutually dependent systems of equations.

For example, consider the standard GTL property

$$\mathsf{rd} \wedge (\mu \mathrm{Written}.(\mathrm{in}\ \mathbf{is}\ u \wedge \mathrm{wr}) \vee (\neg \mathrm{wr} \wedge \mathbf{Y}\mathrm{Written})) \Rightarrow \mathrm{out}\ \mathbf{is}\ u$$

An equivalent property in vector GTL is:

$$\begin{array}{lcl} \mathrm{Write} & = & \mathrm{in}\ \mathbf{is}\ u \wedge \mathrm{wr} \\ \mathrm{Written} & = & \mathrm{Write} \vee (\neg \mathrm{wr} \wedge \mathbf{Y}(\mathrm{Written})) \\ \mathrm{Reading} & = & \mathsf{rd} \wedge \mathrm{Written} \\ \\ \mathrm{Reading} & \Rightarrow & \mathrm{out}\ \mathbf{is}\ u \end{array}$$

Notice that the vector GTL representation uses the same recursion variables as regular GTL to break up the monolithic syntax graph of the original GTL property. Each of these fixed-point variables represents the simulation state that is described by the GTL expression on the right-hand side of the equation.

Vector GTL has several advantages over standard GTL properties. One is that fixed-point variables can be used as top-level handles to reference particular simulation states. This is useful for simulation introspection and transformation, as well as for generally improving legibility. Another advantage is that the notation allows for sharing, which can be used for common sub-expression elimination. Not only does this make the specifications more succinct, but it can be used by the simulator to avoid duplicate work. In a similar vein, vector GTL also allows us to specify multiple consequent checks at various states within the simulation. In addition to these benefits, this form is likely to be more familiar to verification engineers than μ-expressions.

Although the tabular nature of vector GTL generally aids all aspects of human interaction with simulations, it unfortunately also makes automated reasoning more difficult. Deviating from the linear textual form of traditional logic creates a richer structure that no longer directly fits term-based reasoning systems. For example, applying transformations using pattern matching is now not as easy, as patterns may span multiple variable definitions, and transformations can affect other sub-simulations. Scoping of indexing variables is also more complicated. Despite this, we found that vector GTL is still preferable for real verification efforts, and that these reasoning problems are largely surmountable through the use of explicit recursion variable substitutions.

Vector GTL can also be seen as an adaptable hybrid between assertion graphs and GTL, since states can be optionally explicitly labeled, or else hidden in the syntactical structure of GTL formulas. The degree to which the syntax tree is explicitly broken into named chunks is determined by the requirements of the particular user or algorithm.

A GTL vector property is formally defined as follows:

Definition 6.1.2. *A vector GTL specification $\mathcal{P}$ consists of a set of recursive equations and a set of assertions. For each GTL recursion variable Z there must be exactly one equation of the form $Z = f$, where f is any GTL formula. Since GTL requires that each recursion path passes through a $\mathbf{Y}$ operation, we require that there is no atemporal recursive cycle through the definitions that does not pass through a $\mathbf{Y}$ operator. The assertions are of the form $Z \Rightarrow \mathrm{C}$ where C is a closed formula of GTL.*

Such systems of equations prescribe a unique semantic value to each of the recursion variables, as a result of the Unique Fixed-point Theorem (Theorem 4.1.1). These recursion variable assignments are then used to define what it means for a circuit trace to satisfy the assertions associated with the same vector property:

Definition 6.1.3. *Let $\bar{\rho} : \mathcal{F} \rightarrow (\mathcal{V} \rightarrow S^{+})$ be the unique solution to the fixed point equations in vector specification $\mathcal{P}$. Then a circuit trace $\sigma \in S^{+}$ satisfies assertion $\mathrm{A} \Rightarrow \mathrm{C}$ of $\mathcal{P}$, written $\sigma \models \mathrm{A} \Rightarrow \mathrm{C}$, if, for every index context $\nu \in \mathcal{V}$,*

$$\sigma \in \| \mathrm{A} \|^{\nu}_{\bar{\rho}} \quad \textit{implies} \quad \sigma \in \| \mathrm{C} \|^{\nu}$$

Definition 6.1.4. *A circuit $\mathcal{K}_{\mathbf{C}}$ satisfies the entire property $\mathcal{P}$, written $\mathcal{K}_{\mathbf{C}} \models \mathcal{P}$, if every trace of the circuit satisfies every assertion made by the property.*

Example Consider the vector property given by:

$$\begin{array}{rcl} \mathrm{ZERO} & = & \mathsf{reset} \vee \mathbf{Y}(\mathrm{TWO}) \wedge \neg\mathsf{reset} \\ \mathrm{ONE} & = & \mathbf{Y}(\mathrm{ZERO}) \wedge \neg\mathsf{reset} \\ \mathrm{TWO} & = & \mathbf{Y}(\mathrm{ONE}) \wedge \neg\mathsf{reset} \\ & & \\ \mathrm{ZERO} & \Rightarrow & \mathsf{empty} \end{array}$$

In this property, the dependencies between the recursion variables sets up a simple three-state cycle. The variable names correspond to the number of steps, modulo 3, since a reset has occurred. The assertion $\mathrm{ZERO} \Rightarrow \mathsf{empty}$ therefore requires that empty is high during a reset and every three subsequent steps where reset has not held since.

6.1.2.1 Model Checking

The algorithm for model checking vector GTL differs slightly from that of standard GTL (Section 3.7). Instead of calculating the fixed-point for μ-expressions independently, we calculate a single global vector fixed-point. We will use the same notation as Chapter 3 to express model checking contexts and simulations.

Definition 6.1.5 (Model Checking Vector GTL)**.** *Model checking starts off with the empty recursion variable context σ_0 where $\sigma_0(Z)\langle\nu\rangle = \perp$ for each recursion variable $Z \in \tau$ and index context ν. Model checking steps are then calculated using the following recurrence:*

$$\sigma_{i+1}(Z) \;\; = \;\; \lfloor f_Z \rfloor_{\sigma_i}$$

where f_Z is the GTL simulation associated with variable Z in property $\mathcal{P}$. Since we know that abstract simulation is monotonically increasing over a finite domain, a fixed-point, $\bar{\sigma}$, is eventually reached. Each assertion $\mathrm{A} \Rightarrow \mathrm{C}$ of $\mathcal{P}$ is then verified by:

$$\lfloor \mathrm{A} \rfloor_{\bar{\sigma}} \subseteq^{\sharp} \lfloor \mathrm{C} \rfloor \quad \textit{for every } \nu \in \mathcal{V}$$

We will say that vector GTL model checking succeeds, written $\mathbf{MC}_{vec}(\mathcal{K}_{\mathbf{C}}, \mathcal{P})$ if every assertion check holds.

Recall from Definition 3.6.1 that the image of a formula, $\mathrm{im}_{\bar{\rho}}(f)$ is the set of end-states of circuit traces that satisfy f in context $\bar{\rho}$. We will show that model checking is sound by linking the simulated abstract state with the image of the antecedent.

Lemma 6.1.6. *The fixed-point recursion context $\bar{\sigma}(Z)$ defines a sound approximation of the set of circuit traces that are satisfied by Z, $\mathrm{im}_{\bar{\rho}}(Z)$.*

Proof. Let the trace semantics fixed-point approximants, ρ_i, be defined by:

$$\begin{aligned} \rho_0(Z)\langle\nu\rangle &:= \emptyset \\ \rho_{i+1}(Z)\langle\nu\rangle &:= \parallel f_Z \parallel^{\nu}_{\rho_i} \end{aligned}$$

Lemma 3.6.7 states that set-based simulation is a sound approximation of the image of a formula:

$$(\forall Z \in \mathcal{F} \;.\; \mathrm{im}_{\rho}(Z) \subseteq^{\forall\mathcal{V}} [\, Z \,]_{\tau}) \;\text{ implies }\; \mathrm{im}_{\rho}(f) \subseteq^{\forall\mathcal{V}} [\, f \,]_{\tau}$$

Furthermore, Lemma 3.7.12 has stated that abstract simulation is always an upper-approximation of set-based simulation:

$$(\forall Z \in \mathcal{F} \;.\; [\, Z \,]^{\nu}_{\tau} \subseteq \gamma(\lfloor\, Z \,\rfloor^{\nu}_{\sigma})) \;\text{ implies }\; [\, f \,]^{\nu}_{\tau} \subseteq \gamma(\lfloor\, f \,\rfloor^{\nu}_{\sigma})$$

Combining these two Lemmas we therefore have:

$$(\forall Z \in \mathcal{F} \;.\; \mathrm{im}_{\rho}(Z) \subseteq^{\forall\mathcal{V}} \gamma(\lfloor\, Z \,\rfloor_{\sigma})) \;\text{ implies }\; \mathrm{im}_{\rho}(f) \subseteq^{\forall\mathcal{V}} \gamma(\lfloor\, f \,\rfloor_{\sigma}) \tag{6.1}$$

We aim to show by induction that the image of each trace approximant is approximated by the corresponding abstract simulation approximant:

$$\mathrm{im}_{\rho_i}(Z) \;\subseteq^{\forall\mathcal{V}}\; \gamma(\sigma_i(Z))$$

The base case is trivially satisfied since $\mathrm{im}_{\rho_0}(Z) = \emptyset$. Let us assume the ith case holds. Then (6.1) gives

$$\mathrm{im}_{\rho_i}(f_Z) \;\subseteq^{\forall\mathcal{V}}\; \gamma(\lfloor\, f_Z \,\rfloor_{\sigma_i})$$

which is exactly equivalent to:

$$\mathrm{im}_{\rho_{i+1}}(Z) \;\subseteq^{\forall\mathcal{V}}\; \gamma(\sigma_{i+1}(Z))$$

Now, taking the limit of both approximants, we have that

$$\bigcup_i^{\mathcal{V}} \mathrm{im}_{\rho_i}(Z) \;\subseteq^{\forall\mathcal{V}}\; \bigcup_i^{\mathcal{V}} \gamma(\sigma_i(Z))$$

and so model checking is a sound upper-approximation of the semantics:

$$\mathrm{im}_{\bar{\rho}}(Z) \;\subseteq^{\forall\mathcal{V}}\; \gamma(\bar{\sigma}(Z))$$

□

Theorem 6.1.7. *If model checking of vector GTL succeeds, and every assertion consequent is atemporal and does not contain disjunction, then the circuit satisfies the vector GTL property:*

$$\mathbf{MC}_{vec}(\mathcal{K}_{\mathbf{C}}, \mathcal{P}) \quad \textit{implies} \quad \mathcal{K}_{\mathbf{C}} \models \mathcal{P}$$

Proof. Let $\mathrm{A} \Rightarrow \mathrm{C}$ be any assertion in property $\mathcal{P}$. Since model checking succeeds, $\lfloor \mathrm{A} \rfloor^{\nu}_{\bar{\sigma}} \subseteq^{\sharp} \lfloor \mathrm{C} \rfloor^{\nu}$ for every $\nu \in \mathcal{V}$, and so by Lemma 3.7.8:

$$\gamma(\lfloor \mathrm{A} \rfloor^{\nu}_{\bar{\sigma}}) \quad \subseteq \quad \gamma(\lfloor \mathrm{C} \rfloor^{\nu}) \tag{6.2}$$

Now by Lemma 3.7.13, $\gamma(\lfloor \mathrm{C} \rfloor^{\nu}) = [\mathrm{C}]^{\nu} = \mathrm{im}(\mathrm{C})$. By the monotonicity of abstract simulation and Lemma 6.1.6, we have $\mathrm{im}_{\bar{\rho}}(\mathrm{A}) \subseteq^{\forall \mathcal{V}} \gamma(\lfloor \mathrm{A} \rfloor^{\nu}_{\bar{\sigma}})$. Therefore, combining these two results with (6.2), we have that:

$$\mathrm{im}_{\bar{\rho}}(\mathrm{A}) \quad \subseteq^{\forall \mathcal{V}} \quad \mathrm{im}(\mathrm{C})$$

So any circuit trace in $\parallel \mathrm{A} \parallel^{\nu}_{\bar{\rho}}$ must also be in $\parallel \mathrm{C} \parallel^{\nu}$, meeting the requirements of Definition 6.1.5. □

6.2 Simulation Structure Generation

An assertion program describes a high-level specification model, and a series of assertions of the form $\mathrm{A} \Rightarrow \mathrm{C}$ that each describe the expected circuit response C when a given antecedent A holds of the model state. The aim of simulation generation is to create a simulation pattern consisting of all possible input sequences that can bring about a given antecedent condition. Using simulation, it can then be checked that the circuit state satisfies the associated consequent C, which in turn implies that the assertion holds.

Our simulation generation approach rewrites antecedent predicates to obtain a description of the required input traces. Each rewriting step uses substitutions to calculate the *weakest preconditions* for the antecedent to be satisfied following one step of execution. In a similar manner to [Dij76], the substitutions are shaped by the program assignments themselves. By repeating this rewriting procedure, and merging equivalent states, we effectively perform a symbolic backwards traversal of the program's state-space.

During this backwards rewriting process, choices about how to arrange the simulation formulas can affect the shape of the resulting simulation. In particular, many of the rules already explored in Chapter 4 can be applied at any stage of the simulation generation process to change the resulting size of the simulation and/or level of simulation abstraction. For this reason, we make use of an interactive simulation generation environment, where named rules can be called on to transform the simulation in different ways. There is also a composite rule that generates simulations automatically, which is generally useful as a starting point when no manual control is yet required.

6.2.1 Simulation Goals

We use *check predicates* to define the top-level verification goals in our environment. These assert the relationship between the circuit, the program and the vector GTL property to be verified. Following common theorem proving methodology [Mil72], we keep a list of goals, made up of such assertions, that can be rewritten, discharged, or decomposed into further goals.

Definition 6.2.1. *The predicate "*$\mathrm{CHECK}\ \mathcal{K}_{\mathbf{C}}\ \mathbf{AP}\ \mathrm{sim}\ (\mathrm{A} \Rightarrow \mathrm{C})$*" holds for circuit* $\mathcal{K}_{\mathbf{C}}$*, assertion program* $\mathbf{AP}$*, vector GTL equations* sim *and GTL property* $\mathrm{A} \Rightarrow \mathrm{C}$*, if for every circuit trace* $\sigma \in \mathrm{tr}(\mathcal{K}_{\mathbf{C}})$ *and every indexing context* $\nu : \mathsf{IndexVars} \rightarrow \mathsf{Values}$*, if every monitor trace* π *of* σ *satisfies antecedent* A *in the states* $\bar{\rho}$ *given by the fixed-point of the GTL equations* sim*, then* σ *also satisfies the consequent* C*:*

$$(\forall \pi \in \mathrm{monitor}_{\mathbf{AP}}(\sigma)\ .\ \pi \in \parallel \mathrm{A} \parallel^{\nu}_{\bar{\rho}}) \quad \textit{implies} \quad \sigma \in \parallel \mathrm{C} \parallel^{\nu}$$

6.2.2 Initialization

For a given circuit $\mathcal{K}_{\mathbf{C}}$ and program $\mathbf{AP}$, simulation generation starts off with *initial checks* for each assertion. These predicates set-up a goal, using the program to be verified and the initially empty vector GTL simulation.

Lemma 6.2.2. *The circuit* $\mathcal{K}_{\mathbf{C}}$ *satisfies program* $\mathbf{AP}$ *consisting of assertions* $\mathrm{A}_i \Rightarrow \mathrm{C}_i$ *exactly when following predicate holds for each* i*:*

$$\mathrm{CHECK}\ \mathcal{K}_{\mathbf{C}}\ \mathbf{AP}\ ()\ (\mathrm{eval}(\mathrm{A}_i) \Rightarrow \mathrm{C}_i)$$

Proof. When the simulation is empty, it imposes no constraints, so the meaning of the assert predicate for each assertion simplifies to:

$$(\forall \pi \in \mathrm{monitor}_{\mathbf{AP}}(\sigma)\ .\ \pi \in \parallel \mathrm{eval}(\mathrm{A}) \parallel^{\nu}) \quad \text{implies} \quad \sigma \in \parallel \mathrm{C} \parallel^{\nu}$$

which by Definition 6.1.1 is equivalent to

$$\Big(\forall \pi \in \mathrm{monitor}_{\mathbf{AP}}(\sigma)\ .\ (\mathrm{A} \xrightarrow[\nu]{\pi} \mathrm{true}) \Big) \quad \text{implies} \quad (\mathrm{C} \xrightarrow[\nu]{\sigma} \mathrm{true})$$

This is exactly the condition required by Definition 5.3.5 for the circuit to satisfy each assertion made in an assertion program. □

A repeated series of steps can now be performed to rewrite this assertion into a form where the antecedent is defined by a simulation purely in terms of circuit inputs, and independent of the program state. When this stage is reached, symbolic ternary simulation can be used for verification.

6.2.3 Weakest Precondition Rewriting

We use a derivative of Dijkstra's weakest precondition transformer [Dij76], to rewrite assertion antecedents in terms of constraints on state further back in time. We describe the transform with the map $\mathbf{wp}$, which takes a program and a Boolean postcondition expression, and calculates the weakest precondition. The process can also be seen as the selective application of a structured set of rewriting rules described by the program.

In the simplest instance, the weakest precondition of a single assignment statement $\mathrm{x} := E$ can be calculated via the single substitution of E for each free variable x in the antecedent condition:

$$\mathbf{wp}\ \ (\mathrm{x} := E)\ \ E' \quad = \quad E'[E/\mathrm{x}]$$

The weakest precondition of a parallel composition is determined by the fair fixed-point application of the weakest precondition calculation for both arguments:

$$\mathbf{wp}\ \ (T_1 \parallel T_2)\ \ E \quad = \quad (\mathit{fix}\ ((\mathbf{wp}\ \ T_1) \circ (\mathbf{wp}\ \ T_2)))\ E$$

The calculation can be guaranteed to terminate with the non-cyclical dependency assumption for programs in Definition 5.3.1 through the use of a marking procedure. The weakest preconditions of conditional statements can be calculated as conditional substitution:

$$\mathbf{wp}\ \ (\mathbf{if}\ E\ \ \mathbf{then}\ T_1\ \mathbf{else}\ T_2)\ \ E' \quad = \quad E \ \rightarrow\ (\mathbf{wp}\ \ T_1\ \ E') \,|\, (\mathbf{wp}\ \ T_2\ \ E')$$

The second and third constraints given in Definition 5.3.1 ensure that this calculation rewrites the condition as far as possible.

6.2.3.1 Rewriting Arrays

The simplest approach for rewriting with array assignment is to introduce a conditional expression to check the array index:

$$\mathbf{wp}\ \ (\mathrm{x}[E] := G)\ \ (\mathrm{x}[F]) \quad = \quad (\,(E = F)\ \rightarrow\ G \,|\, (\mathrm{x}[F])\,)$$

The simplest approach for rewriting indexed **for** statements is to apply one statement for each variable valuation:

$$\mathbf{wp}\ \ (\mathbf{for}\, i : t\, \mathbf{do}\, T)\ \ E \quad = \quad \mathbf{wp}\ \ (\ \|_{\ v \in \mathrm{values}(t)}\ T[v/i])\ \ E$$

But these approaches can lead to some excessively large terms when simultaneous array update is indexed with a **for** statement. This is because a new conditional expression is generated for every value in the array index.

Since this pattern occurs commonly, especially when dealing with memories, we provide a more efficient way of rewriting these cases. First we rewrite the statement so that only a single indexing variable appears in the indexing expression of each array. Where more complicated index expressions occur, a conditional can take their place.

We then choose to rewrite **for** statements using a syntactical analysis of the term being rewritten. We calculate the weakest precondition of the parallel composition only for each index value that can actually affect the source condition being rewritten. This is given by:

$$\mathbf{wp}\ \ (\mathbf{for}\, i : t\, \mathbf{do}\, T)\ \ E \quad = \quad \mathbf{wp}\ \ (\ \|_{\ v \in \mathrm{matches}(i,T,E)}\ T[v/i])\ \ E$$

where $\mathrm{matches}(i, T, E)$ provides the possible symbolic values of i such that statement T assigns to a value referenced in E. For example,

$$\begin{aligned} & \mathbf{wp}\ \ (\mathbf{for}\ i : t\ \mathbf{do}\ a[i] := \mathrm{true})\ \ (a[0] \wedge a[j]) \\ = \ & \mathbf{wp}\ \ (a[0] := \mathrm{true}\ \ \|\ \ a[j] := \mathrm{true})\ \ (a[0] \wedge a[j]) \end{aligned}$$

Assignment to an array expression is then modified so that there is a direct substitution when there is an exact match:

$$\mathbf{wp}\ \ (\mathrm{x}[E] := G)\ \ (\mathrm{x}[E]) \quad = \quad G$$

6.2.3.2 Example

The following statement sets the two-dimensional array a to an identity matrix:

```
for $i$ : int(100) do
  for $j$ : int(100) do
    if $i \neq j$ then $a[i][j] := 0$
    else $a[i][j] := 1$
```

Now suppose we would like to find the weakest precondition for which $a[k][5] = 1$. Using the first approach for array assignment and for statements would generate 10,000 conditional expressions. By instead doing analysis on the source condition, we can calculate the result as:

$$
\begin{array}{rll}
& \mathbf{wp} \left(\begin{array}{l} \textbf{for } i : \mathbf{int}(100) \textbf{ do} \\ \quad \textbf{for } j : \mathbf{int}(100) \textbf{ do} \\ \quad\quad \textbf{if } i \neq j \textbf{ then } a[i][j] := 0 \\ \quad\quad \textbf{else } a[i][j] := 1 \end{array} \right) & (a[k][5] = 1) \\
= & \mathbf{wp} \left(\begin{array}{l} \textbf{for } j : \mathbf{int}(100) \textbf{ do} \\ \quad \textbf{if } k \neq j \textbf{ then } a[k][j] := 0 \\ \quad \textbf{else } a[k][j] := 1 \end{array} \right) & (a[k][5] = 1) \\
= & \mathbf{wp} \left(\begin{array}{l} \textbf{if } k \neq 5 \textbf{ then } a[k][5] := 0 \\ \textbf{else } a[k][5] := 1 \end{array} \right) & (a[k][5] = 1) \\
= & k \neq 5 \rightarrow & \begin{array}{l} (\mathbf{wp}\ (a[k][5] := 0)\ (a[k][5] = 1)) \\ \mid (\mathbf{wp}\ (a[k][5] := 1)\ (a[k][5] = 1)) \end{array} \\
= & k \neq 5 \ \rightarrow & (0 = 1) \mid (1 = 1) \\
= & k \neq 5 \ \rightarrow & \mathrm{false} \mid \mathrm{true} \\
= & (k = 5) &
\end{array}
$$

6.2.4 Detecting Fixed-points

In the previous example, there are no recursive dependencies between program states. As a result, the weakest precondition rewriting approach is guaranteed to terminate. When temporal recursive dependencies are introduced, however, rewriting can expand a condition indefinitely. For example, consider the following statement expansion:

$$
\mathrm{flip} \quad := \quad \left(\begin{array}{l} \textbf{if } \mathsf{reset} \textbf{ then} \\ \quad \mathrm{b} := \mathrm{true} \\ \textbf{else} \\ \quad \mathrm{b} := \mathbf{last}(\neg \mathrm{b}) \end{array} \right)
$$

$$
\begin{array}{rl}
& \mathbf{wp}\ \mathrm{flip}\ \mathrm{b} \\
= & \mathbf{wp}\ \mathrm{flip}\ (\mathsf{reset} \vee \mathbf{Y}(\neg \mathrm{b})) \\
= & \mathbf{wp}\ \mathrm{flip}\ (\mathsf{reset} \vee \mathbf{Y}(\neg \mathsf{reset} \wedge \mathbf{Y}(\mathrm{b}))) \\
= & \mathbf{wp}\ \mathrm{flip}\ (\mathsf{reset} \vee \mathbf{Y}(\neg \mathsf{reset} \wedge \mathbf{Y}(\mathsf{reset} \vee \mathbf{Y}(\neg \mathrm{b})))) \\
= & \mathbf{wp}\ \mathrm{flip}\ (\mathsf{reset} \vee \mathbf{Y}(\neg \mathsf{reset} \wedge \mathbf{Y}(\mathsf{reset} \vee \mathbf{Y}(\neg (\ldots)))))
\end{array}
$$

In this instance, we would like to be able to generate the fixed-point simulation:

$$\mu Z \,.\, \mathsf{reset} \lor \mathbf{Y}(\neg \mathsf{reset} \land \mathbf{Y}(Z))$$

Our strategy for doing this is to keep track of a set of terms that are already having, or have already had, their weakest precondition calculated. Once the calculation reaches a term that it has already seen, then an explicit fixed-point is created.

$$\begin{aligned}
&\mathbf{wp}\ \text{flip}\ \boxed{\text{eval(b)}}\\
=\,&\mathbf{wp}\ \text{flip}\ (\mathsf{reset} \lor \mathbf{Y}(\text{eval}(\neg\text{b})))\\
=\,&\mathbf{wp}\ \text{flip}\ (\mathsf{reset} \lor \mathbf{Y}(\neg\mathsf{reset} \land \mathbf{Y}(\boxed{\text{eval(b)}})))\\
=\,&\mu Z \,.\, \mathsf{reset} \lor \mathbf{Y}(\neg \mathsf{reset} \land \mathbf{Y}(Z))
\end{aligned}$$

To perform this calculation in practice, we label each equation of a vector GTL simulation with the original predicate from which this segment of the simulation was derived. This labeling also provides a valuable means of debugging in the case of over-abstraction during simulation, since it shows how each of the simulation states are associated with program predicates.

In this labeled form of vector GTL, each equation is a triplet (Z, E, E') where Z is the recursion variable used to name this state, E is the program predicate that this state correspond to, and E' is the GTL simulation necessary to produce this state.

Rather than create a new simulation state for each simulation step, we only create a new state for each time-step that passes in the simulation. We adapt the re-writing algorithm so that substitution only occurs on terms *not enclosed by* $\mathbf{Y}$. When this rewriting is complete, we create new simulation states from those terms enclosed by $\mathbf{Y}$. If there is already a simulation state that matches the term, then this state is referenced instead.

Using this approach, the flip example expands as follows. First, we start off with the initial simulation state named X_0 for condition eval(b):

$$(\quad X_0, \quad \text{eval(b)}, \quad \text{eval(b)} \quad)$$

Now we rewrite the simulation using the `WP` rule on current-time expressions to:

$$(\quad X_0, \quad \text{eval(b)}, \quad \mathsf{reset} \lor \mathbf{Y}(\text{eval}(\neg\text{b})) \quad)$$

Since the expression $\text{eval}(\neg\text{b})$ refers to the previous time-step, we do not expand it further. We now ‘split’ the simulation to form a new simulation state from the subterm of the simulation that refers to the preceding time-frame. This is achieved by creating a new state for this term, with a fresh recursion variable, and substituting the term in the original simulation for this variable.

$$\begin{array}{lcccr} (& X_0, & \mathrm{eval(b)}, & \mathsf{reset} \vee \mathbf{Y} X_1 &) \\ (& X_1, & \mathrm{eval}(\neg\mathrm{b}), & \mathrm{eval}(\neg\mathrm{b}) &) \end{array}$$

Again we rewrite the second simulation state using the `WP` rule:

$$\begin{array}{lcccr} (& X_0, & \mathrm{eval(b)}, & \mathsf{reset} \vee \mathbf{Y} X_1 &) \\ (& X_1, & \mathrm{eval}(\neg\mathrm{b}), & \neg\mathsf{reset} \wedge \mathbf{Y}(\mathrm{eval(b)}) &) \end{array}$$

Now when we come to split the term $\mathrm{eval(b)}$, we notice that this condition has already been expanded in simulation state X_0. Therefore after splitting the state we can apply a rule that checks for equal states and merges them. This ties the fixed-point knot:

$$\begin{array}{lcccr} (& X_0, & \mathrm{eval(b)}, & \mathsf{reset} \vee \mathbf{Y} X_1 &) \\ (& X_1, & \mathrm{eval}(\neg\mathrm{b}), & \neg\mathsf{reset} \wedge \mathbf{Y} X_0 &) \end{array}$$

Now the simulation is completely generated, as no program state remains in the simulation description. Notice that our approach ensures that fixed-point definitions, corresponding to dependency cycles through simulation states, always pass through one unit of time, $\mathbf{Y}$, and hence are well-defined.

6.2.4.1 Boolean Encodings

In most cases, the simple approach of checking for syntactic equivalence is not sufficient for termination. Instead, we can enhance our algorithm by creating Boolean functions to characterize our states, and check for equivalence using standard techniques such as BDD equivalence or SAT solving. Since our program types are all bounded, and expressions all refer to a bounded temporal depth, such checks are decidable.

Formulas of GTL that do not contain fixed-points can be encoded into a single Boolean predicate that characterizes the linear traces that satisfy them. GTL formulas are represented as single Boolean predicates containing variables that encode the combined circuit and program traces. For instance, $\mathsf{n} \wedge \mathbf{Y}(\neg\mathsf{n})$ is encoded as the predicate $n_0 \wedge \neg n_1$.

We encode program expressions into Boolean vectors that describe their evaluation, and an extra Boolean predicate that captures whether the expression fully evaluates or not. For instance, consider a 3-bit integer literal, encoded with:

$$\mathrm{encode}(3 : \mathbf{int}(8)) \;=\; (\langle F, T, T\rangle, T)$$

The first component of this pair is the binary encoding of the integer 3. The second component asserts that this expression always evaluates. Boolean variables are encoded directly:

$$\mathrm{encode}(v : \mathbf{bool}) \;=\; (\langle v\rangle, T)$$

Variables of other types are converted into a vector of suitably-named Boolean variables:

$$\mathrm{encode}(v : \mathbf{int}(8)) \;=\; (\langle v\_\mathit{int8}[2], v\_\mathit{int8}[1], v\_\mathit{int8}[0]\rangle, T)$$

Maps and equality are given bit-vector interpretations, so that complex expressions can be encoded. The application of these encoded maps on the encoding must be isomorphic to the maps on the original value domain. For example:

$$\mathrm{encode}((\{v : \mathbf{int}(8)\} < 4) = b) \;=\; (\langle \neg v\_\mathit{int8}[2] \wedge b \vee v\_\mathit{int8}[2] \wedge \neg b\rangle, T)$$

The importance of the second component of the encoding comes into play when partial functions are applied. For example, the expression that encodes the head of an empty list is encoded as any value together with false, indicating that it can not be evaluated:

$$\mathrm{encode}(\mathrm{head}\,[\,]) \;=\; (\langle *\rangle, F)$$

For the expression E that encodes to (a, b), the corresponding proposition $\mathrm{eval}(E)$ encodes to $a \wedge b$, since such propositions must fully evaluate in order to hold.

6.2.5 Simplification

During simulation generation it is useful to apply various simplification rules to decrease the size of the terms involved, and to clarify any required debugging. The rule `SIMPLIFY` attempts to apply various simplification rules at every depth within a term. The core simplification rules are shown in Figure 6.1, and can be extended to cater for additional data types or functions.

6.2.6 Trimming

As a result of these simulation rules, we will sometimes find that orphaned simulation states become disconnected from the main simulation. The rule named `TRIM` finds the set of simulation recursion variables that the antecedent checks depend on, and removes from the simulation any variables that are not required. A similar rule, `ELIM_FALSE` removes those states whose Boolean encoding is false. Variables that refer to such states in other parts of the simulation are replaced with $\mathbb{ff}$.

$$\begin{array}{rcl@{\qquad}rcl}
\neg\neg a &=& a & a &=& \text{true if } \mathit{encode}\ a = (\langle T\rangle, T) \\
a \wedge \text{false} &=& \text{false} & & & \text{for atemporal } a \\
\text{false} \wedge a &=& \text{false} & a &=& \text{false if } \mathit{encode}\ a = (\langle F\rangle, T) \\
a \vee \text{true} &=& \text{true} & (\mathbf{last}(a))[\mathbf{last}(e)] &=& \mathbf{last}(a[e]) \\
\text{true} \vee a &=& \text{true} & f(a \rightarrow b \,|\, c) &=& a \rightarrow f(b) \,|\, f(c) \\
a \wedge \text{true} &=& a & f(a \rightarrow b \,|\, c, d) &=& a \rightarrow f(b, d) \,|\, f(c, d) \\
\text{true} \wedge a &=& a & f(d, a \rightarrow b \,|\, c) &=& a \rightarrow f(d, b) \,|\, f(d, c) \\
a \vee \text{false} &=& a & f(\mathbf{last}(a)) &=& \mathbf{last}(f(a)) \\
\text{false} \vee a &=& a & f(a, \mathbf{last}(b)) &=& \mathbf{last}(f(a, b)) \text{ if } b \text{ is constant} \\
\neg\text{true} &=& \text{false} & f(\mathbf{last}(a), b) &=& \mathbf{last}(f(a, b)) \text{ if } a \text{ is constant} \\
\neg\text{false} &=& \text{true} & f(\mathbf{last}(a), \mathbf{last}(b)) &=& \mathbf{last}(f(a, b)) \\
\neg\mathbf{last}(a) &=& \mathbf{last}(\neg a) & \neg(\exists i.a) &=& \forall i.\neg a \\
\neg(a \wedge b) &=& \neg a \vee \neg b & \neg(\forall i.a) &=& \exists i.\neg a \\
\neg(a \vee b) &=& \neg a \wedge \neg b & c \rightarrow E \,|\, F &=& (c \wedge E) \vee (\neg c \wedge F) \\
 & & & & & \text{for Boolean } E,F
\end{array}$$

Figure 6.1: Simplification Rules

6.2.7 Parameterization

Although weakest precondition rewriting removes all the program variables via substitution, it is sometimes not possible to directly simulate the resulting terms because they still contain higher-level constructs such as arithmetic operations. Simulation requires that we know the symbolic ternary value that should be used to drive each circuit node involved in such expressions. Deducing this information requires additional analysis. For example, consider how the vector din should be simulated to achieve the condition $\text{int2bvn}(i >> 1) = \langle \text{din}[2], \text{din}[1], \text{din}[0]\rangle$. We need to rewrite such a predicate into a form that determines the possible ranges and interdependencies between the nodes involved.

Fortunately, this problem has already received attention and is part of general STE verification methodology. The act of expressing a predicate in terms of the possible range of values of several variables within it is known as *parametrization*, and can be achieved using the `param` algorithm of [AJS99]. The `param` algorithm receives a set of variables and a target predicate to parameterize, and returns a new predicate for each variable, defining the possible range of that variable under which the original predicate is satisfied. For example,

$$\begin{array}{l}
\texttt{param}\ \ \{\text{din}[2], \text{din}[1], \text{din}[0]\} \\
\qquad (\text{encode}(\text{int2bvn}(i >> 1) = \langle \text{din}[2], \text{din}[1], \text{din}[0]\rangle)
\end{array}$$

returns:

$$\mathrm{din}[2] \;=\; \mathrm{false}, \quad \mathrm{din}[1] \;=\; i\_int8[2], \quad \mathrm{din}[0] \;=\; i\_int8[1]$$

This can be used to directly build the GTL formula

$$\neg\mathrm{din}[2] \wedge (\mathrm{din}[1] \textbf{ is } i\_int8[2]) \wedge (\mathrm{din}[0] \textbf{ is } i\_int8[1])$$

which can then be simulated.

6.2.8 Model Checking

Once a simulation structure has been completely generated, after a couple of adaptations it can be used to drive a simulation run that checks the property.

6.2.8.1 Temporal Mapping

Assertion programs will typically be cycle-accurate descriptions of the required circuit characteristics. In contrast, each step of STE-based simulation normally corresponds to a phase of the fastest clock in the simulation. Therefore some extra temporal transformation is normally required to translate a simulation to the circuit's internal timing.

In practice, many different clocking schemes may be used to match the particular characteristics and requirements of the device. We will consider a common case, where a cycle is composed of a *high* and a *low* phase, and the circuit is considered stable at the rising-edge of the cycle. We will also assume that the clock is distinguished by the logical value of node clk.

The rule `RETIME` rewrites a cycle-accurate simulation description for this timing model. The rule first doubles the temporal delay between asserted antecedent values by replacing each $\mathbf{Y}f$ with $\mathbf{YY}f$. It then adds alternating assertions about the value of node clk. For example, the cycle accurate simulation $\mathsf{a} \wedge \mathbf{Y}\mathsf{b}$ is replaced by the phase accurate simulation:

$$\mathsf{a} \wedge \mathbf{Y}(\mathbf{Y}(\mathsf{b} \wedge \neg\mathsf{clk}) \wedge \mathsf{clk}) \wedge \neg\mathsf{clk}$$

This mapping is illustrated in Figure 6.2. The top arrows illustrate the point in the circuit timing that the program steps are mapped to. These are in alignment with the end of the last clock phase in each cycle. As a result, the simulation input values are stable at the times required by the circuit, and the consequent checks expect the circuit to have a reached a stable value at these sample points.

We may now finally run the model checking algorithm for vector GTL by invoking the rule `SIM` on the constructed simulation.

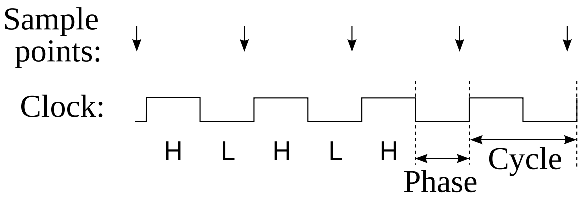


Figure 6.2: Cycle- to Phase-Accurate Temporal Mapping

6.3 Controlling Simulation Generation

In the previous section, we have described a method for the automatic generation of symbolic ternary simulation descriptions. In many cases such simulations succeed with no further intervention. But experience from GSTE shows that the level of control over the simulation approach is of vital importance for verification success. This flexibility can be used both for manual abstraction refinement, in cases of either over- or under-abstraction, as well as for switching between symbolic and explicit forms of simulation.

These types of refinements as well as forms of property decomposition can be achieved using the reasoning rules for GTL that we described in Chapter 4. We will now show how these rules can be applied midway though the simulation generation process to control the characteristics of the resulting simulation run. If desired, the user can choose which rewriting rules should be applied at each generation step, shaping the GTL simulation and its resulting characteristics. The use of the these rules is illustrated in the subsequent chapter, where we examine some particular case study verifications.

6.3.1 Vector GTL Rules

The following two simple rules can be used to manage the way in which simulations are encoded as fixed-point equations.

6.3.1.1 SPLIT

The `SPLIT` rule breaks off sub-terms of a simulation to form a new recursion variable definition. This can be useful for introducing sharing, or to focus the effects of subsequent rule applications, since it introduces a fresh reference name. As an example, `SPLIT` Z $(\mathsf{n} \wedge \mathsf{m})$ transforms

$$Z \;=\; \mathsf{k} \wedge \mathbf{Y}(\mathsf{n} \wedge \mathsf{m}) \qquad \text{into} \qquad \begin{array}{rcl} Z & = & \mathsf{k} \wedge \mathbf{Y} W \\ W & = & \mathsf{n} \wedge \mathsf{m} \end{array}$$

whilst introducing the name W for this formula.

6.3.1.2 SUBSTITUTE

The `SUBSTITUTE` rule is in some ways the dual to the `SPLIT` rule. It substitutes a recursion variable instance with its definition. For example, `SUBSTITUTE` W $(\mathsf{n} \wedge \mathsf{m})$ transforms

$$\begin{array}{rcl} Z & = & \mathsf{k} \wedge \mathbf{Y} W \\ W & = & \mathsf{n} \wedge \mathsf{m} \end{array} \quad \text{into} \quad \begin{array}{rcl} Z & = & \mathsf{k} \wedge \mathbf{Y}(\mathsf{n} \wedge \mathsf{m}) \\ W & = & \mathsf{n} \wedge \mathsf{m} \end{array}$$

Again, this is useful for selectively targeting the effect of further rules. For example, for reasons of abstraction refinement, we may wish to apply a distributive rule. If the terms involved are split over several recursion variables, we can first use the substitution rule before applying standard pattern matching.

6.3.2 Abstraction Refinement Rules

As we have seen in Section 4.3, simple equivalences can be used to rewrite GTL formulas and change the resulting level of simulation abstraction. These give rise to the following abstraction refinement rules for our environment.

6.3.2.1 RAISE_DISJ

Like the GTL rule in Section 4.3.1, this distributes conjunction and $\mathbf{Y}$ over disjunction, throughout the entire simulation. In practice, we have found that it is often useful to routinely apply this rule at each step of every simulation generation, to avoid common cases of over-abstraction.

6.3.2.2 REDISTRIBUTE

Section 4.3.2 has described product reduction, where distributing $\mathbf{Y}$ over conjunction critically determines whether conjunct conditions are simulated together or separately. Simulating different parts of the circuit separately is more efficient, because we do not calculate how the two conditions interact. The resulting state can, however, be too approximate because we do not consider these interactions. The `REDISTRIBUTE` rule allows a user to control these aspects by providing an *abstraction schema* that can be used to determine which types of simulations should occur independently.

We will say that a *subspace* of an assertion program is a set of program variables that we associate with one of its particular aspects. An abstraction schema then consists of a set of subspaces, $\{S_1, S_2, \ldots, S_n\}$. The intention of such a schema

```
if $\mathsf{wr_a}$ then
  $\mathrm{cell_a} := \mathsf{in_a}$
else
  $\mathrm{cell_a} := \mathbf{last}(\mathrm{cell_a})$
if $\mathsf{wr_b}$ then
  $\mathrm{cell_b} := \mathsf{in_b}$
else
  $\mathrm{cell_b} := \mathbf{last}(\mathrm{cell_b})$
```

Figure 6.3: Assertion Program for a Two-Cell Comparator

is that simulations related to different subspaces should take place independently. Given a simulation written as conjuncts in the form $\mathbf{Y}(c_1 \wedge c_2 \wedge \ldots \wedge c_n)$, the rule `REDISTRIBUTE` $\{S_1, S_2, \ldots, S_m\}$ redistributes it to a form $\mathbf{Y}A_1 \wedge \mathbf{Y}A_2 \wedge \ldots \wedge \mathbf{Y}A_m \wedge \mathbf{Y}A$, where each A_i is the conjunction of those c_j that contain a variable from S_i, and A is the conjunction of those c_j that contain no variables from any S_i. The result is that the preconditions associated with these subspaces are then generated and simulated separately.

Example We will show how the `REDISTRIBUTE` rule can be used with reference to the memory cell comparator of Section 2.6. Recall that the assertion graph in Figure 2.18 defines a complete simulation of the cross-product state-space, whereas those in Figure 2.19 simulate each memory cell independently, and then compose each of these simulations. Figure 6.3 shows a suitable program for this verification. We aim to simulate the condition

$$\mathbf{Y}((\mathrm{cell_a} = u) \wedge (\mathrm{cell_b} = v))$$

By default we generate the simulation shown in Figure 6.4 (i), which a similar shape to the assertion graph in Figure 2.18. Noting, however, that the two memory cells operate independently, we can apply `REDISTRIBUTE` $\{\{\mathrm{cell_a}\}, \{\mathrm{cell_b}\}\}$ as the first step in simulation generation to obtain the split condition:

$$\mathbf{Y}(\mathrm{cell_a} = u) \wedge \mathbf{Y}(\mathrm{cell_b} = v)$$

This results in the simulation in Figure 6.4 (ii), which a similar shape to the assertion graph in Figure 2.19, and simulates the two components separately.

$$\begin{aligned}
\mathrm{S}_0 &= \mathbf{Y}((\mathrm{cell}_\mathrm{a} = u) \wedge (\mathrm{cell}_\mathrm{b} = v)) \\
&= \mathbf{Y}\mathrm{S}_1 \\[1em]
\mathrm{S}_1 &= (\mathrm{cell}_\mathrm{a} = u) \wedge (\mathrm{cell}_\mathrm{b} = v) \\
&= (\mathsf{wr}_\mathsf{a} \wedge \mathsf{wr}_\mathsf{b} \wedge \mathsf{in}_\mathsf{a}\ \mathbf{is}\ u \wedge \mathsf{in}_\mathsf{b}\ \mathbf{is}\ v) \\
&\quad \vee(\neg\mathsf{wr}_\mathsf{a} \wedge \neg\mathsf{wr}_\mathsf{b} \wedge \mathbf{Y}\mathrm{S}_1) \\
&\quad \vee(\mathsf{wr}_\mathsf{a} \wedge \neg\mathsf{wr}_\mathsf{b} \wedge \mathsf{in}_\mathsf{a}\ \mathbf{is}\ u \wedge \mathbf{Y}\mathrm{S}_2) \\
&\quad \vee(\neg\mathsf{wr}_\mathsf{a} \wedge \mathsf{wr}_\mathsf{b} \wedge \mathsf{in}_\mathsf{b}\ \mathbf{is}\ v \wedge \mathbf{Y}\mathrm{S}_3) \\[1em]
\mathrm{S}_2 &= (\mathrm{cell}_\mathrm{b} = v) \\
&= (\mathsf{wr}_\mathsf{b} \wedge \mathsf{in}_\mathsf{b}\ \mathbf{is}\ v) \vee (\neg\mathsf{wr}_\mathsf{b} \wedge \mathbf{Y}\mathrm{S}_2) \\[1em]
\mathrm{S}_3 &= (\mathrm{cell}_\mathrm{a} = u) \\
&= (\mathsf{wr}_\mathsf{a} \wedge \mathsf{in}_\mathsf{a}\ \mathbf{is}\ u) \vee (\neg\mathsf{wr}_\mathsf{a} \wedge \mathbf{Y}\mathrm{S}_3)
\end{aligned}$$

(i)

$$\begin{aligned}
\mathrm{S}_0 &= \mathbf{Y}(\mathrm{cell}_\mathrm{a} = u) \\
&\quad \wedge\mathbf{Y}(\mathrm{cell}_\mathrm{b} = v) \\
&= \mathbf{Y}\mathrm{S}_1 \wedge \mathbf{Y}\mathrm{S}_2 \\[1em]
\mathrm{S}_1 &= (\mathrm{cell}_\mathrm{b} = v) \\
&= (\mathsf{wr}_\mathsf{b} \wedge \mathsf{in}_\mathsf{b}\ \mathbf{is}\ v) \\
&\quad \vee(\neg\mathsf{wr}_\mathsf{b} \wedge \mathbf{Y}\mathrm{S}_1) \\[1em]
\mathrm{S}_2 &= (\mathrm{cell}_\mathrm{a} = u) \\
&= (\mathsf{wr}_\mathsf{a} \wedge \mathsf{in}_\mathsf{a}\ \mathbf{is}\ u) \\
&\quad \vee(\neg\mathsf{wr}_\mathsf{a} \wedge \mathbf{Y}\mathrm{S}_2)
\end{aligned}$$

(ii)

Figure 6.4: Comparison of Simulation Generation for a Two-Cell Comparator

6.3.2.3 UNROLL

Given a state-space schema with which to redistribute simulation conditions, we can create a composite rule that automatically applies the rewriting and fixed-point detection steps from Section 6.2 in sequence. The rule `UNROLL` s performs the following steps in sequence:

`WP`	Find weakest preconditions, based on the assertion program
`SIMPLIFY`	Perform basic simplification
`RAISE_DISJ`	Simulate disjunctive conditions independently
`REDISTRIBUTE` s	Simulate specified conjuncts independently
`LAST_SPLIT`	Split off formulas referring to previous time-frame
`ELIM_FALSE`	Remove states that are false
`SIMPLIFY`	Perform basic simplification
`MERGE_EQ`	Merge any equivalent states
`TRIM`	Trim unused states

This sequence of rules is sufficient in most cases to produce a symbolic ternary simulation with a reasonable, intermediate, level of default abstraction, and we will illustrate its use in both of the case studies in the subsequent chapter.

6.3.2.4 SPLIT_STATE

Derived from the case-split rule of Section 4.3.3, the rule `SPLIT_STATE` takes the set of mutually exclusive cases $\{C_1, C_2, \ldots C_n\}$, where $(C_1 \ \vee \ C_2 \ \vee \ldots \vee \ C_n)$ is valid, and case-splits term f into

$$(f \wedge C_1) \vee (f \wedge C_2) \vee \ldots \vee (f \wedge C_n)$$

This is typically used to enable each of these cases to be simulated independently, thus raising the level simulation precision.

Example Suppose, as an optimization, a piece of data can take one of two equivalent routes through a pipeline, depending on whether there is currently a bubble behind it or not. Since this is an optimization, it should not feature in the specification. Therefore the simulation that is generated by default will not distinguish the two cases, and the preceding element will be set to X. The verification will fail, because not enough information exists in the simulation to determine which path the data takes. To refine the simulation we can use the `SPLIT_STATE` rule to distinguish the cases of whether there is a bubble behind it or not.

The use of this rule is also illustrated by FIFO case study in Chapter 7.

6.3.2.5 WKN

All the rules so far have been equivalences. But since GTL formulas are monotonic, and always occur negatively in the antecedent, it is also possible to rewrite a simulation by *weakening* a sub-term of the simulation. Such weakening can result in a more approximate simulation, which potentially consumes less space and time. For example, suppose our simulation is $f \wedge g$. From our knowledge of the design, we may be aware that, in actual fact, f alone is sufficient a condition to verify the consequent condition. The weakened simulation might be considerably simpler, since the nodes affected by g may now be simulated with X. The application of the rule ($\texttt{WKN}\ a\ b$) can achieve this effect, by attempting to replace every instance of a with b, under the condition that a implies b.

6.3.3 Symbolic Rules

In this section we document rules in our framework that manage the symbolic aspects of the resulting simulations.

6.3.3.1 CREATE_VARIABLE

The `CREATE_VARIABLE` rule allows us to manipulate preconditions by creating index variables that are subsequently used for symbolic simulation. For example, suppose that we are trying to simulate the condition $\mathrm{a} = \mathbf{last}(\mathrm{b})$ where both a and b are input program variables. This condition asserts equality between the current value of a and the previous value of b. Since simulation progresses one time step at a time, however, we must introduce a variable with which to link the two values. Application of the rule `CREATE_VARIABLE` a creates a fresh variable i and rewrites the condition as:

$$\exists i \,.\, (\mathrm{a} = i) \;\wedge\; (\mathbf{last}(\mathrm{b}) = i)$$

Since constants are independent of time, this is equivalent to:

$$\exists i \,.\, (\mathrm{a} = i) \;\wedge\; \mathbf{Y}(\mathrm{b} = i)$$

If, furthermore, constant i is not used elsewhere in the simulation, then the quantification becomes unnecessary:

$$(\mathrm{a} = i) \;\wedge\; \mathbf{Y}(\mathrm{b} = i)$$

This is sound because all free antecedent variables are implicitly existential, owing to their negative position in the model checking assertions. This condition is now in a suitable form to be directly simulated.

The use of this rule is illustrated within the scheduler case study verification in Chapter 7.

6.3.3.2 SYM_SUBSTITUTE

Since our simulation approach is symbolic, simulating a condition A with free index variable i gives us a distinct set of image states for each valuation of i. Using this symbolic state, we can directly calculate the alternative condition $\mathrm{A}[E/i]$ using simple symbolic substitution.

For example, suppose we are modeling the value held by a counter using the program variable count. Targeting the condition $(\mathrm{count} = i)$ results in simulation S, which creates an family of circuit states whose indices correspond to the different count states. Suppose we are now required to simulate the condition $(\mathrm{count} = j + 1)$. We can use our existing simulation of $(\mathrm{count} = i)$ and perform the symbolic substitution of $j + 1$ for i, written written $S(i := j + 1)$ in GTL.

The rule SYM_SUBSTITUTE i E rule in our framework rewrites condition A into

$$(A[i/E])(i := E)$$

where i is not free in A and E depends only on index variables. By introducing these self-canceling substitutions we can syntactically apply the first during our simulation generation process, leaving the second as a later simulation step.

We illustrate this rule in the subsequent scheduler case study, where it is used to reduce the number of symbolic conditions that we have to simulate for waiting micro-operations.

6.3.4 A Decomposition Rule

Another approach to overcoming the state-explosion problem is to use structural decomposition to split apart the property at hand. Our framework contains rules for many of the simple decomposition rules for GTL from Section 4.2. In this section we describe another approach that splits the program based on an extra internal *decomposition interface*. The decomposition interface is a mapping between program and circuit state, specified in the same manner as the input interface for a program. Unlike the input interface, however, it does not form part of the specification, but exists purely to define a splitting point in the verification. The DECOMPOSE rule performs such a split.

A standard assertion program verification *assumes* that the input map holds, and demonstrates that the output assertions hold. The DECOMPOSE rule splits this process into two. The first subgoal verifies the decomposition interface, given that the input interface as an assumption. The second verifies the original program assertions using both the decomposition and input interfaces as assumptions. Simulation now takes place in two stages: one from the input interface to the decomposition interface, and another from the decomposition interface to the assertions. This is illustrated in Figure 6.5.

In order to apply the decomposition interface as an assumption in the second stage of the verification, we modify the program $\mathbf{AP}$, so that all the assignments to variable x are replaced with the assignment $\mathrm{x} := E$. The result of this modification is denoted $\mathbf{AP} \lhd (\mathrm{x} := E)$. By replacing these assignments, we direct the simulation generation process to drive the simulation using the decomposition interface, wherever possible. For example, DECOMPOSE $(\mathrm{x} := E)$ splits the verification goal

$$\mathrm{CHECK}\ \mathcal{K}_{\mathbf{C}}\ \mathbf{AP}\ \mathrm{sim}\ (\mathrm{A} \Rightarrow \mathrm{C})$$

into

$$\text{CHECK } \mathcal{K}_{\mathbf{C}}\ \mathbf{AP}\ \text{sim}\ ((\text{x} = d) \Rightarrow (E = d))$$

which verifies the decomposition interface, and

$$\text{CHECK } \mathcal{K}_{\mathbf{C}}\ (\mathbf{AP} \lhd (\text{x} := E))\ \text{sim}\ (\text{A} \Rightarrow \text{C})$$

which verifies the original assertion under the assumption that the decomposition interface holds.

Example Suppose a circuit consists of a register bank and an arithmetical unit. At each cycle, either a number can be written to the register with address i_1, or else the product of registers i_1 and i_2 can be calculated and placed on line out. Such functionality might be modeled by the program fragment:

```
if write then
  reg[$\mathsf{i}_1$] := in
else
  result := reg[$\mathsf{i}_1$] × reg[$\mathsf{i}_2$]
```

together with the assertion

$$\textbf{forall } c\ .\ (\text{result} = c) \Rightarrow (\mathsf{out} = c)$$

It is quite possible that simulation of both the memory and the arithmetical unit will together be too large for a single simulation run. We can, however, decompose the task and simulate each of these units separately. First we rewrite the program so that we have fresh names to reference the two arithmetical operands:

```
if write then
  reg[$\mathsf{i}_1$] := in
else
  $\text{op}_\text{A}$ := reg[$\mathsf{i}_1$]
  $\text{op}_\text{B}$ := reg[$\mathsf{i}_2$]
  result := $\text{op}_\text{A}$ × $\text{op}_\text{B}$
```

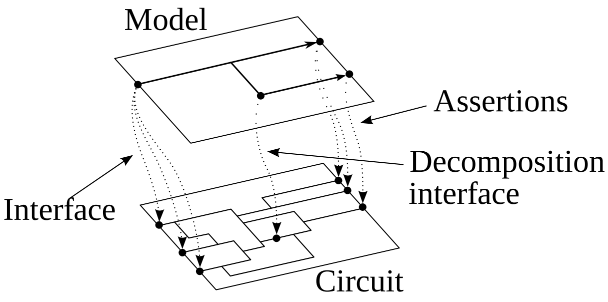


Figure 6.5: Introducing a Decomposition Interface

Now suppose we can find an internal interface mapping that describes how the memory connects to the arithmetical unit, given by $\mathrm{op}_\mathrm{A} := \mathsf{bus}_\mathsf{A}$ and $\mathrm{op}_\mathrm{B} := \mathsf{bus}_\mathsf{B}$. Applying our rule for decomposition twice, we end up with two simulations that verify the register bank and one simulation that verifies the arithmetic.

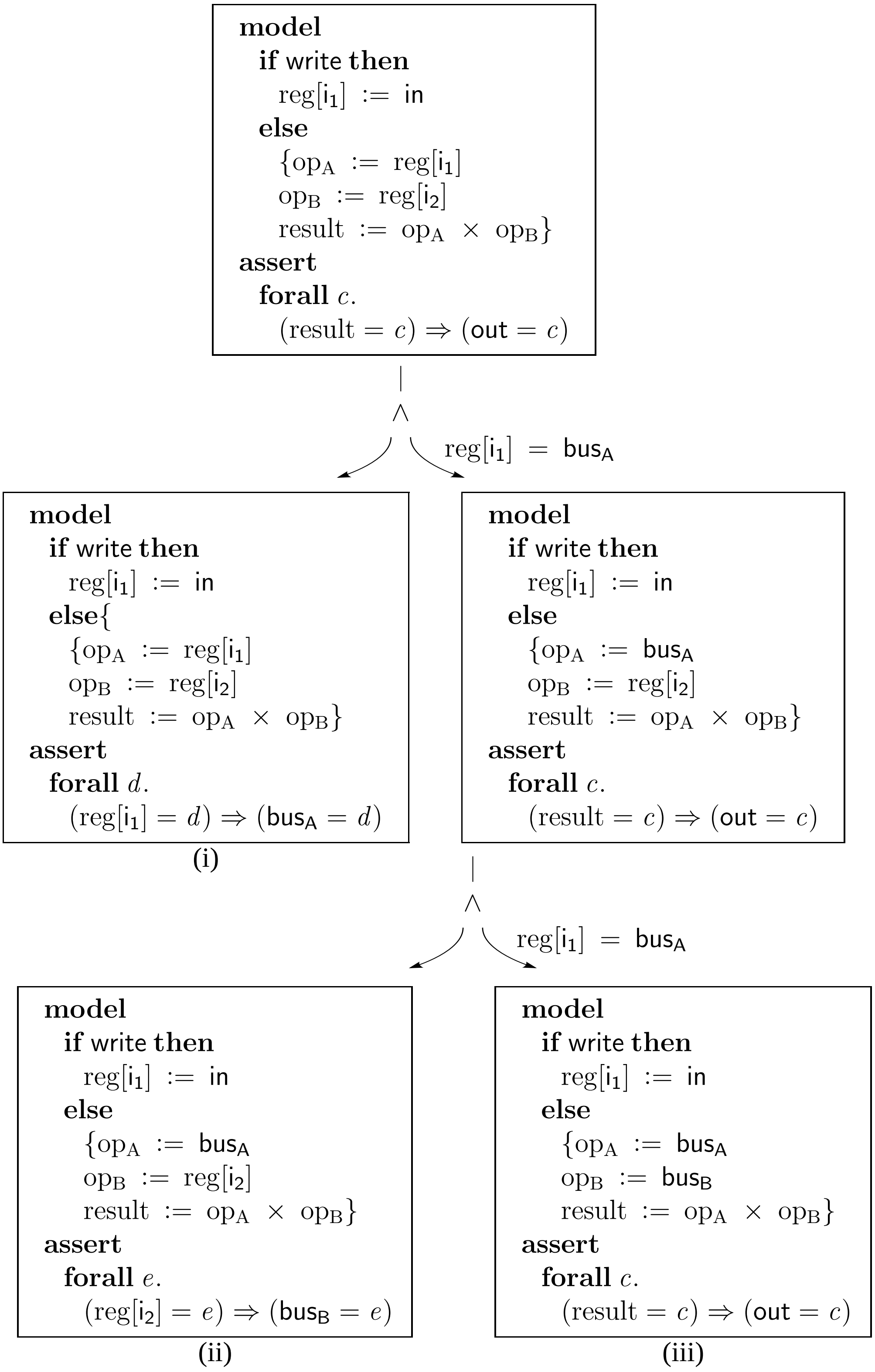
Figure 6.6 shows a proof tree for the two decompositions. The first split introduces the intermediate assertion that the values held on bus_A correspond to the abstract state op_A in the program. The second split similarly relates bus_B to op_B. Figures 6.7(i)-(iii) illustrates the areas of the circuit simulated by the programs labeled (i)-(iii) respectively in Figure 6.6. The solid lines represent areas that are actively simulated in each case.

6.4 Related Work

There are several existing methods for creating symbolic ternary simulations from higher-level descriptions, including, in particular, the work of Joyce and Seger [SJ92], and Jain [Jai97]. These frameworks use simple mappings from their specifications to low-level simulation outlines. In contrast, generating the simulations using our approach requires significantly more reasoning. This is because we have chosen to use a high-level model with abstract state, so generating the simulations requires a fixed-point state traversal calculation rather than a direct mapping.

The core algorithm of our process is the re-writing process, which effectively performs a symbolic backwards traversal of the property state-space. There are several existing similar approaches to exploring the state-spaces of imperative programs. In [SH97] weakest precondition calculations are used within a theorem prover for forward construction of state graphs. In [BFH$^+$92], weakest preconditions are also used to help create minimal transition system representations. Our algorithm differs from these approaches in several interesting ways. First, many algorithms are concerned with creating state-transition graphs, whereas we produce formulas of GTL, which are significantly more complex structures. Second, we do not aim to explore entire state-spaces, but only those abstract states required for each assertion. Third, we do not aim for our abstract states to *partition* the model state-space, since overlapping abstract states are beneficial to the efficiency of ternary simulations.

Like symbolic ternary simulation itself, our algorithm also relies on separating the data and control aspects of a property, in order to tame state explosion. The control aspects are expanded fully into explicit model checking steps, whereas the data aspects can remain symbolic throughout the entire process. This is a particularly relevant for



```
model
  if write then
    reg[i1] := in
  else
    {opA := reg[i1]
    opB := reg[i2]
    result := opA × opB}
assert
  forall c.
    (result = c) ⇒ (out = c)
```

```
model
  if write then
    reg[i1] := in
  else{
    {opA := reg[i1]
    opB := reg[i2]
    result := opA × opB}
assert
  forall d.
    (reg[i1] = d) ⇒ (busA = d)
```

```
model
  if write then
    reg[i1] := in
  else
    {opA := busA
    opB := reg[i2]
    result := opA × opB}
assert
  forall c.
    (result = c) ⇒ (out = c)
```

```
model
  if write then
    reg[i1] := in
  else
    {opA := busA
    opB := reg[i2]
    result := opA × opB}
assert
  forall e.
    (reg[i2] = e) ⇒ (busB = e)
```

```
model
  if write then
    reg[i1] := in
  else
    {opA := busA
    opB := busB
    result := opA × opB}
assert
  forall c.
    (result = c) ⇒ (out = c)
```

∧
$\mathrm{reg}[i_1] = \mathsf{bus}_A$
(i)
∧
$\mathrm{reg}[i_1] = \mathsf{bus}_A$
(ii)
(iii)

Figure 6.6: Decomposition Proof Tree

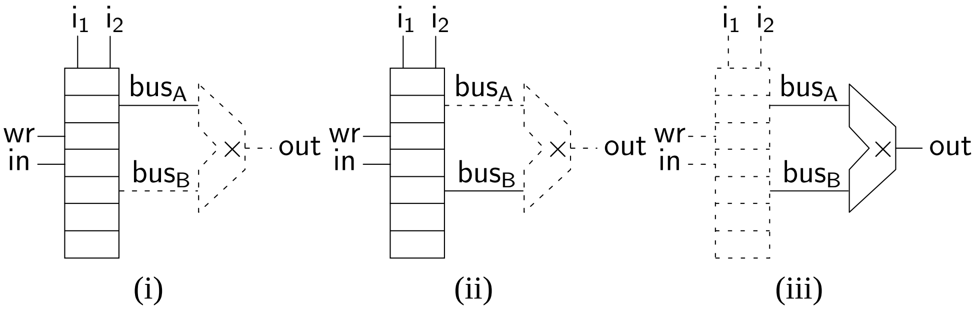


Figure 6.7: Three-Way Decomposition

hardware verification, where large datapaths are often the bottleneck in verification. This orthogonal treatment of data is the basis behind many other approaches to verification, from control state graph generation [HGD95] to the use of uninterpreted functions [BD94, HB95].

Chapter 7

Case Studies

In this chapter we examine two case-study verification efforts that allow us to explore the benefits of our proposed approach compared to existing methods of assertion graph specification. To be able to compare and contrast our approach with the most up-to-date existing verification methodology, for each case study we first provide details of verification using assertion graphs. We then show how the corresponding assertion programs can provide clearer and more succinct property representations, allowing a more structured approach to abstraction control. At the end of the chapter we provide a discussion of the two approaches. This chapter also serves to provide illustrative examples of the rules presented in Chapter 6.

7.1 First-In-First-Out Buffer

The first example is a 4-entry 10-bit-wide First-In-First-Out (FIFO) buffer, derived from [YS02]. Although the FIFO specification is relatively straight-forward, the verification makes a useful case study because of the different approaches to abstraction that are required.

7.1.1 Circuit Specification

The buffer is intended to hold a queue of 10-bit data elements. A reset operation initializes the buffer to an empty queue. Data can be enqueued, which means that it is added to the back of the queue. When data is dequeued, it is read and removed from the front of the queue, so that the first piece of data to enter the buffer is also the first piece of data to leave the buffer. In addition to these operations, the FIFO has two outputs that describe when the buffer is empty and full.

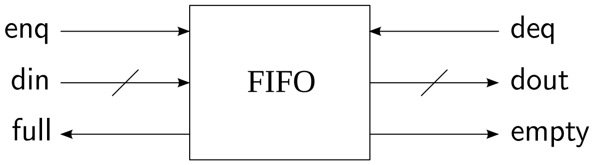


Figure 7.1: The FIFO Interface

As with most hardware, the FIFO circuit incorporates a delay factor, so that fewer gates are required between stateful elements, and the circuit can operate at a higher clock speed. The result of this is that the operations supported by the buffer take one clock cycle to complete. For example, if the buffer is empty and we enqueue a piece of data at cycle 0, then only at cycle 1 will the empty flag be set low and the data ready to be dequeued.

Figure 7.1 illustrates the FIFO circuit interface. In order to enqueue a piece of data, the enq line should be set high and the data presented on the din lines. If the FIFO is not already full then the data will be added to the buffer. In order to dequeue an element, the deq line should be set high. Provided the FIFO is not empty, then the oldest piece of data in the buffer will be presented at the dout lines and removed from the buffer.

7.1.2 Circuit Implementation

The success of GSTE verification depends on how well the employed abstraction fits with the structure of the circuit implementation. For this reason, it is important to take account of the circuit architecture when the verification is planned.

Our FIFO implementation has a memory array of content data together with a head and a tail pointer to mark the start and the end of the queue. When an enqueue occurs, data is written to the head pointer location and the head pointer is incremented. When data is dequeued from the buffer, it is read from the tail pointer location before the tail pointer is incremented. An extra bit of state, the *full bit*, is used to determine whether the buffer is full or empty in the cases where the head and tail pointers match. Figure 7.2 illustrates the state of the FIFO after enqueuing data 7 followed by 8.

7.1.3 Assertion Graph Verification

This section describes a FIFO verification using assertion graphs, closely based on the example in [YS02]. Of particular importance is the degree of clarity of the specification, and the difficulties with the application of refinement steps. In this example, a

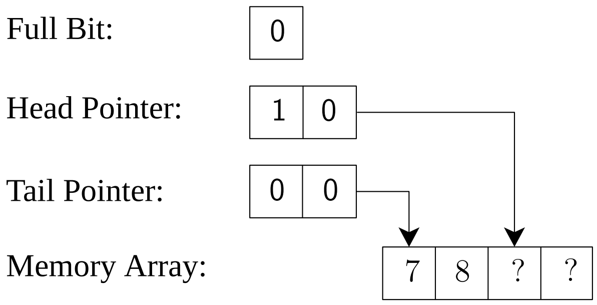


Figure 7.2: FIFO State After Adding 7 and 8

generic FIFO property assertion graph is progressively transformed through abstraction refinement to match the particular implementation at hand.

7.1.3.1 Generic FIFO Assertion Graph

Verification aims to check the following aspects of the circuit:

1. The empty output signal is set only when there are no entries in the FIFO.
2. The full output signal is set only when the FIFO is full.
3. If data is enqueued when the FIFO is not full, it is correctly dequeued after each of the previously enqueued entries have been dequeued.

In order to verify that the full and empty flags are set correctly, we can create an assertion graph with states that correspond to the number of entries currently in the FIFO. We can add to this graph all the possible transitions from one state to another, and assert for each of these transitions that the value of the empty and full lines are correct. Such an assertion graph, for a FIFO of depth four, is shown in Figure 7.3. Forward transitions represent enqueues, and backward transitions represent dequeues.

Next we must verify that data going through the FIFO is not corrupted. In order to do this, we consider an arbitrary piece of enqueued data and make sure than it is unchanged when it is dequeued. We use the variable v to represent the value of this data. During the transitions at which the data is enqueued, we will assert ‘din **is** v’. When we expect that same piece of data to be dequeued, we assert ‘dout **is** v’ in the consequent.

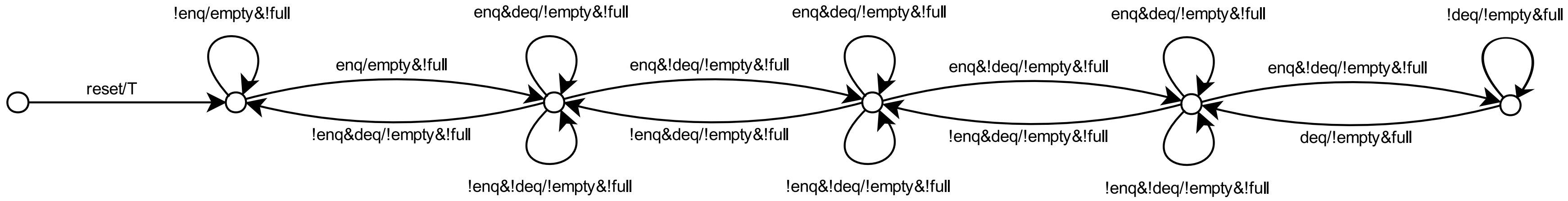


Figure 7.3: Assertion Graph for Empty and Full Signals

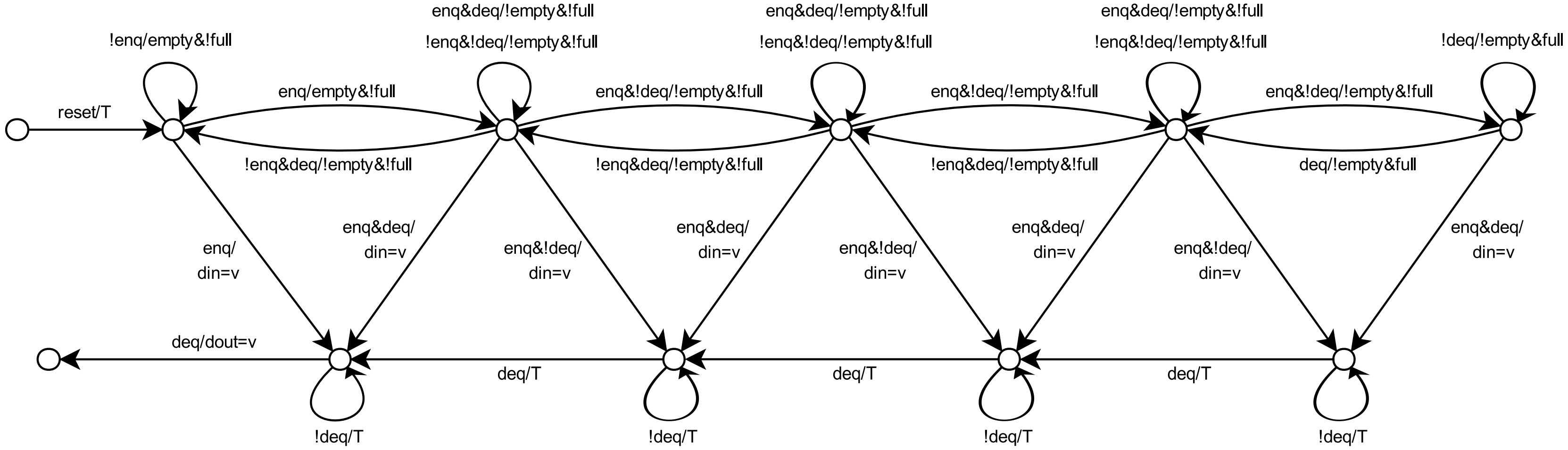


Figure 7.4: Generic 4-Place FIFO Assertion Graph

Since we need to make sure all enqueued data is handled correctly, we need to generalize our check to cover all possible starting states at which a piece of data might be enqueued. We also need to know the number of existing entries at this point, since we will expect the same number of dequeues before our data should be dequeued. We therefore include transitions that enqueue v starting from each of the states in Figure 7.3. After the relevant number of dequeues, we check that the correct data is dequeued at dout. The resulting graph is shown in Figure 7.4. The downward arrows are those transitions where v is enqueued, and v should be dequeued during the last transition in the bottom-left.

It is not immediately clear that this is the complete specification of a FIFO. In particular, the assertion graph only contains one symbolic constant, v, which is the value set on the din line when data is enqueued on certain edges of the graph. It therefore seems that if another piece of data were to be subsequently enqueued, then we would have no way of ensuring that this second data is not corrupted inside the FIFO. Using this single symbolic variable, however, we in fact are able to verify all possible sequences of enqueues and dequeues on the FIFO, because of the temporal abstraction introduced by the for-all semantics of assertion graphs. Consider that for every possible enqueue of data to the FIFO, there is some path through the graph where we check that this data is not corrupted. Hence it must be that all data enqueued to the FIFO is handled correctly.

7.1.3.2 Assertion Graph Refinement

When we use our first assertion graph attempt to verify our circuit, GSTE fails due to over-abstraction. To see why, we must consider the effects of using ternary states to characterize the states of our particular implementation.

The first vertex of the graph represents those states where the FIFO is empty. In our implementation, this corresponds to those circuit states where the head pointer is equal to the tail pointer and the full bit is low. By default, GSTE explores each of these states and forms the most precise ternary representation that includes all of them. Therefore it forms the $(\mathsf{head}, \mathsf{tail})$ vector

$$(00, 00) \sqcup (01, 01) \sqcup (10, 10) \sqcup (11, 11) = (\mathsf{XX}, \mathsf{XX})$$

In other words, since this head and tail pointers can, independently, take on any value in the states being characterized, GSTE loses all information about them. This then results in verification failure, since there is not enough information to determine if the

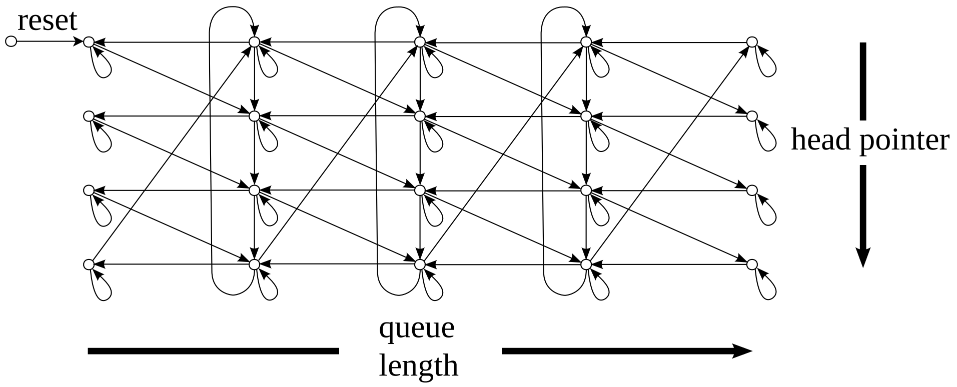


Figure 7.5: Assertion Graph After the Head Pointer Case-Split

FIFO is full or empty. We explore both explicit and symbolic approaches to surpassing this over-abstraction.

Splitting Graph Vertices The first approach involves splitting every vertex of the graph so that each resulting vertex corresponds to a single particular head/tail value pair. In order to preserve the property specified by the graph, we must ensure that the transitions cover the same possibilities as those of the original graph. Edges that do not completely specify enqueues and dequeues, such as those edges on the lower half of Figure 7.4, must be case-split to do so. This effect of this split on the first half of the assertion is shown in Figure 7.5. Now when simulation occurs, each ternary state stores a single concrete head/tail pointer state, so there is no over-abstraction.

Introducing Precise Nodes Although refinement by splitting vertices is sufficient for verification to succeed, the resulting number of states is quadratic in the size of the buffer. To avoid creating so many distinct ternary states, we can instead case-split symbolically by introducing variables to connect the head and tail pointers. This is done using *precise nodes* (see Section 2.5.4.5).

One advantage of this approach is that the case-split states and their corresponding simulation calculations are shared. For example, symbolically case-splitting based on a single node does not require us to keep two distinct simulation states. Another advantage is that it requires significantly less work to adapt the assertion graph. We must find the relevant head and tail pointer nodes in the circuit, and instruct the model checker to keep them precisely. We then need to introduce symbolic variables to ensure that enqueues and dequeues are completely specified.

Now when the simulation is run, GSTE internally allocates temporary symbolic variables z_0 and z_1 to capture the relation between the head and tail pointers. For example, the requirement that the two pointers are equal in the first vertex of the graph is captured by the symbolic state $(\mathsf{head}, \mathsf{tail}) = (z_1 z_0, z_1 z_0)$. Simulation propagates such symbolic dependencies to other areas of the circuit, for example, deducing that the empty flag is high in the example state.

7.1.4 Assertion Program Verification

We will now use our proposed methods to verify the FIFO. First, we write an assertion program to capture the requirements of a generic FIFO buffer. Our implementation framework includes support for bounded-length lists, which we will use to model the state of the buffer. We will then apply some of the simulation generation rules described in Chapter 6, and refine the simulation until verification succeeds.

7.1.4.1 A FIFO Assertion Program

The FIFO assertion program specification is shown in Figure 7.6. First we define attributes that parameterize the model. The size of the FIFO, SIZE, is 4, since it holds a maximum of 4 elements. The type of data being held in the FIFO, data, is set to be the type of 10-bit vectors.

We then declare the variables used by the program. The variable q holds the current list of elements stored in the buffer, in order of arrival. We use a second list variable, q', to store the partially updated contents. We also declare Boolean variables for the various input and output control signals, and variables din and dout of type data to read and write elements.

The model block contains the main substance of the high-level model. If reset is high, then the buffer is set to empty. Otherwise the contents state is set to its previous value, with enqueued data appended, and dequeued data removed. The variable q' holds the state at a stage where it has been updated to reflect enqueues but not dequeues. The empty and full flags are set to indicate the number of elements in the buffer. Since we have built the FIFO buffer as an independent circuit, we have been free to chose intuitive names for the interface nodes. Therefore the program interface is a simple directly-mapped interface.

Finally, the assertion block captures what circuit responses we wish to verify. Here it is important to note that the asserted responses are generally delayed by one time-step from the assertion programs. This accommodates the lag inherent in the circuit.

```
const SIZE = 4                              // Number of entries
type data = bool[9 : 0]                     // Type of contents

model
  var q, q′ : data list(SIZE)               // Queue states
  var reset, enq, deq, empty, full : bool   // Boolean signals
  var din, dout : data                      // Data in and out

  if reset then                             // Empty on reset
    q := [ ]
  else{
    if enq ∧ ¬last(full) then               // Handle enqueue
      q′ := last(q) ++ [din]
    else
      q′ := last(q)

    if deq ∧ ¬last(empty) then              // Handle dequeue
      q := tail(q′)
    else
      q := q′
  }

  empty := (length(q) = 0)                  // Set status bits
  full := (length(q) = SIZE)

interface
  node n_reset, n_enq, n_deq, n_din[9 : 0]
  node n_full, n_deq, n_dout[9 : 0]

  reset  :=  n_reset                       // Our FIFO has a
  enq    :=  n_enq                         // direct interface
  deq    :=  n_deq
  din    :=  n_din

assert
  last(empty) ⇒ n_empty                    // Check empty bit
  ¬last(empty) ⇒ ¬n_empty
  last(full) ⇒ n_full                      // Check full bit
  ¬last(full) ⇒ ¬n_full
  forall v : data .                        // Check data out
    deq ∧ last(¬empty ∧ head(q) = v) ⇒ n_dout = v
```

Figure 7.6: FIFO Buffer Assertion Program

Control Conditions	Precondition
reset	true
$\neg\mathsf{reset} \wedge \neg\mathsf{deq} \wedge \neg\mathsf{enq}$	$\mathrm{length(q)} < 4$
$\neg\mathsf{reset} \wedge \mathsf{deq} \wedge \neg\mathsf{enq}$	$1 < \mathrm{length(q)}$
$\neg\mathsf{reset} \wedge \neg\mathsf{deq} \wedge \mathsf{enq}$	$\mathrm{length(q)} < 3$
$\neg\mathsf{reset} \wedge \mathsf{deq} \wedge \mathsf{enq}$	$0 < \mathrm{length(q)} < 4$
$\neg\mathsf{reset} \wedge \mathsf{enq}$	$\mathrm{length(q)} = 0$
$\neg\mathsf{reset} \wedge \neg\mathsf{enq}$	$\mathrm{length(q)} = 0$
$\neg\mathsf{reset} \wedge \mathsf{deq}$	$\mathrm{length(q)} = 4$

Table 7.1: Preconditions for $\mathrm{length(q)} \neq 4$

7.1.4.2 A First Simulation Attempt

As a first attempt, we try applying the default `UNROLL` rule from Section 6.2 to generate the FIFO simulation run. This starts with the initial states characterised by the assertion antecedents:

1. $\mathbf{last}(\mathrm{length(q)} = 0)$
2. $\mathbf{last}(\mathrm{length(q)} \neq 0)$
3. $\mathbf{last}(\mathrm{length(q)} = 4)$
4. $\mathbf{last}(\mathrm{length(q)} \neq 4)$
5. $\mathrm{deq} \wedge \mathbf{last}(\neg\mathrm{empty} \wedge \mathrm{head(q)} = v)$

The first iteration of the backwards rewriting process finds the weakest preconditions of each of these states, under the various combinations of enqueues and dequeues that are possible. This is then repeated until all relevant simulation states have been explored.

As would be expected without any further intervention, this first simulation fails due to over-abstraction. To illustrate why, we will consider the preconditions for $\mathrm{length(q)} \neq 4$. Simulation generation considers the different control inputs that can bring about this condition, summarized in Table 7.1.

This shows that simulation generation by default creates different states for different *ranges* of queue lengths. Not only will this result in a large simulation, but it also introduces abstraction problems, since the most precise ternary representations of the head and tail pointers that characterize these states are all Xs.

7.1.4.3 Case-Splitting Simulation States

In order to avoid this over-abstraction, as well as to provide some extra structure to our simulation, we can instruct our environment to split states at the start of generation

```
if reset then
  $\text{head} := 0$
else {
  if $\text{enq} \wedge \neg\textbf{last}(\text{full})$ then
    $\text{head} := \textbf{last}(\text{head}) + 1$
  else
    $\text{head} := \textbf{last}(\text{head})$
}
```

Figure 7.7: Program Augmentation for Head Pointer

based on the queue length. This is achieved with the SPLIT_STATE from Section 6.3.2.4.

$$\texttt{SPLIT\_STATE} \left\{ \begin{array}{l} \text{length}(q) = 0, \text{length}(q) = 1, \\ \text{length}(q) = 2, \text{length}(q) = 3, \\ \text{length}(q) = 4 \end{array} \right\}$$

This helps, but when GSTE is invoked on the resulting simulation, verification still fails due to over-abstraction. As with the assertion graph case, this occurs because the dependencies between the head and tail pointers are not precisely represented. To avoid this we can instruct simulation generation to split cases based on the value of the head pointer. But since the assertion program does not include any information about this pointer, we must first augment our assertion program to describe its behaviour, as shown in Figure 7.7.

Now we split the abstract simulation states based on this pointer state:

$$\texttt{SPLIT\_STATE} \left\{ \begin{array}{l} \text{head} = 0,\ \text{head} = 1, \\ \text{head} = 2,\ \text{head} = 3 \end{array} \right\}$$

For the verification of the empty and full flags, the end result closely resembles the assertion graph of Figure 7.5. In order to verify the final assertion, we split the states according to the number of dequeues required before data element v should be seen. This hint has the effect of reducing the number of simulation states, since it aligns the abstract property states so that their images do not overlap. This ensures that the control state of the FIFO is never approximated to X, so verification succeeds. Some further miscellaneous term rewriting and weakening is also useful, as is trimming the predicates to keep them legible during debugging.

7.1.4.4 Case-Splitting Symbolically

As with the assertion graph case, rather than splitting over a quadratic number of simulation states, it can be more efficient to encode the pointer dependencies symbolically.

This has the added benefit that we are not required to augment the assertion program to describe how the head pointer behaves. In fact, symbolic simulation instead derives this information from the circuit model itself.

To achieve this symbolic-explicit hybrid, we use the `PRECISE_NODES` rule based on the transformations in Section 4.3.4, with the circuit nodes that make up the head and tail pointer states. This then introduces symbolic variables that capture the required dependencies between the two pointers, leading to successful verification.

7.2 Micro-Operation Scheduler

A *micro-operation (uop) scheduler* is a microprocessor component that receives a stream of instructions to be executed and is responsible for delivering each of these to an execution unit at an appropriate time. We verify a simple scheduler, based on a resource scheduler from the Intel Pentium 4 Microprocessor [Sch03, YGT05].

7.2.1 Circuit Specification

Each uop instruction consists of an *opcode*, a *source register* and a *destination register*, as shown below:

Opcode	Source Register	Destination Register
3 bits	4 bits	4 bits

The interface to the scheduler is shown in Figure 7.8. An instruction may only execute after all relevant previous instructions have finished writing to its source register. To signal when this condition occurs, each instruction carries a *ready* bit with it during its route through the scheduler. An instruction's ready bit is set high when all its dependencies have been executed. Instructions may be set ready before they enter the scheduler, or else become ready while inside. In order to enqueue an instruction

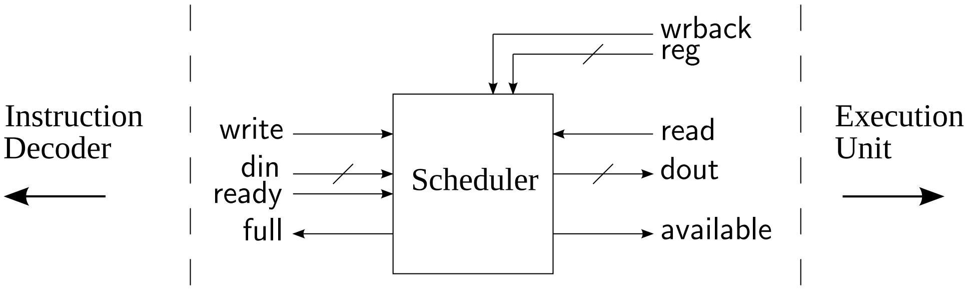


Figure 7.8: The Micro-Instruction Scheduler

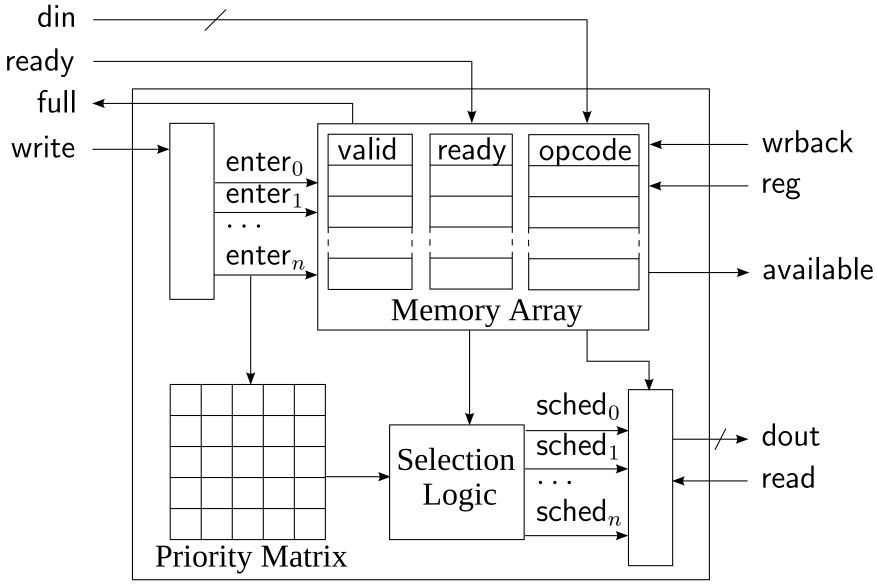


Figure 7.9: The Scheduler Implementation

into the scheduler (provided the full line is not set), the write line of the scheduler must be set high, the instruction presented on the din lines, and the ready bit for this instruction put on the ready line.

The wrback line from the execution unit is used to signal when the execution of an instruction has resulted in the writing of data to a particular register. The index of this register is supplied on the reg lines. There is an environmental assumption that any instruction in the scheduler is waiting for at most one other instruction to complete. Therefore any waiting instructions that have a source register matching a write-back register reg can have their ready bits safely set to high.

The scheduler should set the line available to high when it contains a waiting ready instruction. When it receives this signal, the execution unit can request the instruction on the dout lines by setting read to high. When there is more than one ready instruction, the scheduler provides the one that entered the scheduler first. Like the FIFO, the circuit incorporates some delay, so enqueues and write-backs require one cycle to become committed to state.

7.2.2 Circuit Implementation

As with the FIFO buffer, it is imperative to consider the structure of the implementation to be able to shape and balance the simulation abstraction. Figure 7.9 shows an outline of the scheduler implementation. The instructions are stored in a memory array, together with their valid and ready bits. The valid bits are used to signal whether there is currently an instruction stored in that index of the array. In order to store the relative arrival times of each instruction, the scheduler also contains a *priority matrix*. The (i, j)th entry of this matrix stores whether the instruction in index i of the memory array arrived before that stored in index j of the array. There is then some logic to determine which of the instructions stored is the earliest-entered ready instruction that is next to be sent off to the execution unit.

7.2.3 Assertion Graph Verification

We will first describe how the scheduler can be verified using GSTE assertion graphs. This approach follows the example from [YS04], which bypasses the entry logic and starts the simulation from the enter_i nodes rather than directly from the write input (see Figure 7.9). We also restrict ourselves to considering only the correctness of the data lines, missing out the full and available status lines.

Since the instruction emitted by the scheduler depends on the relative arrival times of all the instructions held, verifying the scheduler explicitly using a traditional assertion graph would require too many simulation states to be practical. This is because there would need to be one state for each set of possible relative arrival times.

We can make use of compositional GSTE, however, to overcome these difficulties. Recall that compositional GSTE allows us to simulate different parts of the circuit independently. In particular, whether an instruction is ready or not, and the relative arrival times of instructions are handled separately by independent areas of the circuit. Therefore if we simulate these sections separately, we can significantly reduce the required number of simulation states.

The symbolic quantification operations supported by compositional GSTE also allow us to effectively combine independent symbolic simulations. In particular, we are sometimes required to simulate conditions that apply to *every* instruction. Rather than simulate each possible interleaving of input conditions that brings about such a condition, we can instead simulate the requirement for only the general instruction at memory address i, and then universally quantify over i to reach the required condition.

The top-level assertion graph for the scheduler verification is shown in Figure 7.10. $EarliestReady(i, op)$ asserts that the ith instruction has associated data op, and is the earliest ready instruction being held. The node sched_i in the circuit signals that the ith index of the array is being scheduled. Hence the edge of the top-level graph can be read as saying that for every address i, if the instruction op has been stored in i and is the earliest ready instruction, and the execution unit is trying to read an instruction, then the instruction i is scheduled and value op is presented on dout.

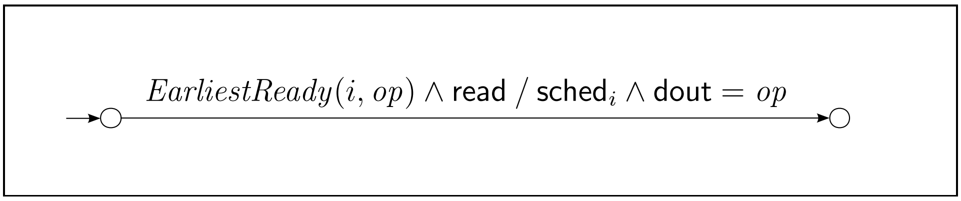

$EarliestReady(i, op) \wedge$ read / sched$_i \wedge$ dout $= op$

Figure 7.10: The Top Level Scheduler Assertion

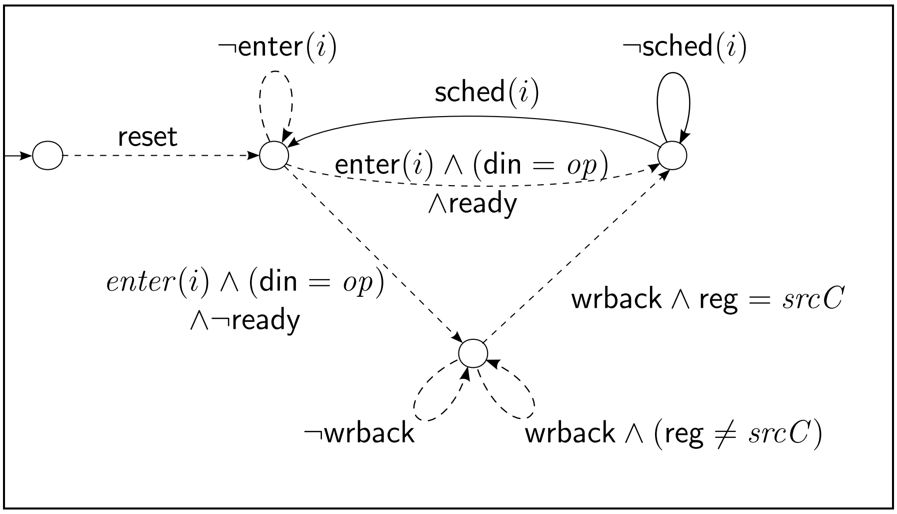

$\neg$enter(i)
$\neg$sched(i)
sched(i)
reset
enter$(i) \wedge$ (din $= op$)
$\wedge$ready
$enter(i) \wedge$ (din $= op$)
$\wedge\neg$ready
wrback $\wedge$ reg $= srcC$
$\neg$wrback
wrback $\wedge$ (reg $\neq srcC$)

Figure 7.11: The $Ready(i, op)$ Assertion

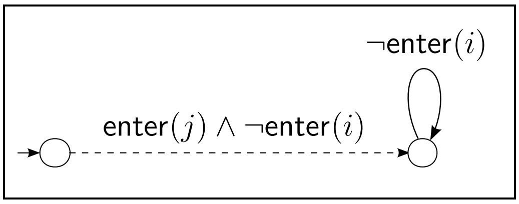


Figure 7.12: The $Earlier(i, j)$ Assertion Graph

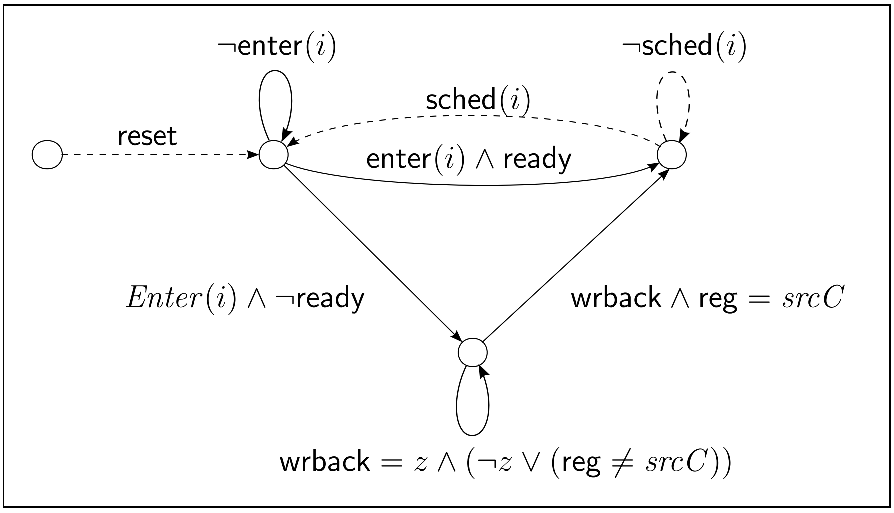

$\neg$enter(i)
$\neg$sched(i)
sched(i)
reset
enter$(i) \wedge$ ready
$Enter(i) \wedge \neg$ready
wrback $\wedge$ reg $= srcC$
wrback $= z \wedge (\neg z \vee ($reg $\neq srcC))$

Figure 7.13: The $NotReady(i)$ Assertion

The $EarliestReady(i)$ condition can be defined in terms of further simulation conditions, as:

$$EarliestReady(i, op) \quad := \quad Ready(i, op) \; \wedge \; \forall j \neq i \, . \, (Earlier(i, j) \vee NotReady(j))$$

The predicates $Ready(i, op)$, $Earlier(i, j)$ and $NotReady(j)$ are then defined in terms of the a set of edges on the additional assertion graphs shown in Figures 7.11-7.13. In these diagrams, the union of the states assigned to the solid lines is the condition represented.

The $Ready(i)$ assertion graph shows different paths for the two possible ways in which an instruction can become ready: either it is ready when it enters the scheduler (the top path), or else it is not ready when it enters and subsequently a write-back occurs to its source register (the bottom path).

The meaning of the $Earlier(i, j)$ graph is that an instruction has entered index j and no instruction has entered index i since. The final graph, $NotReady(i)$ uses the same outline of $Ready(i)$ to express those states in which the ith instruction either is not valid or else is not ready. These graphs are then enough to verify our circuit without the need for abstraction refinement.

7.2.4 Assertion Program Verification

We will now describe how we have verified the scheduler using GTL-based simulations created from an assertion program. The style of simulation generation is more interactive than that of the FIFO buffer, since the success of the verification is more dependent on the correct simulation approach that keeps the memory footprint small.

7.2.4.1 The Scheduler Assertion Program

The scheduler assertion program is shown in Figure 7.14. It first defines some types: index is the type of memory address locations, uop is the type for 11-bit uops, and $\mathrm{reg\_addr}$ is the type of register addresses.

We then declare the abstract state and variables used. The status bits for each uop held by the scheduler are stored as arrays indexed by the uop address. The state bit $\mathrm{valid}[i]$ indicates that there is an instruction in $\mathrm{contents}[i]$ with ready status $\mathrm{ready}[i]$. Bits $\mathrm{earlier}[i][j]$ are used to store whether or not the instruction at position i arrived before that at position j. There is then a series of variables that corresponds to the interface of the scheduler.

The state of the assertion program is updated as follows. When a reset occurs, every entry is set invalid, to empty the scheduler. Otherwise, if entry i is empty, and the environment is attempting to enqueue an instruction at position i, then the valid, ready and contents indices are stored for this new instruction. When an entry is valid and scheduled, it is subsequently set to invalid so that it will not be rescheduled. A write-back occurs when the write-back register input matches the last four bits of one of the held instructions. Under this condition, the corresponding ready bit is set to true. The earlier array is updated depending on which entries are entering the scheduler. Finally, $\mathrm{sched}[i]$ is set to describe whether the instruction at index i is to be currently scheduled or not. This is calculated using the condition that in order to be scheduled, a uop must be ready and valid, and have entered the scheduler earlier than any other valid ready entry. We omit an interface block for the assertion program, since our test circuit interface has the same names as those variables on the interface of the high-level model.

The aim of verification is to show that the scheduler circuit emits the correct instruction at any given time. This can be described using a single assertion where the antecedent describes the conditions under which uop op at memory address i should be scheduled. The consequent asserts that under this condition, the output lines of the scheduler match value op.

7.2.4.2 Generating the Simulation

We first simplify the verification by using the `DECOMPOSE` rule (Section 6.3.4) to relate the sched array of the assertion program with the circuit nodes sched_i, that we expect to signal that a given instruction address is being scheduled. This is a useful place to

```
type index = int(4)              // Contains maximum 4 uops
type uop = bool[int(11)]         // 11-bit uops
type reg_addr = bool[int(4)]     // 4-bit register addresses

model
  state contents : uop[index]                      // Stored uops
  state valid, ready, enter, sched : bool[index]   // Status bits
  state earlier : bool[index][index]               // Arrival times

  var reset, wrback, ready_in, read : bool         // Interface
  var wrback_reg : reg_addr                        // variables
  var uop_in : uop

  for i : index do {                             // For each uop...
    if last(reset) then
       valid[i] := false                         // Clear it on reset
    else if last(enter[i] ∧ ¬valid[i]) then {
      valid[i] := true                           // Store it on entry
      contents[i] := uop_in
      ready[i] := ready_in
    } else if last(read ∧ sched[i] ∧ valid[i]) then
      valid[i] := false                // Invalidate on schedule
    else if last(wrback ∧ valid[i] ∧ (contents[i][7 : 10] = wrback_reg))
      then
      ready[i] := true                 // Set ready on write-back

    for j : index do {                       // Record arrival times
      if last(enter[i]) ∧ ¬enter[j]) then
        earlier[i][j] := false
      else if last(enter[j]) then
        earlier[i][j] := true
    }

    // Schedule when a uop is the earliest ready uop
    sched[i] := ready[i] ∧ valid[i] ∧ forall j : index .
        ((i = j) ∨ ¬ready[j] ∨ ¬valid[j] ∨ earlier[i][j])
  }

assert                    // Check correct data is scheduled
  forall i : index , op : uop .
    read ∧ (sched[i] ∧ ∀j : index . (i = j) ∨ ¬sched[j]) ∧ (contents[i] = op)
      ⇒ (dout[0 : 10] = op)
```

Figure 7.14: Scheduler Assertion Program

decompose the verification because it allows us to separate out the control and data aspects of the circuit. This decomposition splits the main antecedent:

$$\text{read} \wedge (\text{sched}[i] \wedge \forall j : \text{index} \ . \ (i = j) \vee \neg\text{sched}[j]) \wedge (\text{contents}[i] = op)$$

into the two remaining simulation obligations:

1. $\text{sched}[i]$ **is** c
2. $\text{read} \wedge (\mathsf{sched}_i \wedge \forall j : \text{index} \ . \ (i = j) \vee \neg\mathsf{sched}_j) \wedge (\text{contents}[i] = op)$

First we must check that the sched array and sched_i nodes are in fact equivalent. Then, we must show that the scheduler operates correctly, given that this equivalence holds.

Verifying the Circuit Control The decomposition interface might be verified either symbolically, using:

$$\forall\, i : \text{index} \ , \ v : \mathbf{bool} \ . \ (\text{sched}[i] \ \mathbf{is} \ v \Rightarrow \mathsf{sched}_i \ \mathbf{is} \ v)$$

or explicitly, using:

$$\forall\, i : \text{index} \ . \ (\text{sched}[i] \Rightarrow \mathsf{sched}_i)$$
$$\forall\, i : \text{index} \ . \ (\neg\text{sched}[i] \Rightarrow \neg\mathsf{sched}_i)$$

Since it is a control signal, the input patterns that generate $\text{sched}[i]$ and $\neg\text{sched}[i]$ will have little in common. There is therefore no benefit to symbolically sharing the simulations, and an explicit approach is likely to be more efficient.

In order to go about generating the two conditions, we use the backwards rewriting technique from Chapter 6. By taking a single step in rewriting both conditions, we obtain:

$$\begin{aligned} \text{sched}[i] &= \neg(\mathbf{last}(\text{reset})) \wedge \text{valid}[i] \wedge \text{ready}[i] \\ &\quad \wedge \forall j.((i = j) \vee \text{earlier}[i][j] \vee \neg\text{valid}[j] \vee \neg\text{ready}[j]) \\ \\ \neg\text{sched}[i] &= \mathbf{last}(\text{reset}) \vee \neg\text{valid}[i] \vee \neg\text{ready}[i] \\ &\quad \vee \exists j.((i \neq j) \wedge \neg\text{earlier}[i][j] \wedge \text{valid}[j] \wedge \text{ready}[j]) \end{aligned}$$

At this point, we can make use of our knowledge of the circuit structure. Since we know that whether an instruction is valid, ready or entered before another instruction are all *independently* managed by the circuit, we can start creating separate simulations for each of these sub-formulas, leaving them to be combined at the final stage of simulation. We therefore use the `SPLIT` rule (Section 6.3.1.1) to create new recursion variables for each of: $\text{valid}[i]$, $\neg\text{valid}[i]$, $\text{ready}[i]$, $\neg\text{ready}[i]$, $\text{earlier}[i][j]$, $\neg\text{earlier}[i][j]$,

$\text{valid}[j]$, $\neg\text{valid}[j]$, $\text{ready}[j]$ and $\neg\text{ready}[j]$. We can reduce the number of these conditions by using the SYM_SUBSTITUTE rule (Section 6.3.3.2) to re-express $\text{valid}[j]$, $\neg\text{valid}[j]$, $\text{ready}[j]$ and $\neg\text{ready}[j]$ as conditions parameterized instead by i. For example, generating $\text{valid}[j]$ is the same as generating the condition $\text{valid}[i](i := j)$.

The remaining predicates can be simulated using the standard approach described by the UNROLL rule (Section 6.3.2.3). At each step in this generation process, we use the REDISTRIBUTE rule (Section 6.3.2.2) to make sure that simulation of the valid, ready, earlier and schedule conditions remain independent. Whether an instruction is ready or not depends on whether its source register matches any write-back registers. It is therefore necessary to also use the CREATE_VARIABLE rule to introduce variables that correspond to the ith instruction's source register. No further abstraction refinement steps are required for the verification of this aspect of the control to complete.

Verifying The Data Output We take a similar approach to the second condition, where we create the simulation of

$$\text{read} \wedge (\mathsf{sched}_i \wedge \forall j : \text{index} \,.\, (i = j) \vee \neg\mathsf{sched}_j) \wedge (\text{contents}[i] = op)$$

through the composition of separate simulations for each sub-formula. The only sub-formula that requires backwards propagation is the condition $\text{contents}[i] = op$, which is covered using the default rewriting strategy of the UNROLL rule (Section 6.3.2.3). No further abstraction refinement is required for the entire verification to succeed.

7.3 Discussion

In this chapter we have applied our verification approach to a first-in-first-out buffer and micro-operation scheduler. In both cases, we believe that the assertion program approach provides a more appropriate specification description than the use of assertion graphs. The cleanliness of the synchronous programming approach gives a clarity that lends a greater confidence to the interpretation of a property. In particular, the use of higher-level data-types such as lists and arrays allow us to reuse well-understood data structure concepts. Being textual and less explicit than assertion graphs, assertion programs seem less vulnerable to small specification errors. Assertion programs also enforce the separation of concerns between the model, the interface mapping, and the verification approach.

As well as this, assertion program specifications are more parameterized than assertion graphs because they can be configured via abstract constants. This not only allows for reuse, but provides far better scope for scalability. For example, the size of an assertion graph for a FIFO is linear in its depth. In contrast, changing the depth for an assertion program specification requires changing only a single constant literal. There is, however, additional effort required to generate the simulation outline for an assertion program. We have not explored the scalability of the simulation generation algorithm, partly due to the work involved with each generation and partly because our tool was built with a low degree of optimization. The scalability will, however, certainly exceed what is possible with manual assertion graph construction. The following cases highlight some other particular strengths of our approach that have been illustrated by the case studies.

In the original FIFO assertion graph approach, it is difficult to be assured of the correctness of *every* dequeue output, given that we represent them all using only a single instance of symbolic data. The verification is sound, however, due to the unusual for-all semantics of assertion graphs, coupled with the lack of initial hardware state. In the assertion program approach, we are aware all along that the simulation we produce will be sound, since each step in the simulation generation process is justified.

The original scheduler verification is difficult to understand, since it requires us to look at the concurrent execution of multiple assertion graphs in order to fully understand the original verification property. This is necessary because the specification must be decomposed in a particular way for the verification to succeed with limited memory space. In contrast, because the assertion program methodology separates the property from the verification approach, the scheduler assertion program is not required to be segregated in this way.

The scheduler verification proceeds via a decomposition based on the node sched_i, which describes if the operation in index i is scheduled at this time-step. In our approach, this step is extremely clear, and we have formally justified that the decomposition does not involve circular reasoning. In the assertion graph approach, however, node sched_i is used both in antecedents and consequents within the same graph, and it is not clear that the inductive step of the verification is actually constructive.

The case studies also demonstrated some of the practical advantages of using assertion programs over assertion graphs. Because the specification rules could all be described textually and applied automatically, specifications in GTL allowed for a more adaptive verification approach, where it was easier to test out different strategies. The textual nature of GTL also allowed for rapid command-line driven prototyping of simulations, greatly aiding abstraction refinement. Furthermore, model checking could be optimized in new ways, for example, through the use of simplification transformations and the caching of simulations.

Chapter 8

Conclusion

This dissertation has presented two specification notations and a methodology for verification using symbolic ternary simulation. The work has focused on taking particular promising techniques from generalized symbolic trajectory evaluation (GSTE), and recasting them using cleaner and more general notations that are more amenable to formal reasoning. The approach is organized into two layers: A low-level logic, *generalized trajectory logic (GTL)*, for specifying low-level simulation details, and the synchronous language of *assertion programs*, for the high-level modeling of circuit behaviour.

GTL is an intuitive and compositional linear temporal logic that provides a fine-grained formalism for describing symbolic ternary simulations. Each formula describes a particular flow of simulation, but also has a clean, trace-based, property semantics. By drawing an analogy between the atomic steps of simulation and the constructs of propositional logic, GTL and its algebraic properties are made to look familiar. Since it is textual in nature, GTL is also directly amenable to mechanized reasoning, easing the cleanliness of model checking algorithms, introspection techniques, and the construction and application of reasoning rules.

An appropriate choice of foundational semantics for GTL has enabled us to develop a wide range of reasoning rules, applicable in areas of abstraction refinement, decomposition, and simulation optimization. These go far beyond existing rule-sets for GSTE. The rules are generally based on the observation that sound simulation transformations in GTL correspond to semantic-preserving rewriting rules. Our rules also classify and further reveal the nature of GSTE verification choices, particularly for abstraction refinement and the mixing of symbolic and explicit verification.

For describing complete high-level specifications models, we have presented the language of *assertion programs*. This synchronous programming language allows model transitions to be described using abstract state built from rich and familiar vocabularies of arbitrary data types. Statements in the language are based around imperative assignments, but have a declarative semantics that allows for equational reasoning. In order to allow for greater clarity and reuse, assertion programs allow for the separation of concerns between the model, the circuit interface mapping, the verification approach and specification parameters. Assertion programs are more concise and descriptive than GSTE assertion graphs, because their abstract transition structure is not explicitly unwound. Combined with their textual nature, this makes assertion programs more prone to mechanized reasoning, and less vulnerable to small errors in specification.

The semantic nature of assertion programs has much in common with GTL: both are based on a finite trace-based semantics, a last time operator and similar variable characteristics. This has allowed us to describe a rule-based framework that connects the two formalisms by translating assertion programs into GTL properties that drive simulation. In this framework, the equational constraints of the assertion program are used to apply weakest precondition calculations that progressively construct the input sequence patterns necessary. Selective application of the reasoning rules for GTL allow simulations to be tailored to the implementations at hand. This provides a rigorous overall verification methodology that we have successfully applied to verify both a first-in-first-out buffer, and micro-operation scheduler.

Future Work

This section contains suggestions for future work, organized into topics that follow the same order as the dissertation contents.

Generalized Trajectory Logic

Although GTL is expressive enough to cover the most commonly used forms of GSTE, there are various currently inexpressible model checking extensions that the logic might be adapted to handle. For example, the GSTE approach to liveness properties [YS00] may correlate well with infinite trace variants of GTL. The backwards simulation approach in [YS02] may naturally correspond to the inclusion of a next time operator. The scope of GTL could also be extended for the specification of other abstraction techniques, such as *node weakening*, where a node is forced to X to limit the propagation of constraints, or *dynamic weakening*, which is a sound approach to trimming the model checking BDD structures [AJM$^+$00]. Goel's work [Goe04], which unifies the interaction between symbolic and ternary simulation, suggests the additional use of a partial order on variables to provide complete specifications of the variable dependencies that should be retained.

Each of these opportunities for extending GTL would naturally also increase the potential for additional reasoning rules for managing them. Future reasoning rules for GTL may also concentrate on how to encode additional abstraction techniques into the logic. For example, we have not included an analysis of rules for re-parameterization of indexing variables, although it is expected that techniques from STE [MJ02] will carry directly to GTL. Another area of interest might be exploring variation in the temporal scoping of indexing variables. For example, interesting new abstractions for capturing dependencies across time ranges might be attained by extending the persistence of the ghost variables in our precise nodes encoding.

Verification with Assertion Programs

There are various potential improvements to the language of assertion programs. The development of a suitable module system would increase scalability, heighten abstraction, improve variable scoping, and potentially map down into modular or incremental verification approaches. Other beneficial extensions might include the use of new data types, the addition of assumed environmental constraints, or polymorphic forms of specification.

Our interactive simulation generation framework opens questions about how best to selectively apply the various rewriting rules that we have described. In particular, there may be heuristics for suggesting various state abstractions, based on various analyses of the circuit or specification. For example, the identification of datapaths within the circuit might be a good indicator for the automatic symbolic treatment of the corresponding assertion program state. In cases where some of the assertion program structure is shared with the circuit, there may be techniques and heuristics for automatically determining internal equivalent nodes, or *cut points*, between the two. Such cut points are ideal candidates for the application of our decomposition rule for assertion programs. These techniques could potentially be based on scalar simulation tests, top-down structural comparisons, or on the similar techniques developed for equivalence checking [KK97].

There are many potential practical improvements to simulation generation that might either increase capacity or ease the abstraction refinement process. One way of increasing capacity might be to make use of different reasoning engines, such as those based on satisfiability checking [NO05]. Unbounded or very large data types might be better handled using decision procedures that operate directly on terms, or perhaps via the identification of a small model property. One difficulty experienced during our work that might inspire further research was the question of how to apply pattern matching across shared equational constraints, such as those found in vector GTL properties.

To help the process of abstraction refinement, the executable nature of assertion programs might be useful for generating illustrative counter-examples, or for discovering and validating circuit interface specifications. Graphical mapping of vector GTL simulations might help to emulate the visual benefits of assertion graph specifications. Suitable rewriting rules for identifying the causes of over-abstraction might be based on unrolling, simplification or case-splitting properties. It might also be beneficial to characterize existing (G)STE abstraction refinement algorithms [RC06b, ABMS07, CHXY07] using simulation generation rules for GTL.

There is also scope for extending the nature of verification above the level of abstraction provided by assertion programs. In particular, assertion programs have similar style to many other hardware modeling specifications, such as Murphi [DDHY92] and the Symbolic Analysis Laboratory (SAL) [Sha00], which are generally used during the verification of other forms of properties, such as liveness properties. If a high-level model is verified using these techniques, then it may then be connected directly to the gate-level circuit description through the application of our approach. In such a process symbolic ternary simulation would effectively abstract some of the circuit optimization and nondeterminism complexity, leaving a cleaner model for higher-level verification. There is also scope for using or adapting our approach to verify different types of refinement. For example, at the moment our high-level models must be either phase- or cycle-accurate. By exploiting different forms of temporal mappings, it should be possible to allow for temporal deviations within the refinement process.

Appendix A

GTL Characteristics

This appendix contains detailed proofs about the semantics of generalized trajectory logic.

A.1 Monotonicity

Our aim in this section is to demonstrate that GTL is monotonic with respect to recursion variables. Since GTL is a form of symbolically indexed structure, we first demonstrate some monotonicity results for symbolically indexed structures in general.

A.1.1 Symbolic Indexing Operators

In this section we show that both symbolic if-then-else and symbolic substitution are monotonic operations.

Lemma A.1.1. *If a and b are symbolic representations of the partial order $(X, \sqsubseteq)$, then 'if Q then a else b', written $Q \,\rightarrow\, a \,|\, b$, is monotonic in both a and b with respect to $\sqsubseteq^{\forall\mathcal{V}}$.*

Proof. First we show that the operation is monotonic with respect to the first argument. Suppose $a \sqsubseteq^{\forall\mathcal{V}} a'$ and pick any $\nu \in \mathcal{V}$.

Case $Q\langle\nu\rangle$. Then

$$\begin{aligned} (Q \,\rightarrow\, a \,|\, b)\langle\nu\rangle &= a\langle\nu\rangle \\ &\sqsubseteq a'\langle\nu\rangle \\ &= (Q \,\rightarrow\, a' \,|\, b)\langle\nu\rangle \end{aligned}$$

Case $\neg Q\langle\nu\rangle$. Then

$$\begin{aligned}(Q \to a \,|\, b)\langle\nu\rangle &= b\langle\nu\rangle \\ &= (Q \to a' \,|\, b)\langle\nu\rangle\end{aligned}$$

A similar argument demonstrates monotonicity with respect to b. □

Lemma A.1.2. *If a is a symbolic representation of the partial order $(X, \sqsubseteq)$, then the symbolic substitution of Q for u, $a(u := Q)$, is monotonic as a function of a with respect to $\sqsubseteq^{\forall\mathcal{V}}$.*

Proof. If $a \sqsubseteq^{\forall\mathcal{V}} a'$ then $a\langle\nu\rangle \sqsubseteq a'\langle\nu\rangle$ for any valuation ν. Since $\nu[u \mapsto Q\langle\nu\rangle]$ is also another valuation, it also implies that:

$$a\langle\nu[u \mapsto Q\langle\nu\rangle]\rangle \quad \sqsubseteq \quad a'\langle\nu[u \mapsto Q\langle\nu\rangle]\rangle$$

This is true for all ν, so $a(u := Q) \quad \sqsubseteq^{\forall\mathcal{V}} \quad a'(u := Q)$. □

A.1.2 Generalized Trajectory Logic

Theorem (3.3.2). *$\mathcal{R}_{f,\rho,Z}$ is monotonically increasing with respect to $\subseteq^{\forall\mathcal{V}}$.*

Proof. The proof is by structural induction over f.

Case $f = \mathbb{tt}, \mathbb{ff}, \mathsf{n}, \neg\mathsf{n}, W$ or $\mu Z \,.\, g$: Under any of these conditions, f is independent of Z. Therefore $\mathcal{R}_{f,\rho,Z}$ is constant, and the hypothesis trivially holds.

Case $f = Z$. In this case, $\mathcal{R}_{Z,\rho,Z}$ is the identity function, which is trivially monotonic.

Case $f = g \vee h$. Here $\mathcal{R}_{f,\rho,Z}(Q) = \mathcal{R}_{g,\rho,Z}(Q) \cup^{\mathcal{V}} \mathcal{R}_{h,\rho,Z}(Q)$. By the inductive hypothesis, both $\mathcal{R}_{g,\rho,Z}$ and $\mathcal{R}_{h,\rho,Z}$ are monotonic. Union is a monotonic set operation in both its operands. Therefore, $\cup^{\mathcal{V}}$ is monotonic with respect to $\subseteq$ in each valuation ν, so it is monotonic with respect to $\subseteq^{\forall\mathcal{V}}$. Hence $\mathcal{R}_{f,\rho,Z}$ is the composition of monotonic maps, and so is monotonic itself.

Case $f = g \wedge h$. Proof is as in the previous case, where union is replaced by intersection.

Case $f = \mathbf{Y}g$. Pick an arbitrary variable valuation ν, and let $Q, R \in (2^{S^+})^{\mathcal{V}}$ be semantic values where $Q \subseteq^{\forall\mathcal{V}} R$. Then, by the semantics of Yesterday, a trace t is in $\mathcal{R}_{\mathbf{Y}g,\rho,Z}(Q)\langle\nu\rangle$ if and only if $\mathrm{front}(t) \in \mathcal{R}_{g,\rho,Z}(Q)\langle\nu\rangle$. By the inductive hypothesis, $\mathcal{R}_{g,\rho,Z}$ is monotonic, so $\mathrm{front}(t) \in \mathcal{R}_{g,\rho,Z}(Q)\langle\nu\rangle$ implies $\mathrm{front}(t) \in \mathcal{R}_{g,\rho,Z}(R)\langle\nu\rangle$. Then by the semantics of Yesterday, this in turn implies $t \in \mathcal{R}_{\mathbf{Y}g,\rho,Z}(R)\langle\nu\rangle$. Therefore we have that $\mathcal{R}_{\mathbf{Y}g,\rho,Z}(Q)\langle\nu\rangle \subseteq \mathcal{R}_{\mathbf{Y}g,\rho,Z}(R)\langle\nu\rangle$ for any valuation ν, so $\mathcal{R}_{\mathbf{Y}g,\rho,Z}$ is monotonic.

Case $f = \mu W \,.\, g$ where $W \neq Z$. Pick an arbitrary variable valuation ν and assume $Q, R \in (2^{S^+})^{\mathcal{V}}$ are semantic values where $Q \subseteq^{\forall\mathcal{V}} R$. Now, by the semantics of μ-expressions, if $t \in \mathcal{R}_{(\mu W\,.\,g),\rho,Z}(Q)\langle\nu\rangle$ then for any semantic value $T \in (2^{S^+})^{\mathcal{V}}$:

$$\mathcal{R}_{g,\rho[W\mapsto T],Z}(Q) \;\subseteq^{\mathcal{V}}\; T \quad \text{implies} \quad t \in T\langle\nu\rangle \tag{A.1}$$

By the induction hypothesis, $\mathcal{R}_{g,\rho[W\mapsto T],Z}$ is monotonic. Hence for any other value $T' \in (2^{S^+})^{\mathcal{V}}$:

$$\begin{aligned}\mathcal{R}_{g,\rho[W\mapsto T],Z}(R) \;\subseteq^{\mathcal{V}}\; T' \quad &\Rightarrow \quad \mathcal{R}_{g,\rho[W\mapsto T],Z}(Q) \;\subseteq^{\mathcal{V}}\; T' && \text{(I.H.)}\\ &\Rightarrow \quad t \in T'\langle\nu\rangle && \text{(Equation A.1)}\end{aligned}$$

By applying the fixed-point definition again, this is equivalent to

$$t \in \mathcal{R}_{(\mu W\,.\,g),\rho,Z}(R)\langle\nu\rangle$$

hence $\mathcal{R}_{(\mu W\,.\,g),\rho,Z}$ is monotonic.

Case $f = Q \rightarrow g \,|\, h$ or $g(u := Q)$. By the induction hypothesis, $\mathcal{R}_{g,\rho,Z}$ and $\mathcal{R}_{h,\rho,Z}$ are both monotonic. By Lemmas A.1.1, and A.1.2 respectively, the semantics of f is also monotonic with respect to its sub-formulas. Therefore $\mathcal{R}_{f,\rho,Z}$ is also monotonic.

□

A.2 Continuity

Theorem (3.3.8). *For every recursion context ρ and GTL formula f:*

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(R))) \;\subseteq^{\forall\mathcal{V}}\; \mathcal{R}_{f,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,f)}(R))$$

and, as a consequence, the map $\mathcal{R}_{f,\rho,Z}$ is continuous.

Proof. For each formula it is shown that the following condition holds:

$$L^{\leq n}(\mathcal{R}_{f,\rho,Z}(R))) \;\subseteq^{\forall\mathcal{V}}\; \mathcal{R}_{f,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,f)}(R))$$

Each formula is consistently lengthening map by virtue of this and Lemma 3.3.2. Hence by Lemma 3.3.6, each is also chain-continuous.

Case $f = \mathtt{tt}, \mathtt{ff}, \mathsf{n}, \neg\mathsf{n}, \mu Z \,.\, g, W$. In these cases f is independent of Z, so $\mathcal{R}_{f,\rho,Z}$ is a constant, Q, and $\mathrm{depth}(Z,f)$ is ∞. It is clear that $L^{\leq n}(Q) \subseteq^{\mathcal{V}} Q$ for any such constant Q.

Case $f = Z$. In this case, $\mathcal{R}_{Z,\rho,Z}$ is the identity function, and $\mathrm{depth}(Z,Z) = 0$. Hence the hypothesis trivially holds.

Case $f = g \wedge h$.

$$\begin{array}{lll}
 & L^{\leq n}(\mathcal{R}_{(g\wedge h),\rho,Z}(R)) & \\
= & L^{\leq n}(\mathcal{R}_{g,\rho,Z}(R) \cap^{\mathcal{V}} \mathcal{R}_{h,\rho,Z}(R)) & \text{(Def. 3.2.2)} \\
= & L^{\leq n}(\mathcal{R}_{g,\rho,Z}(R)) \cap^{\mathcal{V}} L^{\leq n}(\mathcal{R}_{h,\rho,Z}(R)) & \text{(Property of } L^{\leq n}\text{)} \\
\subseteq^{\mathcal{V}} & \mathcal{R}_{g,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,g)}(R)) \cap^{\mathcal{V}} \mathcal{R}_{h,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,h)}(R)) & \text{(I.H.)} \\
\subseteq^{\mathcal{V}} & \begin{array}{l} \mathcal{R}_{g,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,g\wedge h)}(R)) \\ \quad \cap^{\mathcal{V}} \mathcal{R}_{h,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,g\wedge h)}(R)) \end{array} & \text{(Depth of } \wedge\text{)} \\
= & \mathcal{R}_{(g\wedge h),\rho,Z}(L^{\leq n-\mathrm{depth}(Z,g\wedge h)}(R)) & \text{(Def. 3.2.2)}
\end{array}$$

Case $f = g \vee h$. Proof mirrors the above case for conjunction.

Case $f = \mathbf{Y} g$. Pick any symbolic valuation ν and integer n. If $n \leq 0$ then the hypothesis trivially holds, since $L^{\leq n}(\mathcal{R}_{\mathbf{Y}g,\rho,Z}(R))) = \emptyset$. Suppose $n > 0$, then:

$$\begin{array}{lll}
 & L^{\leq n}(\mathcal{R}_{(\mathbf{Y}g),\rho,Z}(R))\langle\nu\rangle & \\
= & L^{\leq n}(\{\ \sigma.s \mid \sigma \in \|\ g\ \|_{\rho[Z\mapsto R]}\langle\nu\rangle\ \}) & \text{(Def. 3.2.2)} \\
= & \{\ \sigma.s \mid \sigma \in L^{\leq n-1}(\|\ g\ \|_{\rho[Z\mapsto R]})\langle\nu\rangle\ \} & \text{(Property of } L^{\leq n}\text{)} \\
\subseteq & \{\ \sigma.s \mid \sigma \in \|\ g\ \|^{\nu}_{\rho[Z\mapsto L^{\leq n-1-\mathrm{depth}(Z,g)}(R)]}\ \} & \text{(I.H.)} \\
= & \mathcal{R}_{(\mathbf{Y}g),\rho,Z}(L^{\leq n-1-\mathrm{depth}(Z,g)}(R))\langle\nu\rangle & \text{(Def. 3.2.2)} \\
= & \mathcal{R}_{(\mathbf{Y}g),\rho,Z}(L^{\leq n-\mathrm{depth}(Z,\mathbf{Y}g)}(R))\langle\nu\rangle & \text{(Depth of } \mathbf{Y}\text{)}
\end{array}$$

Case $f = \mu W \,.\, g$. Since $\mathcal{R}_{g,\rho,W}$ is monotonic it has a least fixed-point

$$\mu\mathcal{R}_{g,\rho,W} \quad = \quad \bigcap\{\ T \in (2^{S^+})^{\mathcal{V}} \mid \mathcal{R}_{g,\rho,W}(T) \subseteq^{\mathcal{V}} T\ \}$$

by the Knaster-Tarski Theorem. This is precisely the definition of the semantics of $\mu W \,.\, g$. By the inductive hypothesis, $\mathcal{R}_{g,\rho,W}$ is continuous, so the Knaster-Tarski Theorem guarantees that the fixed-point is the limit of the approximants given by:

$$\| \, \mu W \,.\, g \, \|_\rho \quad = \quad \bigcup_{n \geq 0}^{\mathcal{V}} (\mathcal{R}_{g,\rho,W})^n(\bot) \tag{A.2}$$

Hence

$$\begin{aligned}
& L^{\leq n}(\mathcal{R}_{(\mu W \,.\, g),\rho,Z}(R)) \\
= \ & L^{\leq n}(\bigcup_{m \geq 0}^{\mathcal{V}} (\mathcal{R}_{g,\rho[Z \mapsto R],W})^m(\bot)) && \text{(Equation A.2)} \\
= \ & \bigcup_{m \geq 0}^{\mathcal{V}} L^{\leq n}((\mathcal{R}_{g,\rho[Z \mapsto R],W})^m(\bot)) && \text{(Lemma 3.3.4)} \\
\subseteq^{\mathcal{V}} \ & \bigcup_{m \geq 0}^{\mathcal{V}} (\mathcal{R}_{g,\rho[Z \mapsto L^{\leq n - \mathrm{depth}(Z,g)}(R)],W})^m(\bot) && \text{(I.H.)} \\
= \ & \mathcal{R}_{(\mu W \,.\, g),\rho,Z}(L^{\leq n - \mathrm{depth}(Z,\mu W \,.\, g)}(R)) && \text{(Equation A.2)}
\end{aligned}$$

Case $f = Q \;\rightarrow\; g \,|\, h$.

$$\begin{aligned}
& L^{\leq n}(\mathcal{R}_{(Q \rightarrow g \,|\, h),\rho,Z}(R)) \\
= \ & L^{\leq n}(\lambda\nu \,.\, \text{if } Q\langle\nu\rangle \text{ then } \mathcal{R}_{g,\rho,Z}(R)\langle\nu\rangle \text{ else } \mathcal{R}_{h,\rho,Z}(R)\langle\nu\rangle) && \text{(Def. 3.2.2)} \\
= \ & \lambda\nu \,.\, \text{if } Q\langle\nu\rangle \text{ then } L^{\leq n}(\mathcal{R}_{g,\rho,Z}(R))\langle\nu\rangle \\
& \quad \text{else } L^{\leq n}(\mathcal{R}_{h,\rho,Z}(R))\langle\nu\rangle && \text{(Property of } L^{\leq n}\text{)} \\
\subseteq^{\mathcal{V}} \ & \lambda\nu \,.\, \text{if } Q\langle\nu\rangle \text{ then } \mathcal{R}_{g,\rho,Z}(L^{\leq n - \mathrm{depth}(Z,g)}(R))\langle\nu\rangle \\
& \quad \text{else } \mathcal{R}_{h,\rho,Z}(L^{\leq n - \mathrm{depth}(Z,h)}(R))\langle\nu\rangle && \text{(I.H.)} \\
\subseteq^{\mathcal{V}} \ & \lambda\nu \,.\, \text{if } Q\langle\nu\rangle \text{ then } \mathcal{R}_{g,\rho,Z}(L^{\leq n - \mathrm{depth}(Z,Q \rightarrow g \,|\, h)}(R))\langle\nu\rangle \\
& \quad \text{else } \mathcal{R}_{h,\rho,Z}(L^{\leq n - \mathrm{depth}(Z,Q \rightarrow g \,|\, h)}(R))\langle\nu\rangle && \text{(Depth of } \rightarrow \;|\text{)} \\
= \ & \mathcal{R}_{(Q \rightarrow g \,|\, h),\rho,Z}(L^{\leq n - \mathrm{depth}(Z,Q \rightarrow g \,|\, h)}(R)) && \text{(Def. 3.2.2)}
\end{aligned}$$

Case $f = g(u := Q)$.

$$\begin{array}{lll} & L^{\leq n}(\mathcal{R}_{g(u:=Q),\rho,Z}(R)) & \\ = & L^{\leq n}(\lambda\nu \,.\, \mathcal{R}_{g,\rho,Z}(R)\langle\nu[u \mapsto Q\langle\nu\rangle]\rangle) & \text{(Def. 3.2.2)} \\ = & \lambda\nu \,.\, L^{\leq n}(\mathcal{R}_{g,\rho,Z}(R))\langle\nu[u \mapsto Q\langle\nu\rangle]\rangle & \text{(Property of } L^{\leq n}\text{)} \\ \subseteq^{\mathcal{V}} & \lambda\nu \,.\, \mathcal{R}_{g,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,g)}(R))\langle\nu[u \mapsto Q\langle\nu\rangle]\rangle & \text{(I.H.)} \\ = & \lambda\nu \,.\, \mathcal{R}_{g,\rho,Z}(L^{\leq n-\mathrm{depth}(Z,g(u:=Q))}(R))\langle\nu[u \mapsto Q\langle\nu\rangle]\rangle & \text{(Depth of :=)} \\ = & \mathcal{R}_{(g(u:=Q)),\rho,Z}(L^{\leq n-\mathrm{depth}(Z,g(u:=Q))}(R)) & \text{(Def. 3.2.2)} \end{array}$$

□

A.3 Set-based Model Checking

Lemma (3.6.3)**.** *For any closed atemporal formula f, symbolic valuation ν, and word $t.s \in S^+$, where s is a state in S, $t.s$ satisfies f if and only if the singleton word s satisfies f:*

$t.s \in \parallel f \parallel^\nu$ *iff* $s \in \parallel f \parallel^\nu$

Proof. Since f is atemporal, it does not contain $\mathbf{Y}$, fixed-points, or recursion variables. Hence every sub-formula is also closed. We show the property directly for these cases, using structural induction on f.

Case $f = \mathtt{tt}$. The result trivially true, since $\parallel f \parallel^\nu = S^+$.

Case $f = \mathtt{ff}$. The result vacuously true, since $\parallel f \parallel^\nu = \emptyset$.

Case $f = \mathsf{n}$.

$$\begin{array}{llll} t.s \in \parallel \mathsf{n} \parallel^\nu & \Longleftrightarrow & (\mathrm{last}(t.s))(\mathsf{n}) = 1 & \text{(Def. 3.2.2)} \\ & \Longleftrightarrow & (\mathrm{last}(s))(\mathsf{n}) = 1 & \text{(Property of last)} \\ & \Longleftrightarrow & s \in \parallel \mathsf{n} \parallel^\nu & \text{(Def. 3.2.2)} \end{array}$$

Case $f = \neg\mathsf{n}$. Proof follows the previous case.

Case $f = g \vee h$.

$$\begin{array}{llll} t.s \in \parallel g \vee h \parallel^\nu & \Longleftrightarrow & t.s \in \parallel g \parallel^\nu \cup \parallel h \parallel^\nu & \text{(Def. 3.2.2)} \\ & \Longleftrightarrow & s \in \parallel g \parallel^\nu \cup \parallel h \parallel^\nu & \text{(I.H.)} \\ & \Longleftrightarrow & s \in \parallel g \vee h \parallel^\nu & \text{(Def. 3.2.2)} \end{array}$$

Case $f = g \wedge h$. Proof is as previous case, by replacing union with intersection.

Case $f = Q \rightarrow g \,|\, h$.

$$\begin{aligned} t.s \in \|\, Q \rightarrow g \,|\, h \,\|^{\nu} &\iff t.s \in (\text{if } Q\langle\nu\rangle \text{ then } \|\, g \,\|^{\nu} \text{ else } \|\, h \,\|^{\nu}) && \text{(Def. 3.2.2)} \\ &\iff s \in (\text{if } Q\langle\nu\rangle \text{ then } \|\, g \,\|^{\nu} \text{ else } \|\, h \,\|^{\nu}) && \text{(I.H.)} \\ &\iff s \in \|\, Q \rightarrow g \,|\, h \,\|^{\nu} && \text{(Def. 3.2.2)} \end{aligned}$$

Case $f = g(u := Q)$.

$$\begin{aligned} t.s \in \|\, g(u := Q) \,\|^{\nu} &\iff t.s \in \|\, g \,\|^{\nu[u \mapsto Q\langle\nu\rangle]} && \text{(Def. 3.2.2)} \\ &\iff s \in \|\, g \,\|^{\nu[u \mapsto Q\langle\nu\rangle]} && \text{(I.H.)} \\ &\iff s \in \|\, g(u := Q) \,\|^{\nu} && \text{(Def. 3.2.2)} \end{aligned}$$

□

Lemma (3.6.5)**.** *For every formula f, simulation terminates and is monotonic in the simulation context of each recursion variable.*

Proof. Proof is by induction, ordering first by the number of fixed-points in formula f, and secondly by length of f.

Case $f = \mathtt{tt}, \mathtt{ff}, \mathsf{n},$ or $\neg\mathsf{n}$. In these cases, termination is trivial, and since the formulas contain no recursion variables, the simulations are constant with respect to the simulation context.

Case $f = g \vee h, g \wedge h, \mathbf{Y}g, Q \rightarrow f \,|\, g,$ or $f(u := Q)$. For each of these, the simulation of the sub-formulas terminate by the induction hypothesis. As a result, each of these cases terminates, since they consist of single calculation steps. Also by the induction hypothesis, the simulation of each sub-formula is monotonic with respect to each recursion variable. Therefore each of these simulation cases are also monotonic, using monotonicity of $\cup$, $\cap$, post, $(\rightarrow |)$ (Lemma A.1.1) and $(:=)$ (Lemma A.1.2) respectively.

Case $f = Z$. Termination is assured, since the simulation of a recursion variable only consists of looking up the variable in a store. Now suppose $\tau(Z) \subseteq^{\forall\mathcal{V}} \tau'(Z)$ for any $Z \in \mathcal{F}$. Then $[\, Z \,]^{\nu}_{\tau} = \tau(Z) \subseteq \tau'(Z) = [\, Z \,]^{\nu'}_{\tau}$, and hence simulation is monotonic.

Case $f = \mu Z \,.\, f$. By the induction hypothesis, simulation of f is monotonic with respect to Z. Hence the simulation iterations $((\lambda S \,.\, [\, f \,]_{\tau[Z \mapsto S]})^n \, (\lambda\nu.\emptyset))$ form a chain. Since the domain of simulation is finite, we reach a fixed-point where simulation terminates. For monotonicity, suppose $\tau \subseteq^{\forall\mathcal{V}} \tau'$. Then:

$$[\, \mu Z \,.\, f \,]_\tau \;=\; \bigcup_{n \geq 0} [\, \mu^n Z \,.\, f \,]_\tau \;\subseteq^{\forall\mathcal{V}}\; \bigcup_{n \geq 0} [\, \mu^n Z \,.\, f \,]_{\tau'} \;=\; [\, \mu Z \,.\, f \,]_{\tau'} \qquad \square$$

Lemma (3.6.7)**.** *For any closed GTL formula f, the simulation of f is an upper-approximation of the image of f:*

$$\mathrm{im}(f) \;\subseteq^{\forall\mathcal{V}}\; [\, f \,]$$

Proof. We show a stronger property for any formula f of GTL, including those with free variables. The added assumption is that the simulation of each recursion variable up until now is an upper-approximation of the traced-based context. The condition is that for any trace recursion context ρ and simulation recursion context τ:

$$(\forall Z \in \mathcal{F} \,.\, \mathrm{im}_\rho(Z) \subseteq^{\forall\mathcal{V}} [\, Z \,]_\tau) \quad \text{implies} \quad \mathrm{im}_\rho(f) \subseteq^{\forall\mathcal{V}} [\, f \,]_\tau$$

Proof is by induction, ordering first by the length of a formula, and secondly by the number of fixed-points in a formula. We assume $\mathrm{im}_\rho(Z) \subseteq^{\forall\mathcal{V}} [\, Z \,]_\tau$ for each variable in each case, and demonstrate $\mathrm{im}_\rho(f) \subseteq^{\forall\mathcal{V}} [\, f \,]_\tau$.

Case $f = \mathtt{tt}$. Trivially true, since $[\, \mathtt{tt} \,]^\nu_\tau = S$.

Case $f = \mathtt{ff}$. Vacuously true, since:

$$\begin{array}{lll}
 & \mathrm{im}_\rho(\mathtt{ff})\langle\nu\rangle & \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \|\, \mathtt{ff} \,\|^\nu_\rho) & \text{(Def. 3.6.1)} \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \emptyset) & \text{(Def. 3.2.2)} \\
= & \emptyset &
\end{array}$$

Case $f = \mathsf{n}$.

$$\begin{array}{lll}
 & \mathrm{im}_\rho(\mathsf{n})\langle\nu\rangle & \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \|\, \mathsf{n} \,\|^\nu_\rho) & \text{(Def. 3.6.1)} \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \{t \in S^+ \mid (\mathrm{last}(t))(\mathsf{n}) = 1\}) & \text{(Def. 3.2.2)} \\
\subseteq & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}})) \cap \mathrm{last}(\{t \in S^+ \mid (\mathrm{last}(t))(\mathsf{n}) = 1\}) & \text{(Property of last)} \\
= & S \cap \{t \in S \mid t(\mathsf{n}) = 1\} & \text{(Properties of last)} \\
= & \{t \in S \mid t(\mathsf{n}) = 1\} & \text{(Set algebra)} \\
= & [\, \mathsf{n} \,]^\nu_\tau & \text{(Def. 3.6.4)}
\end{array}$$

Case $f = \neg \mathsf{n}$. Proof follows the previous case.

Case $f = Z$. Covered directly by assumption.

Case $f = g \vee h$.

$$\begin{aligned}
& \mathrm{im}_\rho(g \vee h)\langle \nu \rangle && \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \vee h \, \|^\nu_\rho) && \text{(Def. 3.6.1)} \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap (\| \, g \, \|^\nu_\rho \cup \| \, h \, \|^\nu_\rho)) && \text{(Def. 3.2.2)} \\
= \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho) \cup (\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, h \, \|^\nu_\rho)) && \text{(Set Algebra)} \\
= \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho)) \cup \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, h \, \|^\nu_\rho)) && \text{(Property of } \mathrm{last}) \\
\subseteq \;& [\, g \,]^\nu_\tau \cup [\, h \,]^\nu_\tau && \text{(I.H.)} \\
= \;& [\, g \vee h \,]^\nu_\tau && \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = g \wedge h$.

$$\begin{aligned}
& \mathrm{im}_\rho(g \wedge h)\langle \nu \rangle && \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \wedge h \, \|^\nu_\rho) && \text{(Def. 3.6.1)} \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap (\| \, g \, \|^\nu_\rho \cap \| \, h \, \|^\nu_\rho)) && \text{(Def. 3.2.2)} \\
= \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho) \cap (\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, h \, \|^\nu_\rho)) && \text{(Set Algebra)} \\
\subseteq \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho)) \cap \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, h \, \|^\nu_\rho)) && \text{(Property of } \mathrm{last}) \\
\subseteq \;& [\, g \,]^\nu_\tau \cap [\, h \,]^\nu_\tau && \text{(I.H.)} \\
= \;& [\, g \wedge h \,]^\nu_\tau && \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = \mathbf{Y} g$.

$$\begin{aligned}
& \mathrm{im}_\rho(\mathbf{Y} g)\langle \nu \rangle && \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, \mathbf{Y} g \, \|^\nu_\rho) && \text{(Def. 3.6.1)} \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \{ t.s \in S^+ \mid t \in \| \, g \, \|^\nu_\rho \}) && \text{(Def. 3.2.2)} \\
= \;& \mathrm{last}(\{ t.s \in S^+ \mid (\mathrm{last}(t), s) \in T \wedge t \in \mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho \}) && \text{(Prop. of } \mathrm{tr}) \\
= \;& \{ s \in S \mid (\mathrm{last}(t), s) \in T \wedge t \in \mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho \} && \text{(Def. } \mathrm{last}) \\
= \;& \{ s \in S \mid (s', s) \in T \wedge s' \in \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho) \} && \text{(Guard Rewrite)} \\
= \;& \mathrm{post}(\mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \, g \, \|^\nu_\rho)) && \text{(Def. } \mathrm{post}) \\
= \;& \mathrm{post}(\mathrm{im}_\rho(g)\langle \nu \rangle) && \text{(Def. } \mathrm{im}) \\
\subseteq \;& \mathrm{post}([\, g \,]^\nu_\tau) && \text{(I.H., } \mathrm{post} \text{ monotone)} \\
= \;& [\, \mathbf{Y} g \,]^\nu_\tau && \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = \mu Z\,.\,g$

$$\begin{array}{lll}
 & \mathrm{im}_\rho(\mu Z\,.\,g)\langle\nu\rangle & \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap \|\ \mu Z\,.\,g\ \|_\rho^\nu) & \text{(Def. 3.6.1)} \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap \bigcup_{n\geq 0} \|\ \mu^n Z\,.\,g\ \|_\rho^\nu) & \text{(Corollary 3.3.10)} \\
= & \bigcup_{n\geq 0} \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap \|\ \mu^n Z\,.\,g\ \|_\rho^\nu) & \text{(Set Algebra)} \\
\subseteq & \bigcup_{n\geq 0} [\,\mu^n Z\,.\,g\,]_\tau^\nu & \text{(I.H.)} \\
= & [\,\mu Z\,.\,g\,]_\tau^\nu & \text{(Def. 3.6.4)}
\end{array}$$

Case $f = Q \rightarrow g \,|\, h$.

$$\begin{array}{lll}
 & \mathrm{im}_\rho(Q \rightarrow g \,|\, h)\langle\nu\rangle & \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap \|\ Q \rightarrow g \,|\, h\ \|_\rho^\nu) & \text{(Def. 3.6.1)} \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap (\text{if } Q\langle\nu\rangle \text{ then } \|\ g\ \|_\rho^\nu \text{ else } \|\ h\ \|_\rho^\nu)) & \text{(Def. 3.2.2)} \\
= & \text{if } Q\langle\nu\rangle \text{ then } \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap \|\ g\ \|_\rho^\nu) \text{ else } \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap \|\ h\ \|_\rho^\nu)) & \text{(Set Algebra)} \\
\subseteq & \text{if } Q\langle\nu\rangle \text{ then } [\,g\,]_\tau^\nu \text{ else } [\,h\,]_\tau^\nu & \text{(I.H.)} \\
= & [\,Q \rightarrow g \,|\, h\,]_\tau^\nu & \text{(Def. 3.6.4)}
\end{array}$$

Case $f = g(u := Q)$.

$$\begin{array}{lll}
 & \mathrm{im}_\rho(g(u := Q))\langle\nu\rangle & \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap \|\ g(u := Q)\ \|_\rho^\nu) & \text{(Def. 3.6.1)} \\
= & \mathrm{last}(\mathrm{tr}(\mathcal{K}_\mathbf{C}) \cap (\|\ g\ \|_\rho^{\nu[u\mapsto Q\langle\nu\rangle]})) & \text{(Def. 3.2.2)} \\
\subseteq & [\,g\,]_\tau^{\nu[u\mapsto Q\langle\nu\rangle]} & \text{(I.H.)} \\
= & [\,g(u := Q)\,]_\tau^\nu & \text{(Def. 3.6.4)}
\end{array}$$

□

A.3.1 Atemporal Formulas

Lemma (3.6.8)**.** *For any closed atemporal GTL formula f, the simulation of f is equal to the image of f:*

$$\mathrm{im}(f) \quad = \quad [\,f\,]$$

Proof. Since f is atemporal, it does not contain $\mathbf{Y}$, fixed-points, or recursion variables. Hence every sub-formula is also closed. We show by induction that

$$\mathrm{im}(f) \quad = \quad [\, f\,]$$

Case $f = \mathtt{tt}$.

$$\begin{aligned}
& \mathrm{im}(\mathtt{tt})\langle\nu\rangle & \\
= \; & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \|\ \mathtt{tt}\ \|^{\nu}) & \text{(Def. 3.6.1)} \\
= \; & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap S^{+}) & \text{(Def. 3.2.2)} \\
= \; & S & \text{(Def. tr)} \\
= \; & [\, \mathtt{tt}\,]^{\nu} & \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = \mathtt{ff}$.

$$\begin{aligned}
& \mathrm{im}(\mathtt{ff})\langle\nu\rangle & \\
= \; & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \|\ \mathtt{ff}\ \|^{\nu}) & \text{(Def. 3.6.1)} \\
= \; & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \emptyset) & \text{(Def. 3.2.2)} \\
= \; & \emptyset & \text{(Def. tr)} \\
= \; & [\, \mathtt{ff}\,]^{\nu} & \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = \mathsf{n}$.

$$\begin{aligned}
& \mathrm{im}(\mathsf{n})\langle\nu\rangle & \\
= \; & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \|\ \mathsf{n}\ \|^{\nu}) & \text{(Def. 3.6.1)} \\
= \; & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \{t \in S^{+} \mid (\mathrm{last}(t))(\mathsf{n}) = 1\}) & \text{(Def. 3.2.2)} \\
= \; & \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}})) \cap \{t \in S \mid t(\mathsf{n}) = 1\} & \text{(Property of last)} \\
= \; & \{t \in S \mid t(\mathsf{n}) = 1\} & \text{(Set algebra)} \\
= \; & [\, \mathsf{n}\,]^{\nu} & \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = \neg\mathsf{n}$. Proof follows the previous case.

Case $f = g \vee h$.

$$\begin{aligned}
& \mathrm{im}(g \vee h)\langle \nu \rangle \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g \vee h \; \|^{\nu}) && \text{(Def. 3.6.1)} \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap (\| \; g \; \|^{\nu} \cup \| \; h \; \|^{\nu})) && \text{(Def. 3.2.2)} \\
= \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g \; \|^{\nu}) \cup (\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; h \; \|^{\nu})) && \text{(Set Algebra)} \\
= \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g \; \|^{\nu})) \cup \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; h \; \|^{\nu})) && \text{(Set Algebra)} \\
= \;& [\, g \,]^{\nu} \cup [\, h \,]^{\nu} && \text{(I.H.)} \\
= \;& [\, g \vee h \,]^{\nu} && \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = g \wedge h$.

$$\begin{aligned}
& \mathrm{im}(g \wedge h)\langle \nu \rangle \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g \wedge h \; \|^{\nu}) && \text{(Def. 3.6.1)} \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap (\| \; g \; \|^{\nu} \cap \| \; h \; \|^{\nu})) && \text{(Def. 3.2.2)} \\
= \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g \; \|^{\nu}) \cap (\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; h \; \|^{\nu})) && \text{(Set Algebra)} \\
= \;& \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g \; \|^{\nu})) \cap \mathrm{last}((\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; h \; \|^{\nu})) && \text{(Lemma 3.6.3)} \\
= \;& [\, g \,]^{\nu} \cap [\, h \,]^{\nu} && \text{(I.H.)} \\
= \;& [\, g \wedge h \,]^{\nu} && \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = Q \;\rightarrow\; g \,|\, h$.

$$\begin{aligned}
& \mathrm{im}(Q \;\rightarrow\; g \,|\, h)\langle \nu \rangle \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; Q \;\rightarrow\; g \,|\, h \; \|^{\nu}) && \text{(Def. 3.6.1)} \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap (\text{if } Q\langle \nu \rangle \text{ then } \| \; g \; \|^{\nu} \text{ else } \| \; h \; \|^{\nu})) && \text{(Def. 3.2.2)} \\
= \;& \text{if } Q\langle \nu \rangle \text{ then } \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g \; \|^{\nu}) \text{ else } \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; h \; \|^{\nu})) && \text{(Set Algebra)} \\
= \;& \text{if } Q\langle \nu \rangle \text{ then } [\, g \,]^{\nu} \text{ else } [\, h \,]^{\nu} && \text{(I.H.)} \\
= \;& [\, Q \;\rightarrow\; g \,|\, h \,]^{\nu} && \text{(Def. 3.6.4)}
\end{aligned}$$

Case $f = g(u := Q)$.

$$\begin{aligned}
& \mathrm{im}(g(u := Q))\langle \nu \rangle \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap \| \; g(u := Q) \; \|^{\nu}) && \text{(Def. 3.6.1)} \\
= \;& \mathrm{last}(\mathrm{tr}(\mathcal{K}_{\mathbf{C}}) \cap (\| \; g \; \|^{\nu[u \mapsto Q\langle \nu \rangle]})) && \text{(Def. 3.2.2)} \\
= \;& [\, g \,]^{\nu[u \mapsto Q\langle \nu \rangle]} && \text{(I.H.)} \\
= \;& [\, g(u := Q) \,]^{\nu} && \text{(Def. 3.6.4)}
\end{aligned}$$

$\square$

A.4 Abstract Model Checking

Lemma (A.4)**.** *Abstract simulation terminates and is monotonic with respect to the value of each recursion variable, i.e.* $\lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto U]}$ *is monotonic with respect to* $\sqsubseteq^{\forall\mathcal{V}}$ *as a function of* U*.*

Proof. Proof is by induction, ordering first by the number of fixed-points in f, and secondly by the length of f:

Case $f = \mathbb{tt}, \mathbb{ff}, \mathsf{n}, \neg\mathsf{n}$ or W. Then $\lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto U]}$ trivially terminates and is constant with respect to U, so is monotonic.

Case $f = g \vee h, g \wedge h, \mathbf{Y}g$. Then $\lfloor f \rfloor^{\nu}_{\sigma[Z \mapsto U]}$ terminates and is monotonic by the induction hypothesis and monotonicity of $\sqcup, \sqcap$ and $\mathrm{post}^{\sharp}$ respectively.

Case $f = \mu W \,.\, g$. By the induction hypothesis, $\lfloor g \rfloor^{\nu}_{\sigma[W \mapsto S]}$ is monotonic with respect to S. Therefore by the Knaster-Tarski theorem, the abstract simulation of $\mu W \,.\, g$,

$$\text{fix}\ (\lambda S \,.\, \lfloor f \rfloor_{\tau[W \mapsto S]})\ (\lambda \nu' . \bot) \tag{A.3}$$

must reach a fixed-point in our finite abstract domain, which equals

$$\bigsqcup_{n \geq 0} \lfloor \mu^n W.g \rfloor^{\nu}_{\sigma[Z \mapsto U]}$$

This is the least value greater than $\lfloor \mu^n W \,.\, g \rfloor^{\nu}_{\sigma[Z \mapsto U]}$ for any n. Suppose $U \subseteq^{\forall\mathcal{V}} U'$. Then by the induction hypothesis,

$$\lfloor \mu^n W \,.\, g \rfloor^{\nu}_{\sigma[Z \mapsto U]} \ \subseteq^{\forall\mathcal{V}}\ \lfloor \mu^n W \,.\, g \rfloor^{\nu}_{\sigma[Z \mapsto U']}$$

for any n. Therefore,

$$\lfloor \mu^n W \,.\, g \rfloor^{\nu}_{\sigma[Z \mapsto U]} \ \subseteq^{\forall\mathcal{V}}\ \lfloor \mu^n W \,.\, g \rfloor^{\nu}_{\sigma[Z \mapsto U']} \ \subseteq^{\forall\mathcal{V}}\ \bigsqcup_{n \geq 0} \lfloor \mu^n W.g \rfloor^{\nu}_{\sigma[Z \mapsto U']}$$

for any n. So

$$\bigsqcup_{n \geq 0} \lfloor \mu^n W.g \rfloor^{\nu}_{\sigma[Z \mapsto U]} \ \subseteq^{\forall\mathcal{V}}\ \bigsqcup_{n \geq 0} \lfloor \mu^n W.g \rfloor^{\nu}_{\sigma[Z \mapsto U']}$$

as required.

Case $f = Q \ \rightarrow\ g \mid h$ or $g(u := Q)$ By the induction hypothesis, abstract simulation of the sub-formulas g and h is monotonic. Therefore by Lemmas A.1.1, and A.1.2, respectively, abstract simulation of f is also monotonic. □

Appendix B

Grammar for Assertion Programs

This appendix presents a grammar for assertion programs, using the following BNF-style conventions:

- Non-terminals are in *italics* and all other symbols are terminals, except:
- $\{ x \}$ denotes zero or more occurrences of x.
- $x \mid y$ means one of either x or y.

The syntax takes the form:

$$\begin{array}{lcl} \mathit{literal} & ::= & \mathrm{true} \ \mid \ \mathrm{false} \\ & \mid & \mathit{integer\_literal} \end{array}$$

$$\begin{array}{lcl} \mathit{type} & ::= & \mathit{identifier} \\ & \mid & \mathbf{bool} \\ & \mid & \mathbf{int} \ (\ \mathit{integer\_literal} \ ) \\ & \mid & \mathit{type} \ \{ \ [\ \mathit{type} \ ] \ \} \end{array}$$

$$\begin{array}{lcl} \mathit{const\_decl} & ::= & \mathbf{type} \ \mathit{identifier} \ = \ \mathit{type} \\ & \mid & \mathbf{const} \ \mathit{identifier} \ = \ \mathit{literal} \end{array}$$

$$\begin{array}{lcl} \mathit{model\_decl} & ::= & \mathbf{var} \ \mathit{identifier} \ : \ \mathit{type} \end{array}$$

$$\begin{array}{lcl} \mathit{interface\_decl} & ::= & \mathbf{node} \ \mathit{identifier} \\ & \mid & \mathbf{node} \ \mathit{identifier} \ [\ \mathit{integer\_literal} \ : \ \mathit{integer\_literal} \ ] \end{array}$$

$$\begin{array}{lcl} \mathit{lexpr} & ::= & \mathit{identifier} \ \{ \ [\ \mathit{expr} \ ] \ \} \end{array}$$

$$\mathit{infix\_operator} ::= \wedge \mid \vee \mid = \mid + \mid - \mid < \mid \leq \mid > \mid \geq$$

$$\mathit{args} ::= \mathit{expr}\ \{\ , \mathit{expr}\ \}$$

$$\begin{array}{lll}
\mathit{expr} & ::= & \mathit{literal} \\
 & \mid & \mathit{lexpr}\ \{\ [\ \mathit{integer\_literal}\ :\ \mathit{integer\_literal}\ ]\ \} \\
 & \mid & \neg\ \mathit{expr} \\
 & \mid & \textbf{last}\ (\ \mathit{expr}\ ) \\
 & \mid & \mathit{expr}\ \mathit{infix\_operator}\ \mathit{expr} \\
 & \mid & \mathit{expr}\ \rightarrow\ \mathit{expr}\ \text{‘}\mid\text{’}\ \mathit{expr} \\
 & \mid & \textbf{forall}\ \mathit{identifier}\ :\ \mathit{type}\ .\ \mathit{expr} \\
 & \mid & \textbf{exists}\ \mathit{identifier}\ :\ \mathit{type}\ .\ \mathit{expr} \\
 & \mid & \mathit{expr}\ [\ \mathit{expr}\ ] \\
 & \mid & \mathit{identifier}\ (\ \mathit{args}\ )
\end{array}$$

$$\begin{array}{lll}
\mathit{stmt} & ::= & \textbf{skip} \\
 & \mid & \mathit{identifier}\ :=\ \mathit{expr} \\
 & \mid & \mathit{stmt}\ \|\ \mathit{stmt} \\
 & \mid & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
 & \mid & \textbf{for}\ \mathit{identifier}\ :\ \mathit{type}\ \textbf{do}\ \mathit{stmt} \\
 & \mid & \text{‘\{’}\ \mathit{stmt}\ \text{‘\}’}
\end{array}$$

$$\begin{array}{lll}
\mathit{assertion} & ::= & \mathit{expr}\ \Rightarrow\ \mathit{expr} \\
 & \mid & \textbf{forall}\ \mathit{identifier}\ :\ \mathit{type}\ .\ \mathit{assertion}
\end{array}$$

$$\begin{array}{lll}
\mathit{program} & ::= & \mathit{const\_decls} \\
 & & \textbf{model} \\
 & & \quad \{\ \mathit{model\_decls}\ \} \\
 & & \quad \mathit{stmt} \\
 & & \textbf{interface} \\
 & & \quad \{\ \mathit{interface\_decls}\ \} \\
 & & \quad \mathit{stmt} \\
 & & \textbf{assert} \\
 & & \quad \{\ \mathit{assertion}\ \}
\end{array}$$

Bibliography

[ABMS07] Sara Adams, Magnus Björk, Tom Melham, and Carl-Johan Seger. Automatic abstraction in Symbolic Trajectory Evaluation. In Baumgartner and Sheeran [BS07], pages 127–135.

[AFF$^+$02] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec temporal logic: A new temporal property-specification language. In Katoen and Stevens [KS02], pages 296–211.

[AJM$^+$00] Mark Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2000.

[AJS98] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In M. J. Irwin, editor, *ACM/IEEE Design Automation Conference (DAC '98)*, pages 538–541. IEEE Computer Society, 1998.

[AJS99] Mark Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Parametric representations of Boolean constraints. In Mary Jane Irwin, editor, *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC 99)*, pages 402–407. ACM Press, 1999.

[AMO99] Mark Aagaard, Thomas F. Melham, and John W. O'Leary. Xs are for trajectory evaluation, Booleans are for theorem proving. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 1999.

[AS95] Mark Aagaard and Carl-Johan H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In Richard L. Rudell, editor, *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '95)*, pages 7–10. IEEE Computer Society, 1995.

[Bar93] Janet E. Barnes. *A mathematical theory of synchronous communication*. PhD thesis, Oxford University, 1993.

[BB94] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC '94: Proceedings of the 31st Annual Conference on Design Automation*, pages 596–602. ACM Press, 1994.

[BBS91] Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In A. Richard Newton, editor, *Proceedings of the 28th Design Automation Conference*, pages 397–402. ACM Press, 1991.

[BC85] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer, 1985.

[BC86] Michael C. Browne and Edmund M. Clarke. SML: A high level language for the design and verification of finite state machines. In *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*. North-Holland Publishing, September 1986.

[BC96] Giorgio Brajnik and Daniel J. Clancy. Guiding and refining simulation using temporal logic. In *TIME '96: Proceedings of the 3rd Workshop on Temporal Representation and Reasoning (TIME'96)*, page 144. IEEE Computer Society, 1996.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In Richard C. Smith, editor, *DAC '90: Proceedings of the 27th ACM/IEEE Conference on Design Automation*, pages 46–51. ACM Press, 1990.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1994.

[Ben01] Bob Bentley. Validating the Intel Pentium 4 microprocessor. In Jan Rabaey, editor, *DAC '01: Proceedings of the 38th conference on Design automation*, pages 244–248. ACM Press, 2001.

[BFH$^+$92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BG00] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. In Catuscia Palamidessi, editor, *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, pages 168–182. Springer, 2000.

[BMAA01] Jayanta Bhadra, Andrew Martin, Jacob Abraham, and Magdy Abadir. Using abstract specifications to verify PowerPC custom memories by Symbolic Trajectory Evaluation. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME 2001*, volume 2144 of *Lecture Notes in Computer Science*, pages 386–402. Springer, 2001.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BS91] Randal E. Bryant and Carl-Johan H. Seger. Formal verification of digital circuits using symbolic ternary system models. In E. M. Clarke and R. P. Kurshan, editors, *Proc. 2nd International Conference on Computer-Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 33–43. Springer, 1991.

[BS01] Julian Bradfield and Colin Stirling. Modal logics and mu-calculi: an introduction. In A. Ponse J. Bergstra and S. Smolka, editors, *Handbook of Process Algebra*. Elsevier, North-Holland, 2001.

[BS07] Jason Baumgartner and Mary Sheeran, editors. *Formal Methods in Computer-Aided Design: FMCAD 2007*. IEEE Computer Society, 2007.

[CBM90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Proceedings of the international workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373. Springer, 1990.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[CC95] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In John Williams, editor, *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 170–181. ACM Press, 1995.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[Cho99] C-T. Chou. The mathematical foundation of Symbolic Trajectory Evaluation. In Nicolas Halbwahs and Doron Peled, editors, *Proc. 11th International Computer Aided Verification Conference*, volume 1633 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 1999.

[CHXY07] Yan Chen, Yujing He, Fei Xie, and Jin Yange. Automatic abstraction refinement for Generalized Symbolic Trajectory Evaluation. In Jason Baumgartner and Mary Sheeran, editors, *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 111–118. IEEE Computer Society, 2007.

[CJB79] William C. Carter, William H. Joyner Jr., and Daniel Brand. Symbolic simulation for correct machine design. In *DAC '79: Proceedings of the 16th Conference on Design automation*, pages 280–286. IEEE Computer Society, 1979.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead.*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001.

[CS00] K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and verification system, 2000.

[Cyr03] David A. Cyrluk. Proof methodologies for processor verification. Technical report, Stanford University, 2003.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525. IEEE Computer Society, 1992.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[dMOS03] Leonardo de Moura, Sam Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev. 2), SRI International, August 2003.

[Eme90] E. Allen Emerson. *Temporal and Modal Logic*. MIT Press, 1990.

[ER05] Kousha Etessami and Sriram K. Rajamani, editors. *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[FW93] M. Fisher and M. Wooldridge. Executable temporal logic for distributed AI. In K. Sycara, editor, *Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence (IWDAI93)*, pages 131–142, 1993.

[Gab87] Dov M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Temporal Logic in Specification*, pages 409–448. Springer, 1987.

[GB04] Amit Goel and Randal E. Bryant. Symbolic simulation, model checking and abstraction with partially ordered Boolean functional vectors. In Rajeev Alur and Doron Peled, editors, *16th International Conference in Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 255–267. Springer, 2004.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Springer, 2002.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Goe04] Amit Goel. *A Unified Framework for Symbolic Simulation and Model Checking*. PhD thesis, Carnegie Mellon University, 2004.

[Gor95] Michael J. C. Gordon. The semantic challenge of Verilog HDL. In Moshe Y. Vardi and Dexter Kozen, editors, *Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 136–145. IEEE Computer Society, 1995.

[Gor97] Mike Gordon. Synthesizable verilog: Syntax and semantics. University of Cambridge VFE Project, 1997.

[GP02] Elsa L. Gunter and Doron Peled. Temporal debugging for concurrent systems. In Katoen and Stevens [KS02], pages 431–444.

[GRS00] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.

[GS90] Mark R. Greestreet and Jorgen Straunstrup. Synchronized transitions. In Jorgen Straunstrup, editor, *Formal Methods for VLSI Design*, pages 71–128. North-Holland/Elsevier, 1990.

[GSY07] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 3-valued circuit SAT for STE with automatic refinement. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007*, volume 4762 of *Lecture Notes in Computer Science*, pages 457–473. Springer, 2007.

[HA00] James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In L. Miguel Silveira, Srinivas Devadas, and Ricardo Augusto da Luz Reis, editors, *Proceedings of the IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration*, pages 595–619. Kluwer, B.V., 2000.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[Haz96] Scott Hazelhurst. *Compositional Model Checking of Partially Ordered State Spaces*. PhD thesis, University of British Columbia, 1996.

[HB95] Ramin Hojati and Robert K. Brayton. Automatic datapath abstraction in hardware systems. In Pierre Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 98–113. Springer, 1995.

[HCY03a] Alan J. Hu, Jeremy Casas, and Jin Yang. Efficient generation of monitor circuits for GSTE assertion graphs. In Andreas Keuhlmann and Hidetoshi Onodera, editors, *IEEE/ACM International Conference on Computer-Aided Design*, pages 154–160. IEEE Computer Society, 2003.

[HCY03b] Alan J. Hu, Jeremy Casas, and Jin Yang. Reasoning about GSTE assertion graphs. In Daniel Geist and Enrico Tronci, editors, *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2003.

[HGD95] Hardi Hungar, Orna Grumberg, and Werner Damm. What if model checking must be truly symbolic. In Paolo Camurati and Hans Eveking, editors, *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95*, volume 987 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1995.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language lustre. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.

[HMN01] Yoav Hollander, Matthew Morley, and Amos Noy. The e language: A fresh separation of concerns. In Wolfgang Pree, editor, *TOOLS '01: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 41–50. IEEE Computer Society, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HR01] Klaus Havelund and Grigore Rosu. Testing linear temporal logic formulae on finite execution traces. Technical report, 2001.

[Jai97] Alok Jain. *Formal Hardware Verification by Symbolic Trajectory Evaluation*. PhD thesis, Carnegie Mellon University, 1997.

[JGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doran A. Peled. *Model Checking*. MIT Press, 1999.

[KA00] Roope Kaivola and Mark Aagaard. Divider circuit verification with model checking and theorem proving. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 338–355. Springer, 2000.

[Kah87] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[Kai05] Roope Kaivola. Formal verification of Pentium 4 components with symbolic simulation and inductive invariants. In Etessami and Rajamani [ER05], pages 170–184.

[KDG95] Peter Kelb, Dinnis Dams, and Rob Gerth. Practical symbolic model checking of the full μ-calculus using compositional abstractions. Technical Report Computing Science Report 95/31, Technical University of Eindhoven, 1995.

[KK97] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In Ellen J. Yoffa, Giovanni De Micheli, and Jan M. Rabaey, editors, *Design Automation Conference*, pages 263–268. ACM Press, 1997.

[KS02] Joost-Pieter Katoen and Perdita Stevens, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002*, volume 2280 of *Lecture Notes in Computer Science*. Springer, 2002.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LMS02] Francois Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal logic with forgettable past. In Gordon Plotkin, editor, *Proc. IEEE Symp. Logic in Computer Science (LICS'2002)*, pages 383–392. IEEE Computer Society, 2002.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In Rohit Parikh, editor, *Proceedings of the Conference on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, London, UK, 1985. Springer.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking — An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[McM99] Ken L. McMillan. The SMV language. Technical report, Cadence, March 1999.

[Mil72] Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford University, Stanford, CA, USA, 1972.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[Mil85] George J. Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Trans. Program. Lang. Syst.*, 7(2):270–298, 1985.

[MJ02] Thomas F. Melham and Robert B. Jones. Abstraction by symbolic indexing transformations. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2002.

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, 1986.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 1–2. ACM Press, 1987.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, NY, USA, 1992.

[Nav97] Zainalabedin Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, 1997.

[NHY04] Kelvin Ng, Alan J. Hu, and Jin Yang. Generating monitor circuits for simulation-friendly GSTE assertion graphs. In Ed Grochowski and Tom Dillinger, editors, *22nd IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD 2004)*, pages 409–416. IEEE Computer Society, 2004.

[NJB97] Kyle L. Nelson, Alok Jain, and Randal E. Bryant. Formal verification of a superscalar execution unit. In Ellen J. Yoffa, Giovanni De Micheli, and Jan M. Rabaey, editors, *DAC '97: Proceedings of the 34th Annual Conference on Design Automation*, pages 161–166, New York, NY, USA, 1997. ACM Press.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Decision procedures for SAT, SAT modulo theories and beyond. The BarcelogicTools. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005*, volume 3835 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2005.

[OZGS99] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating point hardware. *Intel Technology Journal*, 3(1):1–14, 1999.

[PLC94] P.T. Breuer, L.S. Fernández, and C. Delgado Kloos. Clean formal semantics for VHDL. In *Proceeding of the European Design and Test Conference*, pages 641–647. IEEE Computer Society, 1994.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[PRBA97] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir. Formal verification of content addressable memories using Symbolic Trajectory Evaluation. In Ellen J. Yoffa, Giovanni De Micheli, and Jan M. Rabaey, editors, *Proceedings of the 34th Annual Conference on Design Automation*, pages 167–172. ACM Press, 1997.

[Pre04] Mila Dalla Preda. Completeness refinement in abstract Symbolic Trajectory Evaluation. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004*, volume 3148 of *Lecture Notes in Computer Science*, pages 38–52. Springer, 2004.

[PSL05] IEEE standard for property specification language (PSL). *IEEE Std 1850-2005*, pages 1–143, 2005.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming, 5th Colloquium*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[RC05] Jan-Willem Roorda and Koen Claessen. A new SAT-based algorithm for Symbolic Trajectory Evaluation. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, CHARME 2005*, volume 3725 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2005.

[RC06a] Jan-Willem Roorda and Koen Claessen. Explaining Symbolic Trajectory Evaluation by giving it a faithful semantics. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Proceedings of the First International Computer Science Symposium in Russia, (CSR 2006)*, volume 3967 of *Lecture Notes in Computer Science*, pages 555–566. Springer, 2006.

[RC06b] Jan-Willem Roorda and Koen Claessen. SAT-based assistance in abstraction refinement for Symbolic Trajectory Evaluation. In Thomas Ball and Robert B. Jones, editors, *Proc. of Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, August 2006.

[RHKR00] Jurgen Ruf, Dirk W. Hoffmann, Thomas Kropf, and Wolfgang Rosenstiel. Simulation based validation of FLTL formulas in executable system descriptions. In R. Seepold, editor, *Proceedings of Forum on Design Languages (FDL 2000)*. IEEE Computer Society, 2000.

[RLSC05] Ricardo Rocha, Ricardo Lopes, Fernando M. A. Silva, and Vítor Santos Costa. Impact: Innovative models for prolog with advanced control and tabling. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 416–417. Springer, 2005.

[RMC94] E. Rutten, E. Marchand, and F. Chaumette. The sequencing of data flow tasks in SIGNAL: application to active vision in robotics. In *Proceedings Sixth Euromicro Workshop on Real Time Systems*, pages 80–5. IEEE Computer Society, 1994.

[SB95] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design: An International Journal*, 6(2):147–189, March 1995.

[SC95] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, 1995.

[Sch03] Tom Schubert. High level formal verification of next-generation microprocessors. In Ian Getreu, Limor Fix, and Luciano Lavagno, editors, *Proceedings of the 40th conference on design automation*, pages 1–6. ACM Press, 2003.

[Sch07] David A. Schmidt. Extracting program logics from abstract interpretations defined by logical relations. *Electr. Notes Theor. Comput. Sci.*, 173:339–356, 2007.

[SH97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[Sha00] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In Catuscia Palamidessi, editor, *CONCUR'00: Concurrency Theory*, number 1877 in Lecture Notes in Computer Science, pages 1–16. Springer, aug 2000.

[SJ92] Carl-Johan H. Seger and Jeffrey J. Joyce. A mathematically precise two-level formal hardware verification methodology. Technical Report TR-92-34, University of British Columbia, 1992.

[SJO$^+$05] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[Smi05] Edward Smith. A method for generation of GSTE assertion graphs. In Robert Jones, editor, *Tenth IEEE International High-Level Design Validation and Test Workshop*, volume 9, pages 160–167. IEEE Computer Society, 2005.

[Smi07] Edward Smith. A logic for GSTE. In Baumgartner and Sheeran [BS07], pages 119–126.

[SSTV04] Roberto Sebastiani, Eli Singerman, Stefano Tonetta, and Moshe Y. Vardi. GSTE is partitioned model checking. In Rajeev Alur and Doron Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV) 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2004.

[Sti92] Colin Stirling. Modal and temporal logics. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science (vol. 2): background: computational structures*, pages 477–563. Oxford University Press, Inc., New York, NY, USA, 1992.

[STV05] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: On hybrid approaches for ltl symbolic model checking. In Etessami and Rajamani [ER05], pages 350–363.

[TG06] Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for Symbolic Trajectory Evaluation. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2006.

[TM91] Donald E. Thomas and Philip R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, 1991.

[VB98] Miroslav N. Velev and Randal E. Bryant. Efficient modeling of memory arrays in symbolic ternary simulation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 1998.

[YG02] Jin Yang and Amit Goel. GSTE through a case study. In Lawrence Pileggi and Andreas Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 534–541. ACM Press, 2002.

[YGT05] Jin Yang, Rajnish Ghughal, and Andreas Tiemeyer. Industrial scale formal verification using concurrent GSTE. In Ting-Ao Tang and Yumei Huang, editors, *International Conference on ASIC (ASICON '05)*, pages 18–32. IEEE Computer Society, 2005.

[YS00] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation. Technical report, Intel, 2000.

[YS02] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In Mark Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2002.

[YS03] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):345–353, June 2003.

[YS04] Jin Yang and Carl-Johan H. Seger. Compositional specification and model checking in GSTE. In Rajeev Alur and Doron Peled, editors, *16th International Conference in Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 216–228. Springer, 2004.

[YT00] Jin Yang and Andreas Tiemeyer. Lazy symbolic model checking. In Giovanni De Micheli, editor, *Proceedings of the 37th conference on Design automation*, pages 35–38. ACM Press, 2000.

[YYHS05] Guowu Yang, Jin Yang, William N. N. Hung, and Xiaoyu Song. Implication of assertion graphs in GSTE. In Ting-Ao Tang, editor, *Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 1060–1063. ACM Press, 2005.

[YYSX06] Guowu Yang, Jin Yang, Xiaoyu Song, and Fei Xie. Maximal models of assertion graph in GSTE. In Jinyi Cai, S. Barry Cooper, and Angsheng Li, editors, *Theory and Applications of Models of Computation, Third International Conference, TAMC 2006*, volume 3959 of *Lecture Notes in Computer Science*, pages 684–693. Springer, 2006.