New Directions in Refinement Checking

by

Sriram K. Rajamani

B. Eng. (Anna University, Guindy College of Engineering, Madras, India) 1991
M. S. (University of Virginia, Charlottesville) 1992

A dissertation submitted in partial satisfaction of the requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Thomas A. Henzinger, Chair
Professor Robert K. Brayton
Professor John Steel

1999
The dissertation of Sriram K. Rajamani is approved:

_________________________  _______________________
Chair                                  Date

_________________________  _______________________
                                     Date

_________________________  _______________________
                                     Date

University of California at Berkeley

1999
New Directions in Refinement Checking

Copyright 1999

by

Sriram K. Rajamani
Abstract

New Directions in Refinement Checking

by

Sriram K. Rajamani
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Thomas A. Henzinger, Chair

Formal design verification is a methodology for detecting logical errors in systems. In formal design verification, the designer describes a system in a language with a mathematical semantics, and then the system description is analyzed against various correctness requirements. The paradigm is called model checking when the analysis is performed automatically by exhaustive state-space exploration. A correctness requirement is usually specified either as a formula expressed in temporal logic, or as an abstract design expressed in the system’s description language. If the requirement is specified as an abstract design, the verification problem is called refinement checking. This thesis extends the state-of-art by increasing both the class, and the size of systems on which automatic and semi-automatic refinement checking are viable.

Refinement checking problems are formulated in the form $Impl \preceq Spec$, where $Impl$ is a description of the system, $Spec$ is the requirement, and $\preceq$ is a preorder over system descriptions. Depending on the properties we ascribe to the $\preceq$ preorder, we get different notions of refinement. Three orthogonal property classes characterize the refinement preorder, namely, (1) linear and branching views of time, (2) finite and fair views of system behaviors, and (3) spatial and temporal abstraction of the requirement. We develop the theory of fair branching refinements, and give an efficient algorithm to check such refinements. We propose a novel and efficient algorithm to check refinements involving temporal abstractions.

Refinement checking algorithms usually explore and analyze the state space of the system. As we seek to enhance the applicability of model checking to complex designs,
we are faced with the so-called *state-explosion* problem: the size of the state space grows exponentially with the size of the system description, making exhaustive state-space exploration infeasible. Scalable approaches to refinement checking make use of the compositional structure of both implementation and specification, and divide the verification task at hand into simpler subtasks. Each such subtask involves checking if an implementation-component refines its corresponding specification-component. Discharging such subtasks requires assumptions about inputs to the implementation-component. These assumptions are taken into account by the *assume-guarantee* approach, which uses the specification for the inputs as assumptions on the inputs (the apparent circularity in such proofs is resolved by an induction over time.) We prove soundness of the assume-guarantee rule when applied to branching refinements (both finite and fair cases.) We propose a new assume-guarantee proof rule for checking refinements of implementations against requirements that are both spatially and temporally abstract.

The techniques have been implemented in a new verification tool called MoCHA. MoCHA has been used successfully to detect and fix elusive bugs in a signal-processing chip that has 6 million transistors.

Professor Thomas A. Henzinger
Dissertation Committee Chair
to Rajni ...
# Contents

List of Figures vi
List of Tables vii

1 Introduction 1
   1.1 Refinement checking ........................................ 1
   1.2 Thesis overview ........................................... 5

2 Preliminaries 7
   2.1 Transition constraints ...................................... 7
   2.2 Kripke structures ........................................... 10
   2.3 Modules .................................................. 12
   2.4 Parallel composition ...................................... 16
   2.5 Reachability and invariant checking ....................... 17

3 Refinement 20
   3.1 Linear refinement .......................................... 20
   3.2 Branching refinement ...................................... 24
   3.3 Projection refinement ..................................... 28
   3.4 State explosion and assume-guarantee reasoning ........ 29
   3.5 Assume-guarantee rule for branching refinement ........ 31

4 Fair Branching Refinement 39
   4.1 Fairness and fair refinement .............................. 39
   4.2 Definitions .............................................. 44
   4.3 Init simulations .......................................... 52
   4.4 Checking fair simulation .................................. 56
   4.5 A logical characterization of fair simulation ........... 61
   4.6 Fair similarity quotients ................................ 65
   4.7 Assume-guarantee rule for fair simulation .............. 66

5 Refinement with Sampling 69
   5.1 Motivation for sampling ................................... 69
   5.2 The *Sample* operator ................................... 75
CONTENTS

5.3 Symbolic exploration of hierarchical state spaces ............... 79
  5.3.1 Explicit definition of hierarchical constraints ............... 80
  5.3.2 Efficient computation with implicit update predicates ....... 81
5.4 Assume-guarantee refinement with Sample ............................. 82
5.5 Refinement proof ................................................. 85

6 Verification of the VGI Processor ................................. 88
  6.1 Overview ......................................................... 88
  6.2 Design implementation ........................................... 92
  6.3 Design specification ............................................. 95
  6.4 Refinement proof ................................................ 99
    6.4.1 Step 1: Network of compute processors ..................... 100
    6.4.2 Step 2: Single processor configuration ................. 101
  6.5 Discussion ...................................................... 107

7 Conclusion ......................................................... 110

Bibliography .......................................................... 113

A VGI - Instruction Set Architecture Specification .................. 120

B VGI - Queue Specification ........................................ 130

C VGI - Abstract Constraints and Auxiliary Variables ............. 135
# List of Figures

2.1 Lazy Counter .................................................. 9
2.2 Run-tree and trace-tree ........................................ 10
2.3 Blocking constraint ........................................... 11
2.4 Lazy Counter Module ......................................... 15

3.1 Specification of an arbiter .................................... 22
3.2 Implementation of arbiter ..................................... 23
3.3 Tree merging, .................................................. 27
3.4 Specification of Sender ....................................... 32
3.5 Specification of Receiver ..................................... 33
3.6 Implementation of the sender ................................ 34
3.7 Implementation of Receiver .................................. 35
3.8 A partial trace-tree corresponding to the trace-tree in Figure 2.2 ....... 36

4.1 Incorrect implementation of arbiter .......................... 40
4.2 Fair simulation is stronger than $\exists$-simulation ............. 42
4.3 $\forall$-simulation is stronger than fair simulation ............ 43
4.4 Fair simulation does not have memoryless strategy .......... 50
4.5 A sat-tree of $I$ (Figure 4.2) for $\varphi$ ..................... 64

5.1 GCD Specification ............................................ 72
5.2 GCD Implementation ........................................... 73

6.1 VGI processor configuration with three input and two output queues .... 93
6.2 Specification module for refinement check .................... 95
6.3 Specification module with signaling details .................. 96
6.4 ISA with two input and two output queues .................... 96
6.5 Implementation of a queue .................................... 102
6.6 Abstract timing constraints for the queue .................... 102
6.7 Abstract constraint for $send$ ................................ 104
6.8 Abstract constraint for $stallpipe$ ............................ 106
6.9 Abstract constraint for $abux_r$ ................................ 107
List of Tables

2.1 Module variables ............................................................. 13
4.1 Properties of Init Simulations ........................................... 55
6.1 Component modules of ISA ............................................. 97
Acknowledgements

I have had good and bad times at Berkeley, during the past four years. Fortunately, I have had mostly good times, and I want to thank the people who made it that way.

My biggest fortune was in having Professor Tom Herzinger as my advisor. Tom truly inspires his students. He taught me, listened to me, trusted me, and guided me throughout my Ph.D. Tom's influence can be found in every chapter of this thesis. I have learnt a lot from him about choosing problems, writing with precision, talking with clarity, and much more.

My second biggest fortune was in having Shaz Qadeer as my colleague during these years. Right from the day I met Shaz, he has been an excellent research partner, in addition to being a good friend. Be it battling with a tough theoretical problem, or be it hacking code, Shaz was always around to discuss, talk, and contribute. Without Shaz, doing this thesis would not have been so much fun.

My third biggest fortune was in meeting and working with Professor Orna Kupferman and Professor Rajeev Alur. Both Orna and Rajeev were visiting Berkeley during my first summer at Berkeley and the following year. I worked on my first major research problem (Fair Simulation) with Orna at that time. Orna's knowledge in automata theory, and her energy in research have always been a source of inspiration. Rajeev taught me several intricacies about partial-order methods, suggested several ways of pursuing research with the "next" operator, and was a constant source of new ideas.

Professor Bob Brayton took the time to listen to several of my research ideas, and offered constructive feedback on several occasions. I thank him for guidance over the years, and for serving on my thesis committee. I thank Professor John Steel for teaching my first graduate course in mathematical logic, and for serving on my thesis committee. Professor Alistair Sinclair was brave enough to show up for my thesis qualifying examination, days after having an emergency surgery.

A significant amount of my research effort at Berkeley was spent in designing and developing Mocha. It is an understatement to say that these efforts would have been crippled without Freddy Mang. It was a pleasure to work with some one as quick and as bright as Freddy.

Berkeley provided the opportunity to collaborate with several bright people. Shaz and I were battling last summer with a non-constructive proof of the assume-guarantee
rule for simulation. Serdar Tasiran came up with a beautiful constructive proof of this theorem. Xiaojun Liu's persistence and thoroughness in translating the VGI processor model to Reactive Modules was a necessary pre-condition to its verification in MoCHA. Luca de Alfaro helped me understand several facts about the "next" operator.

Andreas Kuehlman provided several real-world insights during his sabbatical at the CAD group. Ken McMillan made himself available several times to talk about compositional verification. Adnan Aziz, Tom Shiple, Rajeev Ranjan, Yuji Kukimoto, Rajeev Murgai, and Gitanjali Swamy taught me several things during my early years here. Amit Mehrotra offered walking-and-talking-encyclopedia services in LaTeX, fixed the Linux kernel in my laptop multiple times, and helped me out in several other ways. Brad Krebs helped keep my laptop and desktop operational, with incredible enthusiasm. Mukul Prasad, Lixin Su, Bassam Tabbara, Janet Wang, Marco Sgroi and Gurmeet Manku braved the CAD group preliminary exam with me, and helped make that a pleasant experience. Pramod Viswanath has been a good friend, and an excellent biking partner, without whom I would have never been able to bike up Spruce. Several CAD group people made my stay enjoyable — Sriram Krishnan, Sunil Khatri, Luca Carloni, Rupak Majumdar, Wilsin Gosti, Ravi Gunturi, Ben Horowitz, and Subarnarekha Sinha to name a few. Cafe Brewed Awakening provided space, tea, a plug for my laptop, and the ambiance to think and work.

During portions of my time at Berkeley, I was also working part-time at Xilinx Inc, San Jose. Several people at Xilinx made this possible. Jon Frankle provided support and encouragement, and good advice. Jon's attitude towards tools and algorithms has influenced me. I also thank Kamal Chaudhary, and Doug Wieland for supporting me in this process. Giriraj Devaraj hosted me, fed me several times, and made my trips to the south bay enjoyable.

My undergraduate teachers Sridar Kandaswamy and Seshasayee went above and beyond their call of duty, to help me get into graduate school. I owe them a lot!

My parents, and my brother, Kumar, were instrumental in my coming here for my graduate education. Without them, nothing would have been possible. Jothi and Meena supported me with their love and affection. Rajesh, Indu and Mami provided several opportunities to relax, and reminded me that life exists outside of Computer Science. Last but not the least, I want to thank Rajni for helping me keep my sanity in the midst of chaos. Rajni supported me in every possible way, and was always there to listen and help. Words cannot express my thanks to her.
Chapter 1

Introduction

Formal design verification is a methodology for detecting logical errors in systems. System testing is usually performed by “running” the system, and observing how it behaves. For instance, if one writes a C program, testing the program usually involves compiling and running the program with representative input data, and observing the output of the program. Formal verification, on the other hand, tries to establish properties of the system, and find bugs in the system, using mathematical techniques. We are concerned with the formal verification of concurrent systems with reactive components. *Concurrency* implies that the system has multiple agents executing simultaneously, and interacting with each other. *Reactivity* implies that each component has an ongoing interaction with its environment—the component and its environment exchange information with each other, and evolve over time. Examples of concurrent reactive systems are operating systems, digital hardware, and network protocols. Such systems should be contrasted with transformational systems. *Transformational systems* take an input and compute an output—for example, a C program that takes two integers $a$ and $b$ as inputs, and computes the greatest common divisor of $a$ and $b$ (However, one could also build a reactive system that takes several values for $a$ and $b$ over time, and computes the greatest common divisor for all of these over time. See Chapter 5.)

1.1 Refinement checking

In formal verification, the designer describes a system in a language with a mathematical semantics, and then the system description is analyzed against various correctness
requirements. The correctness requirements are also described in a language with mathematical semantics. Since we verify reactive systems, the correctness requirements usually state properties that involve time—for example, “at all time instants the bus is accessed by at most one processor”, or “whenever there is a request, there is a response at a later point in time.” Temporal logics [Pnu77] provide a formal, convenient way to state such correctness requirements. Another way to state such correctness requirements is to use the same language as the design, and state the requirement as an abstract design. For example, if we want to verify if a pipeline works correctly, the requirement can be specified as another design—its instruction set architecture (ISA.) If the requirement is specified as an abstract design, the verification problem is called refinement checking. There are two widely known techniques for doing refinement checking:

- **Theorem proving** involves writing a mathematical proof (in some logical framework, using deductive reasoning) that the system satisfies its requirement. Theorem provers are tools that help construct and check such proofs. There is usually a lot of manual effort in verifying designs using a theorem prover. However, there is no theoretical limit on the size and complexity of designs that could be verified using theorem proving.

- **Model checking** involves analyzing the system automatically, by exhaustive state-space exploration. Since the state spaces of systems are exponential in their descriptions, model checking, if used naively, does not scale for large systems.

In the past decade, model checking has evolved as a viable technique. Quite a few successful model checkers—SMV [McM93], VIS [BHSV+96], Murphi [DiI96], SPIN [HP96], COSPAN [HHK96]—have been developed in the academia and industry. We are interested in refinement checking using model checking. Note that refinement checking is a problem, and model checking is a technique.

Refinement checking problems are formulated in the form $\text{Impl} \preceq \text{Spec}$, where $\text{Impl}$ is a description of the system, $\text{Spec}$ is the requirement, and $\preceq$ is a refinement preorder over system descriptions (a preorder is a reflexive, transitive binary relation.) The assertion $\text{Impl} \preceq \text{Spec}$ is variously pronounced as “$\text{Impl}$ implements $\text{Spec}$,” or “$\text{Impl}$ refines $\text{Spec}$,” or “$\text{Spec}$ specifies $\text{Impl}$,” or “$\text{Spec}$ abstracts $\text{Impl}$.” Depending on the properties we ascribe to the $\preceq$ preorder, we get different notions of refinement. Three orthogonal property classes characterize the refinement preorder, namely, (1) linear and branching views of time, (2)
CHAPTER 1. INTRODUCTION

finite and fair views of system behaviors, and (3) spatial and temporal abstraction of the
requirement.

Linear and branching time. The linear time notion of refinement is trace containment. In
this case, Impl $\preceq$ Spec asserts that every sequence of inputs and outputs (called a
trace) that is possible for Impl is also possible for Spec (at the same time, the more
abstract specification Spec may allow some traces that are not realized by the more concrete
implementation Impl.) One could also state this equivalently using linear temporal logic
(LTL) [Pnu77]: Impl $\preceq$ Spec if every LTL property that holds in Spec also holds in Impl.
The complexity of checking trace containment is linear in the states of Impl and exponential
in the number of states of Spec. The problem has been shown to be PSPACE complete on
the states of the specification [SM73].

In top-down design, if system description Impl fleshes out detail that is left open in
system description Spec, there is a much tighter relation between Impl and Spec than trace
containment would indicate; namely, each implementation state of Impl corresponds to a
specification state of Spec. This tighter relation is captured mathematically by the notion
of a simulation relation, which leads to the branching notion of refinement —simulation.
Intuitively, Spec simulates Impl iff, starting from the initial states and continuing ad in-
finitum, every input-output pair of Impl can be matched by the same input-output pair in
Spec [Mil71]. Clearly, if Spec simulates Impl, then every trace of Impl is also a trace of
Spec. The converse is not true; that is, simulation is a stronger requirement than trace
containment. However, it has been said that trace containment without simulation is more
often than not due to coincidence rather than systematic design [Kur94]. An alternative,
but equivalent, definition of branching refinement is trace-tree containment. In this case,
Impl $\preceq$ Spec asserts that every tree of inputs and outputs (called a trace-tree) that is pos-
sible for Impl is also possible for Spec. One could also define simulation equivalently using
branching temporal logics like CTL, CTL* and $\mu$-calculus [Eme90]: Spec simulates Impl if
every property in the universal fragment of CTL, CTL*, and $\mu$-calculus that holds in Spec
also holds in Impl. The complexity of checking simulation is $O(m_i \cdot n_s + m_s \cdot n_i)$, where $n_i$
and $m_i$ are the number of states and transitions in Impl respectively, and $n_s$ and $m_s$ are
the number of states and transitions in Spec respectively [BP95, HHK95].

Finite and fair refinement. Refinement, as we have discussed so far, verifies only the safe
behaviors of implementations against specifications. It is well known that a safe refinement
implies “nothing bad happens” [Al93] (for example, “two processors cannot access the bus
CHAPTER 1. INTRODUCTION

at the same time.”) Sometimes, it is necessary to check refinements that imply “something good happens” [AL93] (for example, “whenever there is a request, it is eventually followed by a response.”) This is a classical problem in concurrency theory, and in its abstract version, involves reasoning about infinite behaviors. The standard solution is to enhance the system modeling language with fairness conditions, which classify infinite runs of the system into fair and unfair runs. With fairness conditions added to the specification and implementation models, we are concerned with fair refinements. As one would expect, fair refinements could be viewed from linear and branching perspectives. The linear framework of trace-containment generalizes naturally to fair trace-containment: Spec fairly trace-contains Impl iff it is possible to fairly generate by Spec every sequence of observations that can be fairly generated by Impl. Robustness with respect to LTL, and PSPACE-completeness extend to the fair case. It is not so obvious how to generalize the branching framework of simulation to account for fairness [SVW87, Saf88].

Spatial and temporal abstractions. Specifications are typically less detailed than the implementation. For example, the specification of an adder might simply state that the output is the sum of the two inputs, whereas the implementation might be a gate-level adder circuit, which operates at the detail of individual bits. In such cases, we say that the specification is spatially abstract with respect to the implementation. Nonetheless, common notions of correctness require specifications to operate in “lock-step” with the implementation: every possible computation step of the implementation must be matched by an admissible computation step of the specification. If the natural time scale of the specification is less detailed than that of the implementation—for example, if the gate-level adder requires several clock cycles to compute a sum, and its specification requires only one clock cycle—we say that the specification is temporally abstract with respect to the implementation.

Assume-guarantee refinement checking

Refinement checking algorithms usually explore and analyze the state space of the system. As we seek to enhance the applicability of model checking to complex designs, we are faced with the so-called state-explosion problem: the size of the state space grows exponentially with the size of the system description, making exhaustive state-space exploration infeasible. Scalable approaches to refinement checking make use of the compositional structure
of both implementation and specification, and divide the verification task at hand into simpler subtasks. Each such subtask involves checking if an implementation-component refines its corresponding specification-component. Discharging such subtasks requires assumptions about inputs to the implementation-component. These assumptions are taken into account by the assume-guarantee approach \cite{Sta85, CM88, CLM89, GL94, AL95, AH96, McM97}, which uses the specification for the inputs as assumptions on the inputs (the apparent circularity in such proofs is resolved by an induction over time.) If the specification is very abstract and lacks detail, it is usually harder to divide the refinement proof using assume-guarantee reasoning. With assume-guarantee reasoning, the difficulty in refinement checking shifts from the sizes of the involved state spaces to the “semantic gap” between implementation and specification. To bridge this gap, one writes abstract constraints that add detail to the specification and relate specification signals to implementation signals.

1.2 Thesis overview

This thesis extends the state-of-art by increasing both the class, and the size of systems on which automatic and semi-automatic refinement checking (based on model checking) are viable. The main contributions are listed below, along with references to the respective conference papers:

- We develop the theory of fair branching refinements, and give an efficient algorithm to check such refinements \cite{HKR97}.

- We propose a novel and efficient algorithm to check refinements involving temporal abstractions \cite{AHR98}.

- We prove soundness of the assume-guarantee rule when applied to branching refinements (both finite and fair cases) \cite{HQR98}.

- We propose a new assume-guarantee proof rule for checking refinements of implementations against requirements that are both spatially and temporally abstract \cite{HQR99}.

- We have implemented these techniques in a new verification tool called MOCHA \cite{AHM+98}. We describe a case study, where we have used MOCHA successfully to detect and fix elusive bugs in a signal-processing chip that has 6 million transistors \cite{HLQR99}.
CHAPTER 1. INTRODUCTION

This thesis is organized as follows. Transition constraints and reactive modules are our formal languages for describing systems [AH96]. We introduce them in Chapter 2. In Chapter 3 we formally define linear and branching views of safe refinements. We also propose and prove soundness of an assume-guarantee rule for branching refinement. In Chapter 4 we introduce fairness and fair refinements. In particular, we develop the theory of fair branching refinements, and give an efficient algorithm to check such refinements. In Chapter 5 we introduce sampling as a way of checking refinements involving temporal abstractions, and give an efficient algorithm to check such refinements. We also introduce a novel methodology for decomposing refinement proofs in the case of temporally and spatially abstract specifications. We describe an application of this rule to the verification of a large signal-processing chip in Chapter 6. We conclude and discuss open problems in Chapter 7.

Refinement checking tools

MOCHA is a model checker designed to explicitly support assume-guarantee refinement checking. We briefly mention two other tools that support refinement checking. COSPAN [HHK96] supports language-containment checking and homomorphism checking. Several techniques, such as localization reduction, are used in COSPAN to handle state-explosion. However, no explicit support for assume-guarantee reasoning is provided. The new version of SMV [McM99] supports refinement checking using assume-guarantee reasoning explicitly. However, MOCHA has some unique features for compositional refinement checking, like support for decomposing refinement proofs in the case of temporally and spatially abstract specifications.
Chapter 2

Preliminaries

We are interested in the formal verification of concurrent reactive systems. Reactive Modules is a formalism for the modular description of systems with heterogeneous components [AH96]. We generalize modules to transition constraints.

2.1 Transition constraints

A transition constraint (or constraint, for short) is a temporal constraint on a set of variables. The state of a transition constraint is determined by values of two kinds of variables, namely, observable variables and private variables. The state of a transition constraint changes in a sequence of rounds. The first round is called the initial round, and determines initial values for all variables. Each subsequent round is called an update round, and determines new values for all variables. A sequence of states such that the first state results from an initial round, and each successive state in the sequence results from the previous state as a result of an update round is called a run. A tree of states such that the root of the tree is a state that results from an initial round, and each state in the tree results from the state of its parent node as a result of an update round is called a run-tree. The projection of a run to the observable variables is called a trace, and the projection of a run-tree to the observable variables is called a trace-tree. The linear semantics of a transition constraint is its set of traces and the branching semantics is its set of trace-trees.

A transition constraint $C$ has a finite set of typed variables, denoted $X_C$. A state of a transition constraint $C$ is a valuation for all the variables $X_C$. The set of all states of $C$ is denoted by $\Sigma_C$. In each round, every variable $x$ has two values. The value of $x$ at the
beginning of the round is called the \textit{latched value}, and the value of \( x \) at the end of the round is called the \textit{updated value}. We use unprimed symbols, such as \( x \), to refer to latched values, and primed symbols, such as \( x' \), to refer to the corresponding updated values. Given a set \( X \) of unprimed symbols, we write \( X' \) for the set of corresponding primed symbols.

\textbf{Definition 2.1 [Transition constraint]} A transition constraint \( C \) consists of a declaration and a body. The declaration is a finite set \( X_C \) of typed variables, that is partitioned into the set \( \text{obs}X_C \) of observable variables and the set \( \text{priv}X_C \) of private variables. The body consists of an initial predicate \( I_C(X'_C) \), and an update predicate \( U_C(X_C, X'_C) \).

The initial predicate is a boolean function over the primed variables of \( C \). A state \( s \) of \( C \) is \textit{initial} if it satisfies \( I_C \). The set of all initial states of \( C \) is denoted by \( \Sigma^I_C \). The update predicate is a boolean function over both the primed and unprimed variables of \( C \). Given two states \( s \) and \( t \), we write \( s \rightarrow_C t \) if the \( U_C \) evaluates to \textit{true}, when its unprimed variables are assigned values from \( s \) and its primed variables are assigned values from \( t \).

\textbf{Runs and traces.} Let \( \sigma \) be the a finite or infinite sequence. We define \( |\sigma| \) to be equal to the length of \( \sigma \). If \( \sigma = \sigma_0, \sigma_1, \ldots, \sigma_n \) is finite, then \( |\sigma| = n \), otherwise \( |\sigma| \) is the least infinite cardinal \( \omega \). For all \( w \in \Sigma_C \), a \textit{w-run} of \( C \) is a finite or infinite sequence \( \pi = s_0, s_1, \ldots \) of states such that (1) \( s_0 = w \), and (2) for \( 0 \leq i < |\pi| \), we have \( s_i \rightarrow_C s_{i+1} \). A \textit{run} of \( C \) is a \( \hat{w} \)-run of \( C \) for some \( \hat{w} \in \Sigma^I_C \). An \textit{observation} of \( C \) is a valuation for \( \text{obs}X_C \). The set of all observations of \( C \) is denoted by \( \Gamma_C \). If \( s \) is a valuation to a set of variables, we use \( [s]_C \) to denote the valuation restricted to \( \text{obs}X_C \). For a state sequence \( \pi = s_0, \ldots, s_n \), we write \( [\pi]_C = [s_0]_C, \ldots, [s_n]_C \) for the corresponding observation sequence. If \( \pi \) is a run of \( C \), then the projection \( [\pi]_C \) is called a \textit{trace} of \( C \); in this case we say that the run \( \pi \) \textit{witnesses} the trace \( [\pi]_C \). Sometimes there is a need to effect a transition constraint only for a fixed number of update steps. We use \( C^\tau \) to denote the transition constraint with the same variables as \( C \), which restricts their valuations only up to \( \tau \) update rounds. Formally, a state sequence \( \pi = s_0, \ldots, s_n \) is a run of \( C^\tau \) if (1) \( n \leq \tau \) and \( \pi \) is a run of \( C \), or (2) \( n > \tau \) and \( s_0, \ldots, s_\tau \) is a run of \( C \). Note that if \( \tau < 0 \), then every state sequence is a run of \( C^\tau \).

\textbf{Run-trees and trace-trees.} A (finite or infinite) \textit{tree} is a set \( \tau \subseteq \mathbb{N}^* \) such that if \( x_m \in \tau \), for \( x \in \mathbb{N}^* \) and \( n \in \mathbb{N} \), then \( x \in \tau \) and \( x_m \in \tau \) for all \( 0 \leq m < n \). The elements of \( \tau \) represent nodes: the empty word \( \epsilon \) is the root of \( \tau \), and for each node \( x \), the nodes of the form \( x_m \), for \( n \in \mathbb{N} \), are the children of \( x \). The number of children of node \( x \) is denoted by \( \text{deg}(x) \). A tree \( \tau \) is finite if \( \tau \) is a finite set. The \textit{depth} of a node \( x \in \tau \) is defined inductively
constraint counter
  observable in: bool, out: bool
private count: (0..5)
init
  (count' = 0) & ~out'
update
  ((in' & ((count' = count + 1) | (count' = count))) |
  (~in' & (count' = count))) &
  (out' = ((count' = 0) & (count = 5)))
endconstraint

Figure 2.1: Lazy Counter

as follows: (1) the depth of $\epsilon$ is 0, and (2) if the depth of $x \in \tau$ is $d$, then the depth of
$x^n$ is $d + 1$. If $\tau$ is finite, then the depth of $\tau$ is defined as the maximum of depths over
all nodes of $\tau$. The nodes of $\tau$ with no children are called leaves of $\tau$. A path $\rho$ of $\tau$ is a
finite or infinite set $\rho \subseteq \tau$ of nodes that satisfies the following three conditions: (1) $\epsilon \in \rho$,
(2) for each node $x \in \rho$, there exists at most one $n \in \mathbb{N}$ with $xn \in \rho$, and (3) if $xn \in \rho$,
then $x \in \rho$. Given a set $S$, an $S$-labeled tree is a tuple $\langle \tau, \lambda \rangle$, where $\tau$ is a tree, $\lambda : \tau \rightarrow S$
is a node labeling function that maps each node of $\tau$ to an element in $S$. Then, every path
$\rho = \{\epsilon, n_0, n_0n_1, \ldots\}$ of $\tau$ generates a sequence $\Lambda(\rho) = \lambda(\epsilon) \cdot \lambda(n_0) \cdot \lambda(n_0n_1) \cdots$ in $S^* \cup S^\omega$.

For all $w \in \Sigma_C$, a $w$-run-tree $T$ of $C$ is a $\Sigma_C$-labeled tree $\langle \tau, \lambda \rangle$ such that $\lambda(\epsilon) = w$, and for all edges $\langle x, xn \rangle$ we have $\lambda(x) \rightarrow_C \lambda(xn)$. A run-tree of $C$ is a $w$-run of $C$ for some $w \in \Sigma_C$. Note that for any depth, $C$ has possibly none, one, or many run-trees. A trace-tree $T' = \langle \tau', \lambda' \rangle$ of $C$ is a $\Gamma_C$-labeled tree such that there is a run-tree of $T = \langle \tau, \lambda \rangle$ of $C$, and for every $x \in \tau$ we have $[\lambda(x)]_C = \lambda'(x)$. For brevity we say $T' = [T]_C$, and call $T$ a witness to $T'$.

Example. Figure 2.1 shows a transition constraint that describes a lazy counter. The counter has two boolean observable variables in and out. Though in is intended to be an input to the counter and out an output, the transition constraint does not make this distinction. The counter has a private variable count that counts from 0 through 5. The initial predicate asserts that count is set to 0 and out to false in the initial round. In
each update round, in can take values true or false. The counter may increment count if in is true. When count updates from 5 to 0, out is set to true. Otherwise, out is set to false. A state of counter is a triple \(\langle a, b, c \rangle\), where \(a, b, \) and \(c\) are valuations to \(\text{in}, \text{out}\) and count respectively. The initial states of counter have to satisfy the initial predicate. Thus, counter has two initial states \(\langle \text{true}, \text{false}, 0 \rangle\) and \(\langle \text{false}, \text{false}, 0 \rangle\).

The state-sequence \(\langle \text{true}, \text{false}, 0 \rangle \langle \text{true}, \text{false}, 1 \rangle \langle \text{true}, \text{false}, 2 \rangle \langle \text{true}, \text{false}, 3 \rangle \langle \text{true}, \text{false}, 3 \rangle \langle \text{true}, \text{false}, 4 \rangle \langle \text{true}, \text{false}, 5 \rangle \langle \text{true}, \text{true}, 0 \rangle \langle \text{true}, \text{false}, 0 \rangle \langle \text{true}, \text{false}, 1 \rangle\) is a run of counter. The corresponding trace is obtained by projecting each state in the sequence to its observation: \(\langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle \langle \text{true}, \text{false} \rangle\).

Figure 2.2 shows a run-tree \(T_1\), and a trace-tree \(T_2\) of constraint counter. Note that \(T_2 = [T_1]\)counter.

Note that a transition constraint could block i.e., it could reach a state from which no update round is feasible. For example, Figure 2.3 shows a constraint that blocks. In the initial state, \(x\) is set to true. From this state, no update round is feasible.

### 2.2 Kripke structures

A Kripke structure is a labeled state diagram, used to represent the state space of a constraint.
constraint block
  observable \( x : \text{bool} \)
  init
    \( x' \)
  update
    (\( \neg x \land x' \))
endconstraint

Figure 2.3: Blocking constraint

Definition 2.2 [Kripke Structure] A Kripke structure is a 5-tuple \( K = \langle \Gamma, W, \hat{W}, R, L \rangle \) with the following components:

- A set \( \Gamma \) of observations. Sometimes, we have a finite set \( P \) of propositions and \( \Gamma = 2^P \).
- A set \( W \) of states.
- A set \( \hat{W} \subseteq W \) of initial states.
- A transition relation \( R \subseteq W \times W \). If \( \langle s, t \rangle \in R \), then we write \( R(s, t) \).
- A labeling function \( L : W \to \Gamma \) that maps each state to an observation.

The structure \( K \) is deterministic if whenever \( R(w, w_1) \) and \( R(w, w_2) \) for \( w_1 \neq w_2 \), then \( L(w_1) \neq L(w_2) \). For all \( w \in W \), a \( w \)-run of \( K \) is a finite or infinite sequence \( \bar{w} = w_0 \cdot w_1 \cdot w_2 \cdots \) of states \( w_i \in W \) such that \( w_0 = w \) and \( R(w_i, w_{i+1}) \) for all \( i \geq 0 \). A run of \( K \) is a \( \hat{w} \)-run, for some initial state \( \hat{w} \in \hat{W} \). A trace of \( K \) is a finite or infinite sequence \( \gamma = \gamma_0 \cdot \gamma_1 \cdot \gamma_2 \cdots \) of observations \( \gamma_i \in \Gamma \) such that there is a run \( \bar{w} = w_0 \cdot w_1 \cdot w_2 \cdots \) of \( K \) and \( \gamma_i = L(w_i) \) for all \( i \geq 0 \); in this case we say that the run \( \bar{w} \) witnesses the trace \( \gamma \). We can similarly define run-trees and trace-trees of a Kripke structure \( K \). For all \( w \in W \), a \( W \)-labeled tree \( \langle t, \lambda \rangle \) is a \( w \)-run-tree of \( K \) if \( \epsilon \in t \), \( \lambda(\epsilon) = w \), and for all nodes \( x \in t \), if \( xn \in t \) then \( R(\lambda(x), \lambda(xn)) \). A run-tree of \( C \) is a \( \hat{w} \)-run-tree of \( C \) for some \( \hat{w} \in \hat{W} \). A \( \Gamma \)-labeled tree \( \langle t', \lambda' \rangle \) is a trace-tree of \( K \) if there is a run-tree \( \langle t, \lambda \rangle \) of \( K \) such that \( t' = t \) and \( \lambda' = \lambda \circ L \) (that is, for every node \( x \in t \), we have \( \lambda'(x) = L(\lambda(x)) \)); in this case we say that the run-tree \( \langle t, \lambda \rangle \) witnesses the trace-tree \( \langle t', \lambda' \rangle \).
CHAPTER 2. PRELIMINARIES

A transition constraint $C$ has an associated Kripke structure given by $K_C = \langle \Gamma_C, \Sigma_C, \Sigma_C^I, \rightarrow_C, [\_]_C \rangle$. Though both $C$ and $K_C$ are semantically equivalent (i.e., they have the same set of traces and trace-trees), the size of $K_C$ is exponential in the size of $C$.

**Finite internal nondeterminism.** Transition constraints can model both finite and infinite state systems. Practical applications of the results in this thesis were hardware systems, with a finite number of states. However, some of the theoretical results hold for infinite state systems as well. In the case of infinite state systems, we require that our constraints have the following finite internal nondeterminism property: For any constraint $C$, we require that (1) if $s \in \Sigma_C^I$, there is only a finite number of $t$, such that $t \in \Sigma_C^I$, and $[s] = [t]$, and (2) if $s \in \Sigma_C$, and $s \rightarrow_C t$, then there is only a finite number of distinct $u \in \Sigma_C$ such that, $s \rightarrow_C u$ and $[t]_C = [u]_C$.

### 2.3 Modules

Modules [AH96] are a special class of transition constraints with two special properties: (1) differentiation between inputs and outputs, and (2) executable non-blocking semantics.

Since a module $P$ is a transition constraint as well, it has a finite set of typed variables $X_P$, partitioned into private and observable variables. Some of the variables in $X_P$ are updated by the system, and the other variables in $X_P$ are updated by the environment. Hence, the set $X_P$ is also partitioned (orthogonally) into two sets: the set $\text{ctr} X_P$ of controlled variables, and the set $\text{extl} X_P$ of external variables. Observable variables that are also controlled are called interface variables, and observable variables that are not controlled are called external variables. The various classes of module variables and their relationships are summarized in Table 1. The distinction between private, interface, and external variables is similar to the distinction between internal, output, and input events in the formalism of I/O automata [Lyn96].

Like any transition constraint, the module $P$ proceeds in a sequence of rounds. The first round is an initialization round, during which the variables in $X_P$ are initialized. Each subsequent round is an update round, during which the variables in $X_P$ are updated. However, each round consists of several subrounds, and the system and the environment take turns in executing subrounds to update variables. The number of subrounds per round is fixed: each variable is updated in exactly one subround of each round. For this purpose, the
controlled variables of a module are partitioned into groups called atoms, and the variables within a group are updated simultaneously, in the same subround. The atoms are partially ordered. If atom \( A \) precedes atom \( B \) in the partial order, then in each round, the \( A \)-subround must precede the \( B \)-subround, and the updated values of the variables controlled by \( B \) may depend on the updated values of the variables controlled by \( A \). Inside an atom, the initialization and updating of variables are specified by actions. An action from \( X \) to \( Y \) is a nondeterministic guarded command—that is, a list of guarded assignments—so that (1) all guards and all right-hand-sides of assignments are expressions over the variables in \( X \), and (2) all left-hand-sides of assignments are variables from \( Y \). The action \( \alpha \) from \( X \) to \( Y \) is executable if the disjunction of all guards is true. Executable actions are enabled in all states. For each atom \( A \), in the \( A \)-subround of the initialization round, all variables controlled by \( A \) are initialized simultaneously, as defined by an initial action. In the \( A \)-subround of each update round, all variables controlled by \( A \) are updated simultaneously, as defined by an update action.

**Definition 2.3** [Atom] Let \( X \) be a finite set of typed variables. An \( X \)-atom \( A \) consists of a declaration and a body. The atom declaration consists of a set \( \text{ctr} X_A \subseteq X \) of controlled variables, a set \( \text{read} X_A \subseteq X \) of read variables, and a set \( \text{wait} X_A \subseteq X \setminus \text{ctr} X_A \) of awaited variables. The atom body consists of an executable initial action \( \text{Init}_A \) from \( \text{wait} X'_A \) to \( \text{ctr} X'_A \) and an executable update action \( \text{Update}_A \) from \( \text{read} X_A \cup \text{wait} X'_A \) to \( \text{ctr} X'_A \).

In the initialization round, the initial action of the atom \( A \) assigns initial values to the controlled variables as a nondeterministic function of the initial values of the awaited variables. In each update round, the update action of \( A \) assigns updated values to the controlled variables as a nondeterministic function of the latched values of the read variables and the updated values of the awaited variables. Hence, in each round, the \( A \)-subround can take place only after all awaited variables have already been updated. The variable \( x \) awaits the
variable \( y \), written \( x \succ_A y \), if \( x \in \text{ctr} X_A \) and \( y \in \text{wait} X_A \). Now, we can define reactive modules.

**Definition 2.4 [Module]** A (reactive) module \( P \) consists of a declaration and a body. The module declaration is a finite set \( X_P \) of typed variables that is partitioned as shown in Table 1. The module body is a set \( A_P \) of \( X_P \)-atoms such that (1) \( (\cup_{A \in A_P} \text{ctr} X_A) = \text{ctr} X_P \); (2) for all atoms \( A \) and \( B \) in \( A_P \), \( \text{ctr} X_A \cap \text{ctr} X_B = \emptyset \); and (3) the transitive closure \( \succ_P = (\cup_{A \in A_P} \succ_A)^+ \) is asymmetric.

The first two conditions ensure that the atoms of \( P \) control precisely the variables in \( \text{ctr} X_P \), and that each variable in \( \text{ctr} X_P \) is controlled by precisely one atom. The third condition ensures that the await dependencies among the variables in \( X_P \) are acyclic. A linear order \( A_0, \ldots, A_{k-1} \) of the atoms in \( A_P \) is consistent if for all \( 0 \leq i < j < k \), the awaited variables of \( A_i \) are disjoint from the controlled variables of \( A_j \). The asymmetry of \( \succ_P \) ensures that there exists a consistent order of the atoms in \( A_P \).

**Finite internal nondeterminism.** When executing the module \( P \), in each round, first the external variables are assigned arbitrary values of the correct types, and then the atoms in \( A_P \) are executed in an arbitrary consistent order \( A_0, \ldots, A_{k-1} \). Specifically, in the initialization round, after the external variables have been initialized, the initial action of \( A_0 \) is followed by the initial action of \( A_1 \) etc.; and in each update round, after the external variables have been updated, the update action of \( A_0 \) is followed by the update action of \( A_1 \) etc. In this manner, all awaited values are available when they are needed during the execution, and thus every round can be completed in \( k + 1 \) subrounds — one subround for the environment followed by one subround for each atom. Moreover, all nondeterminism in the completion of a round is caused by (1) nondeterminism of the environment in choosing values for external variables, and (2) by the nondeterminism of individual atoms in choosing an enabled guarded command to execute. However, since each nondeterministic choice of the environment results in a different valuation to observable variables, and the number of guards is finite (the description of the module itself is finite, even though the number of states could be infinite), a module has to necessarily have finite internal nondeterminism (even if the variables of the module range over infinite domains such as integers.)

A module \( P \) can be interpreted as a transition constraint \( C_P \), which has the same set of traces and trace-trees as \( P \). The observable variables of \( C_P \) are the observable variables of \( P \), and the private variables of \( C_P \) are the private variables of \( C \). The initial
module counter
  external in:bool
  interface out:bool
  private count:(0..5)
  atom controls count awaits in
    init
      [] true -> count’ := 0
    update
      [] in’ -> count’ := count +1
      [] true -> count’ := count
  endatom
  atom controls out reads count awaits count
    init
      [] true -> out’ := false
    update
      [] true -> out’ := (count’=0) & (count =5)
  endatom
endmodule

Figure 2.4: Lazy Counter Module

and update predicates of \( C \) can be derived from the initial and update commands of \( P \), respectively. The initial predicate of \( C_P \) is defined to be \( true \) in states \( s \) such that \( s \) can result from the initial round of \( P \). In a similar spirit, the update predicate of \( C_P \) is defined to be \( true \) in a state pairs \((s, t)\) such that \( P \) can start with state \( s \) and change to state \( t \) by the execution of an update round.

Example. Figure 2.4 shows a module that describes a lazy counter, that is intended to have the same semantics as the transition constraint shown in Figure 2.1. Note that the module distinguishes in as an external variable, and out as an interface variable (whereas the constraint does not make this distinction.) Further, unlike the transition constraint, the module has executable non-blocking semantics.
2.4 Parallel composition

The composition operation combines two transition constraints into a single transition constraint whose behavior captures the interaction between the two components. Two modules can be composed to form a single module as well. However, the rules for compositability of modules are more stringent than those of transition constraints. Two transition constraints $A$ and $B$ are compatible if their sets of private variables is disjoint.

**Definition 2.5 [Constraint composition]** If $A$ and $B$ are two compatible transition constraints, then the composition $A \parallel B$ is the transition constraint with the set $\text{priv} X_{A \parallel B} = \text{priv} X_A \cup \text{priv} X_B$ of private variables, the set $\text{obs} X_{A \parallel B} = \text{obs} X_A \cup \text{obs} X_B$ of observable variables. The initial predicate of $A \parallel B$ is the conjunction of the initial predicates of $A$ and $B$, and the update predicate of $A \parallel B$ is the conjunction of the update predicates of $A$ and $B$.

Two modules $P$ and $Q$ are compatible if (1) the interface variables of $P$ and $Q$ are disjoint, and (2) the await dependencies among the observable variables of $P$ and $Q$ are acyclic—that is, the transitive closure $(\succ_P \cup \succ_Q)^+_{\succ}$ is asymmetric. It follows that if $P$ and $R$ are compatible modules, and $P \preceq Q$, then $Q$ and $R$ are also compatible.

**Definition 2.6 [Module composition]** If $P$ and $Q$ are two compatible modules, then the composition $P \parallel Q$ is the module with the set $\text{priv} X_{P \parallel Q} = \text{priv} X_P \cup \text{priv} X_Q$ of private variables, the set $\text{intf} X_{P \parallel Q} = \text{intf} X_P \cup \text{intf} X_Q$ of interface variables, the set $\text{extl} X_{P \parallel Q} = (\text{extl} X_P \cup \text{extl} X_Q) \setminus \text{intf} X_{P \parallel Q}$ of external variables, and the set $A_{P \parallel Q} = A_P \cup A_Q$ of atoms.

Given two compatible modules $P$ and $Q$, it is seen that one gets the same set of traces by (1) viewing $P$ and $Q$ as transition constraints and interpreting $P \parallel Q$ as composition of transition constraints, and (2) first viewing $P \parallel Q$ as composition of modules and then interpreting the result as a transition constraint. The following two propositions follow directly from the definition of parallel composition:

**Proposition 2.1** Let $A$ and $B$ be two compatible transition constraints, and $\sigma$ be any sequence. Then, $\sigma$ is a trace of $A \parallel B$ iff (1) $[\sigma]_A$ is a trace of $A$, and (2) $[\sigma]_B$ is a trace of $B$.

**Proposition 2.2** Let $A$ and $B$ be two compatible transition constraints, and $T$ be any tree. Then, $T$ is a trace-tree of $A \parallel B$ iff (1) $[T]_A$ is a trace-tree of $A$, and (2) $[T]_B$ is a trace-tree of $B$. 
2.5 Reachability and invariant checking

For a transition constraint $C$, recall that $\Sigma_C$ is the set of all states. The \textit{reachable state-set} of $C$ is the subset of states obtainable by running $C$ from some initial state. Formally, the reachable state-set of $C$, denoted by $\Omega_C$, is defined as:

$$\Omega_C = \{s \mid \text{there exists a run } s_0, s_1, \ldots, s \text{ of } C\}.$$

The reachable state-set of a transition constraint can be obtained enumeratively, by doing a depth first or breadth first search on its Kripke structure $K_C$, starting from the initial states. The search will terminate if the number of states is finite. However, the worst case complexity of the search is exponential in the size of $C$.

\textbf{Invariant checking.} A set of states $\Theta$ is said to be an \textit{invariant} of constraint $C$ if $\Omega_C \subseteq \Theta$. If a state set $\Theta$ is not an invariant of $C$, there must exist a run $s_0, s_1, \ldots, s$ of $C$, such that $s \not\in \Theta$; such a run is called an \textit{error trace} that witnesses the violation of $\Theta$. An instance of the \textit{invariant-checking problem} consists of a constraint $C$ and a set of states $\Theta \subseteq \Sigma_C$. The answer to the invariant-checking problem is true if $\Theta$ is an invariant of $C$, and otherwise an error trace that witnesses the violation of $\Theta$.

\textbf{Symbolic reachability.} If the set $\Theta$ is given as a predicate, the invariant-checking problem is PSPACE-complete, so we cannot hope to find an efficient solution. However, several heuristics have been developed that work well in practice. One such heuristic is \textit{symbolic reachability}. Symbolic reachability algorithms operate on implicit or symbolic representations for sets of states. For example, if $x, y$ and $z$ are boolean variables, the predicate $x + y + z$ is a symbolic representation the following set of seven states

$$\{(a, b, c) \mid a = true, \text{ or } b = true, \text{ or } c = true\}.$$

If all the variables of a constraint are boolean variables, then symbolic reachability can be done using boolean predicates. Even if a variable is not boolean, as long as the domain of the variable is finite, say $n$, it can be encoded using $\log_2 n$ boolean variables. \textit{Binary decision diagrams} or \textit{BDDs} [Bry86] provide a compact and canonical representation for boolean predicates. The canonicity is with respect to a total ordering of the variables. We have used BDDs extensively in this work. However, for most parts of this thesis, BDDs are used as \textit{“black boxes”} and a detailed exposition is not necessary. The interested reader should refer to [Bry86] for an introduction. Below, we briefly describe computing the reachable state-set of constraints using BDDs.
A BDD over \( n \) variables is of size \( 2^n \) in the worst case. However, several boolean predicates that arise during symbolic reachability of constraints tend to have polynomial-sized BDDs. Set operations — union, intersection, complementation and test for equality — can be done in time polynomial on the size of the input BDDs. In fact, clever implementation tricks are employed to do complementation in constant time and equality checking in almost constant time.

Let \( C \) be a constraint with only boolean variables. The initial predicate \( I_C \) and update predicate \( U_C \) can be represented as BDDs. The symbolic representation of the update predicate is commonly known as the transition relation. Given a BDD \( \sigma \) that represents some state-set of \( C \) (in terms of its unprimed variables), the operation \( R^1_C(\sigma) \) (called the single-step successor) returns a BDD that represents the image of \( \sigma \) with respect to \( U_C \). Formally, \( R^1_C(\sigma) \) returns the set \( \{ t \mid \exists s.s \in \sigma \text{ and } (s, t) \in U_C \} \). In BDD terminology, the single-step successor operation is called image computation, and consists of three steps: (1) computing the intersection of state-set and the transition relation, (2) existentially quantifying unprimed variables from the intersection, and (3) renaming primed variables to unprimed variables. The worst case complexity of image computation is exponential on the size of the input BDD. Algorithm 2.5 computes the BDD representing the reachable state-set of the given transition constraint using above mentioned operations.

**Algorithm 2.5**

\{Given constraint \( C \) compute BDD representing \( \Omega_C \) \}

\( \sigma := \) BDD representing \( I_C \) in terms of unprimed variables of \( C \)

**loop**

\( \text{image} := R^1_C(\sigma) \)

\( \sigma' := \sigma \cup \text{image} \)

**if** \( \sigma' = \sigma \) **then return** \( (\sigma) \)

**else** \( \sigma := \sigma' \)

**forever**

The performance of BDD based heuristics are usually hard to analyze mathematically. It has been proved that BDDs for transition relations of digital circuits grow exponentially with the path-width (which is a measure of communication complexity in the circuit) of the circuit [McM93]. Several useful circuits in practice tend to have small path-widths, and consequently their transition relations are small BDDs, provided we find a good ordering of the variables. Finding a good ordering, however, is NP-Complete. No general
results on bounding sizes are available for BDDs that represent images and reachable state-sets that arise during symbolic reachability. The general experience has been that BDDs tend to work well for systems with up to 50 variables. With larger systems, one has to employ domain-specific heuristics to be able to complete symbolic reachability.
Chapter 3

Refinement

In hierarchical verification, we need to check proof obligations of the form $P \preceq Q$, where $P$ and $Q$ are system descriptions, and $\preceq$ is a refinement preorder (either linear, or branching) on system descriptions. The assertion $P \preceq Q$ holds if $P$ describes the same system as $Q$, but possibly on a finer level of detail (or equivalently, $Q$ describes the same system as $P$ but possibly on a coarser level of abstraction.) For example, $P$ may be an RTL-level description of a pipelined processor, and $Q$ may be an ISA description of the same processor. The assertion $P \preceq Q$ is therefore variously pronounced as “$P$ implements $Q$,” or “$P$ refines $Q$,” or “$Q$ specifies $P$,” or “$Q$ abstracts $P$.” In this chapter, we define two notions of refinement —linear and branching. We explore equivalent alternative views of linear and branching refinements, and computational aspects of checking these refinements. For a particular class of implementation-specification pairs, linear and branching refinements coincide; we call such refinements as projection refinements. We also introduce the need for assume-guarantee reasoning, review an assume-guarantee rule for linear refinement, and present soundness proof of an assume-guarantee rule for branching refinement.

3.1 Linear refinement

The notion that two transition constraints describe the same system at different levels of detail is captured by the refinement relation between transition constraints.

The transition constraint $B$ is refirable by transition constraint $A$ if every observable variable of $B$ is an observable variable of $A$. The module $Q$ is refirable by the module $P$, if the following conditions are met: (1) every interface variable of $Q$ is an in-
terface variable of \( P \); (2) every external variable of \( Q \) is an observable variable of \( P \); and (3) for all observable variables \( x \) of \( Q \) and all interface variables \( y \) of \( Q \), if \( y \succ_Q x \), then \( y \succ p \ x \)

Refinability is a syntactic criterion. Suppose transition constraint (module) \( B \) is refinable by transition constraint (module) \( A \). The definition of \( A \) refines \( B \) depends on the notion of time —trace containment in linear time, and trace-tree containment in branching time.

**Definition 3.1 [Linear refinement]** The transition constraint (module) \( A \) linear-refines or trace contains the transition constraint (module) \( B \), written \( A \preceq_L B \), if (1) \( B \) is refinable by \( A \), and (2) for every trace \( \tau \) of \( A \), the projection \( [\tau]_B \) is a trace of \( B \).

**Refinement example.** Consider a situation where two processors want to access a shared bus. An arbitration unit is needed to arbitrate access requests to the bus. The arbiter has two boolean external variables \( \text{req1} \) and \( \text{req2} \), and two interface variables \( \text{grant1} \) and \( \text{grant2} \). We do not model the processors themselves, but we expect the following (obvious) protocol: If processor 1 (processor 2) wants the bus, it asserts \( \text{req1} \) (\( \text{req2} \)) and waits for the arbiter to assert \( \text{grant1} \) (\( \text{grant2} \)). Figure 3.1 shows a specification of the arbiter. The specification \( \text{arbSpec} \) has a private variable \( \text{state} \) that is set to \( \text{free} \) in the initial round. In every update round, if \( \text{req1} \) (\( \text{req2} \)) is asserted, then \( \text{state} \) could be set to \( \text{gt1} \) (\( \text{gt2} \)). Also, regardless of the requests \( \text{state} \) could be set to \( \text{free} \). Note the internal nondeterminism in the specification —if both \( \text{req1} \) and \( \text{req2} \) are asserted, \( \text{state} \) could be set to any one of \( \text{gt1} \), \( \text{gt2} \) or \( \text{free} \). The value of \( \text{state} \) directly influences the outputs \( \text{grant1} \) and \( \text{grant2} \). The output \( \text{grant1} \) (\( \text{grant2} \)) is asserted whenever \( \text{state} \) is set to \( \text{gt1} \) (\( \text{gt2} \)).

One possible implementation of the arbiter is given by module \( \text{arbImpl} \), shown in Figure 3.2. Module \( \text{arbImpl} \) has the same variables as the module \( \text{arbSpec} \). The private variable \( \text{state} \) can now take one of four values \( \text{free1} \), \( \text{free2} \), \( \text{gt1} \) and \( \text{gt2} \). It is set to \( \text{free1} \) in the initial round. In the update rounds, \( \text{state} \) toggles between \( \text{free1} \) and \( \text{free2} \) in the absence of external requests. In effect, the implementation has two free states, instead of the single free state in the specification. Also, \( \text{state} \) is set to \( \text{grant1} \) (\( \text{grant2} \)) only if currently \( \text{state} \) is \( \text{free1} \) (\( \text{free2} \)) and \( \text{req1} \) (\( \text{req2} \)) is asserted. Module \( \text{arbImpl} \) is deterministic —given a current state and particular values for external variables, there is a
module arbSpec
  external req1, req2: bool
  interface grant1, grant2: bool
  private state: { free, grant1, grant2 }
  atom controls state reads state awaits req1, req2
  init
      [] true -> state’ := free
  update
      [] req1’ -> state’ := grant1
      [] req2’ -> state’ := grant2
      [] true -> state’ := free
  endatom
  atom controls grant1 awaits state
  init update
      [] state’ = grant1’ := true
      [] default’ := grant1’ := false
  endatom
  atom controls grant2 awaits state
  init update
      [] state’ = grant2’ := true
      [] default’ := grant2’ := false
  endatom
endmodule

Figure 3.1: Specification of an arbiter
module arbImpl

external req1, req2 : bool

interface grant1, grant2 : bool

private state: [free1, free2, grant1, grant2]

atom controls state reads state awaits req1, req2

init
[] true → state’ := free1

update
[] state = free1→ state’ := if req1’ then grant1 else free2
[] state = free2→ state’ := if req2’ then grant2 else free1
[] state = grant1→ state’ := free2
[] state = grant2→ state’ := free1
endatom

atom controls grant1 awaits state

init update
[] state’ = grant1→ grant1’ := true
[] default→ grant1’ := false
endatom

atom controls grant2 awaits state

init update
[] state’ = grant2→ grant2’ := true
[] default→ grant2’ := false
endatom

endmodule

Figure 3.2: Implementation of arbiter
unique next state. Note that \( \text{arbImpl} \preceq_L \text{arbSpec} \) (every trace of \( \text{arbImpl} \) is an allowable trace of \( \text{arbSpec} \)).

**Logical view.** Temporal properties of transition constraints can be specified using temporal logics. A formal treatment of temporal logics is beyond the scope of this thesis. We refer the interested reader to [Eme90]. Also, refinement between transition constraints can be also viewed from the perspective of temporal logics. We state such connections, whenever possible. For linear refinement, the logical viewpoint is provided by linear temporal logic (LTL) [Eme90].

**Proposition 3.1** For any two transition constraints \( A \) and \( B \), the following two statements are equivalent:

1. \( A \preceq_L B \)

2. For every LTL formula \( \varphi \), if \( B \) satisfies \( \varphi \), then \( A \) satisfies \( \varphi \).

**Linear refinement between Kripke structures.** Given two Kripke structures \( K_1 = \langle \Gamma, W_1, \hat{W}_1, R_1, L_1 \rangle \), and \( K_2 = \langle \Gamma, W_2, \hat{W}_2, R_2, L_2 \rangle \), we say that \( K_1 \preceq_L K_2 \), if every trace of \( K_1 \) is a trace of \( K_2 \). Note that we require the observation alphabet of both Kripke structures to be the same.

**Complexity.** Given two transition constraints \( A \) and \( B \), the complexity of checking \( A \preceq_L B \) is given by \( |K_A| \times 2^{|K_B|} \). i.e., the check is linear in the state space of \( A \) and exponential in the state space of \( B \). The problem has been shown to be PSPACE complete on \( |K_B| \) [SM73].

### 3.2 Branching refinement

While simple and intuitive, trace containment has several shortcomings. First, it is a practical impossibility to check trace containment \( A \preceq_L B \), automatically for all but the smallest examples, because the check is exponential in the number of states of \( B \) if \( B \) is nondeterministic (as specifications often are.)

Second, in top-down design, if system description \( A \) fleshes out detail that is left open in system description \( B \), there is a much tighter relation between \( B \) and \( A \) than trace containment would indicate; namely, each implementation state of \( A \) corresponds to a specification state of \( B \). This tighter relation is captured mathematically by \( \preceq_B \) (branching refinement or simulation.) Intuitively, \( B \) simulates \( A \) iff, starting from the initial states and
continuing ad infinitum, every input-output pair of $A$ can be matched by the same input-output pair in $B$ [Mil71]. Alternative, equivalent characterizations can be given in terms of trace-tree containment and game theory. Clearly, if $B$ simulates $A$, then every trace of $A$ is also a trace of $B$. The converse is not true; that is, simulation is a stronger requirement than trace containment. However, it has been said that trace containment without simulation is more often than not due to coincidence rather than systematic design [Kur94].

While trace containment is defined globally, for input-output sequences of arbitrary length, simulation is defined locally, by considering individual input-output pairs for all states. It is this locality in the definition of simulation that leads to significant advantages. First, if $B$ is claimed to simulate $A$, then a witness to this claim can be produced in the form of a relation between states of $A$ and states of $B$, and the witness can be efficiently checked for correctness (the check is linear in the number of states of $A$ and $B$.) Such witness relations are widely used in verification methods and tools, under various names like homomorphisms [Kur94] and refinement mappings [AL91, Lyn96]. Second, even if no witness is available, the existence of a simulation can be checked in polynomial time. We now define branching refinement formally.

**Definition 3.2** [Branching refinement] The transition constraint (module) $A$ branching-refines or trace-tree contains the transition constraint (module) $B$, written $A \triangleleft_B B$, if (1) $B$ is refinable by $A$, and (2) for every trace-tree $T$ of $A$, the projection $[T]_B$ is a trace-tree of $B$.

Milner [Mil71] gave the following definition for when one system simulates another.

**Definition 3.3** [Simulation relation] Let $A$ and $B$ be transition constraints such that $B$ is refinable by $A$. A relation $H \subseteq \Sigma_A \times \Sigma_B$ is a simulation if the following conditions are met: (1) For all states $s \in \Sigma_A$ and $t \in \Sigma_B$, if $H(s,t)$ then $[s]_B = [t]_B$; (2) For all states $s \in \Sigma_A$ and $t \in \Sigma_B$, if $H(s,t)$ and $s \rightarrow_A s'$ then there is a state $t'$ of $B$ such that $t \rightarrow_B t'$ and $H(s',t')$.

**Definition 3.4** [Simulation] Transition constraint $B$ simulates transition constraint $A$ (written $A \triangleleft_S B$) if the following conditions are met: (1) $B$ is refinable by $A$; and (2) There is a simulation relation $H \subseteq \Sigma_A \times \Sigma_B$ such that, for every initial state $s$ of $A$ there is an initial state $t$ of $B$ satisfying $H(s,t)$. We say that $H$ witnesses the simulation $A \triangleleft_S B$.

The following proposition relates simulation and branching refinement. Its correctness requires the assumption that transition constraints have finite internal nondeterminism.
CHAPTER 3. REFINEMENT

Proposition 3.2 Let A and B be any two transition constraints. Then \( A \preceq_B B \) iff \( A \preceq_S B \).

Proof: Suppose \( B \) simulates \( A \). Let \( \Theta \) be the witnessing simulation relation. Consider any trace-tree \( \langle t, \lambda'_A \rangle \) of \( A \), with \( \langle t, \lambda_A \rangle \) as the witnessing run-tree. We can construct a corresponding run-tree \( \langle t, \lambda_B \rangle \) of \( B \) as follows: Set \( \lambda_Q(\epsilon) = r \), where \( r \in \Sigma_B \), and \( \Theta(\lambda_Q(\epsilon), r) \). Since \( \Theta \) is a witness to the simulation, such an \( r \) exists. If \( \lambda_A(x) = p, \lambda_B(x) = q \), and \( \lambda_A(xn) = \tilde{p} \), then set \( \lambda_B(xn) \) to a state \( \tilde{q} \) of \( B \), such that \( q \rightarrow_B \tilde{q} \) and \( \Theta(q, \tilde{q}) \). Since \( \Theta \) is a valid simulation, such a \( \tilde{q} \) exists. Now note that \( \langle t, \lambda_B \rangle \) is a trace-tree of \( B \) and is identical to \( \langle t, \lambda'_A \rangle \).

Conversely, suppose \( B \) does not simulate \( A \). We define the notion of \( k \)-simulation for \( k \in \mathbb{N} \) inductively. For a given \( k \), the \( k \)-simulation \( H^k \) is a binary relation contained in \( \Sigma_A \times \Sigma_B \).

1. \( H^0(p, q) \) iff \( \langle p \rangle_B = \langle q \rangle_B \).

2. \( H^k(p, q) \) iff \( \langle p \rangle_B = \langle q \rangle_B \) and for every \( \tilde{p} \in \Sigma_A \), such that \( p \rightarrow_A \tilde{p} \), there exists \( \tilde{q} \in \Sigma_B \) such that \( q \rightarrow_B \tilde{q} \) and \( H^{(k-1)}(\tilde{p}, \tilde{q}) \).

We use \( \neg H^k(p, q) \) to denote that \( H^k(p, q) \) is not satisfied.

We show that for any states \( p \) and \( q \), if \( \neg H^k(p, q) \), then there exists a \( p \)-run-tree \( T \) of \( A \) with depth \( k \) (called the distinguishing tree of \( p \) with respect to \( q \)), such that for every \( q \)-run-tree \( T' \) of \( B \), \( \langle T \rangle_B \neq \langle T' \rangle_B \). The proof is by induction on \( k \). If \( \neg H^0(p, q) \), the proof is trivial.

For the inductive case, we need to define a notion for merging trees. Let \( T_1 = \langle t_1, \lambda_1 \rangle, T_2 = \langle t_2, \lambda_2 \rangle, \ldots, T_m = \langle t_m, \lambda_m \rangle \) be a set of labelled trees with \( \lambda_1(\epsilon) = \lambda_2(\epsilon) = \cdots = \lambda_m(\epsilon) \). Then \( \text{Merge}(T_1, T_2, \ldots, T_m) \) is defined to be a tree \( T = \langle t, \lambda \rangle \), where the root has the same label as \( T_1 \) (i.e., \( \lambda(\epsilon) = \lambda_1(\epsilon) \)). Further, the children of the root of \( T \) are all the children of the roots of \( T_1, T_2, \ldots, T_m \), along with their respective subtrees. Figure 3.3 shows an example.

If \( \neg H^k(p, q) \) for some \( k > 0 \), then there exists some state \( \tilde{p} \in \Sigma_A \), such that \( p \rightarrow_A \tilde{p} \) and for every \( \tilde{q} \) such that \( q \rightarrow_B \tilde{q} \) and \( \langle \tilde{p} \rangle_B = \langle \tilde{q} \rangle_B \), we have \( \neg H^{(k-1)}(\tilde{p}, \tilde{q}) \). Further the number of such \( \tilde{q} \) is finite (say \( n \)) due to finite internal nondeterminism of \( B \). For each such \( \tilde{q} \), we can construct (due to induction hypothesis) a distinguishing tree \( T_j \) of \( \tilde{p} \) with respect to \( \tilde{q} \). We can then form \( T_M = \text{Merge}(T_1, T_2, \ldots, T_n) \) (note that all \( T_j \) have the same label \( \tilde{p} \) at the root.) Since all the \( T_j \) have depth at most \( (k-1) \), the depth of \( T_M \)
is also bounded by \((k-1)\). The desired \(p\)-run-tree \(T\) (the distinguishing run-tree of \(p\) with respect to \(q\)) can now be obtained by adding a new root with label \(p\) and making the root of \(T_M\) as its only child. Since the depth of \(T_M\) is bounded by \((k-1)\), the depth of \(T\) is bounded by \(k\).

It is not difficult to see that \(A \leq_S B\) iff for every \(r \in \Sigma^I_A\), there exists a state \(s \in \Sigma^I_B\), such that \(H^k(r,s)\) for every \(k\). Since \(A \nleq_S B\), there is a \(r \in \Sigma^I_A\), and a \(k\) such that there is no \(s \in \Sigma^I_B\) satisfying \(H^k(r,s)\). For every \(s\) in \(\Sigma^I_B\) such that \([s]_B = [r]_B\), we can construct a distinguishing tree \(T_s\) of \(r\) with respect to \(s\). There are only finitely many such states due to the finite internal nondeterminism property. We can then apply the Merge operation all such trees to form tree \(T\). Therefore, there is a distinguishing \(r\)-run-tree \(T\) of \(B\), such that \([T]_B\) is not a trace-tree of \(B\) and \(B\) does not tree-contain \(A\).

**Logical view.** For branching refinement, the logical viewpoint is provided by any of the branching logics, namely universal fragment of CTL (computation tree logic), CTL* and AFMC (alternation free \(\mu\)-calculus) [Eme90].

**Proposition 3.3** For any two transition constraints \(A\) and \(B\), the following two statements are equivalent:
1. $A \preceq_B B$

2. For every formula $\varphi$ in the universal fragment of CTL (CTL*, AFMC), if $B$ satisfies $\varphi$, then $A$ satisfies $\varphi$.

**Game-theoretic view.** Let $A$ and $B$ be transition constraints such that $B$ is refinable by $A$. Consider a two-player game whose positions are pairs $\langle w_1, w_2 \rangle \in \Sigma_A \times \Sigma_B$ of states. The adversary first chooses a state $\hat{w}_1 \in \Sigma'_A$, and the protagonist then chooses a state $\hat{w}_2 \in \Sigma'_B$, such that $[\hat{w}_1]_B = [\hat{w}_2]_B$. The initial position of the game is $\langle \hat{w}_1, \hat{w}_2 \rangle$. The game then proceeds in a sequence of rounds. In each round, if $\langle w_1, w_2 \rangle$ is the current position, first the adversary updates the first component $w_1$ to some $w'_1$, such that $w_1 \rightarrow_A w'_1$, and then protagonist updates the second component $w_2$ to some $w'_2$ such that $w_2 \rightarrow_A w'_2$ and $[w'_1]_B = [w'_2]_B$. If no such $w'_2$ exists, then the protagonist loses. If the game proceeds ad infinitum, for $\omega$ rounds, then the adversary loses. It is easy to see that $A \preceq_B B$ iff the protagonist has a winning strategy.

**Branching refinement between Kripke structures.** Given two Kripke structures $K_1 = \langle \Gamma, W_1, \bar{W}_1, R_1, L_1 \rangle$, and $K_2 = \langle \Gamma, W_2, \bar{W}_2, R_2, L_2 \rangle$, we say that $K_1 \preceq_B K_2$, if every trace tree of $K_1$ is a trace tree of $K_2$. Note again, that we require the observation alphabet of both Kripke structures to be the same.

**Complexity.** Given two transition constraints $A$ and $B$, the complexity of checking $A \preceq_B B$ is given by $O(|R_A| \cdot |W_B| + |R_B| \cdot |W_A|)$, where $K_A = \langle \Gamma, W_A, \bar{W}_A, R_A, L_A \rangle$, and $K_B = \langle \Gamma, W_B, \bar{W}_B, R_B, L_B \rangle$.

### 3.3 Projection refinement

Note that $A \preceq_B B$ implies that $A \preceq_L B$, but the converse is not true. The problem of checking if $A \preceq_L B$ is PSPACE-hard in the state space of $B$. The problem of checking if $A \preceq_B B$ can be checked in time that is quadratic on the state spaces of $A$ and $B$. For the special case in which all variables of $B$ are observable, linear refinement and branching refinement coincide. In such a case, we just say that $A$ projection-refines $B$, or $A \preceq B$. Therefore, checking if $A \preceq_L B$ reduces to checking if for every trajectory $\bar{s}$ of $A$, the projection $[\bar{s}]_B$ is a trajectory of $B$. This can be done by a transition-invariant check [HQR98], whose complexity is linear in the reachable state-sets of both $A$ and $B$. 
Proposition 3.4 If every variable of B is observable and B is refinable by A, then \( A \preceq B \) iff (1) if \( s \) is an initial state of A, then \([s]_B\) is an initial state of B, and (2) if \( s \) is a reachable state of A and \( s \rightarrow_A t \), then \([s]_B \rightarrow_B [t]_B\).

Proof: Suppose (1) and (2) hold. Consider any trace \( \gamma \) of A, with \( \sigma \) as the witnessing run. Due to (1) and (2), we can do induction on the length of \( \sigma \) and produce a run \( \sigma' \) of B such that \([\gamma]_B = [\sigma]_B = [\sigma']_B\). Conversely, suppose \( A \preceq B \). Since every variable in B is observable, and is also present in A, (1) and (2) follow.

Checking projection refinement amounts to checking if every reachable transition of the implementation is an allowable transition of the specification. This can be achieved by adding such a check to symbolic reachability analysis (Algorithm 2.5.)

Suppose that B is refinable by A, but not projection refinable. This means that there are some private variables in B. Define \( B^u \) to be the transition constraint obtained by making every private variable of B an observable variable. If we compose A with a module \( W \) whose observable variables include the private variables of B, then \( B^u \) is projection refinable by the composition \( A \parallel W \). Moreover, if \( W \) does not constrain any observable variables of A, then \( A \parallel W \preceq_L B^u \) implies \( A \preceq_L B \) (in fact, A is simulated by B.) Such a module \( W \) is called a witness to the refinement \( A \preceq_L B \). We require \( W \) to be a module, because we need to ensure that \( W \) does not constrain the observable variables of A, and that it does not deadlock. In order to check refinement, it is sufficient to first find a witness module and then check projection refinement, as stated by the following proposition.

Proposition 3.5 [Witnesses] Consider two transition constraints A and B, such that B is refinable by A. Let \( W \) be a module such that the interface variables of \( W \) include the private variables of B, and are disjoint from the observable variables of A. Then (1) \( B^u \) is projection refinable by \( A \parallel W \), and (2) \( A \parallel W \preceq B^u \) implies \( A \preceq_L B \).

Proof: Since the interface variables of \( W \) are disjoint from the observable variables of A, for every run \( \sigma \) of A, there exists a run \( \sigma' \) of \( A \parallel W \), such that \( \sigma = [\sigma']_A \). The proposition follows immediately.

3.4 State explosion and assume-guarantee reasoning.

Recall that the complexity of checking linear refinement is exponential in the state space of the specification and linear in the state space of the implementation. We also
stated that the complexity of checking branching refinement is quadratic in the state spaces of both the specification and implementation. The number of states of a Kripke structure, however, depends exponentially on the size of the transition constraint. (Note that $n$ boolean variables give rise to $2^n$ states—this is the state-explosion problem.) Thus, even algorithms that are polynomial in the number of states (such as the ones used for checking branching refinement) are often infeasible in practice, and techniques have been studied for dividing a given verification task into simpler subtasks.

Let $P$ and $Q$ be modules such that $Q$ is refinable by $P$. Compositional techniques for dividing the verification task $P \preceq Q$ into simpler subtasks are guided by the structures of $P$ and $Q$. If the refinement relation $\preceq$ is interpreted as $\preceq_L$, a number of compositional techniques are known. Specifically, if $P = P_1 || P_2$ and $Q = Q_1 || Q_2$, then in order to check $P \preceq_L Q$, it suffices to check both $P_1 \preceq_L Q_1$ and $P_2 \preceq_L Q_2$. This compositional principle for trace containment is propositionally valid whenever parallel composition corresponds to trace intersection (replace $\parallel$ by conjunction, and $\preceq_L$ by implication.) Unfortunately, the compositional principle is often not helpful, because $P_1$ typically refines $Q_1$ only when constrained by an environment that behaves like $P_2$, and similarly, $P_2$ may refine $Q_2$ only when constrained by an environment that behaves like $P_1$. Under certain modeling assumptions that are satisfied by all legal modules (namely, nonblocking and finite nondeterminism), the compositional principle can be strengthened to an assume-guarantee principle [Sta85, CM88, CLM89, GL94, AL95, AH96, McM97]. Below, we state the assume-guarantee rule for modules from [AH96].

**Proposition 3.6** Let $P, Q, P', Q'$ be modules such that $P \parallel Q$ and $P' \parallel Q'$ exist. If $P \parallel Q' \preceq_L P', P' \parallel Q \preceq_L Q'$, and every external variable of $P' \parallel Q'$ is an observable variable of $P \parallel Q$, then $P \parallel Q \preceq_L P' \parallel Q'$.

Three observations about this proof rule are important. First, the rule addresses the issue that the environment of $P_1$ may have to be suitably constrained in order to implement $Q_1$, and similarly for $P_2$. Second, the rule avoids reasoning about the compound implementation $P_1 \parallel P_2$, which typically has the largest of the involved state spaces. Third, unlike the compositional principle, the assume-guarantee principle is circular and therefore not propositionally valid—it requires induction on the length of traces. See [Qad99] for how Proposition 3.6 is related to the assume-guarantee rule in [McM97].
3.5 Assume-guarantee rule for branching refinement

By contrast to the case of trace containment, if the refinement relation $\preceq_B$ is interpreted as $\preceq_B$, then little is known about compositional techniques other than the fact that the compositional principle remains valid whenever parallel composition corresponds to the intersection trace-trees. In particular, it would be useful to have an assume-guarantee principle for branching refinement, which, given that $P_1 \parallel Q_2 \preceq_B Q_1$ and $Q_1 \parallel P_2 \preceq_B Q_2$, lets us conclude that $P_1 \parallel P_2 \preceq_B Q_1 \parallel Q_2$. Below, we show that under the same modeling assumptions under which the assume-guarantee principle is sound for linear-refinement, it is also sound for branching refinement.

**Example.** We illustrate our assume-guarantee rule through a simple communication protocol. The specification of the protocol is given by $\text{Spec} := \text{sndSpec} \parallel \text{rcvSpec}$, where $\text{sndSpec}$ and $\text{rcvSpec}$ are modules shown in Figure 3.4 and Figure 3.5 respectively. The $\text{sndSpec}$ module has a boolean external variable $\text{ack}$, interface variables $\text{sndMsg}$ of type $\text{msgType}$ and $\text{snd}$ of type $\text{bool}$, and a boolean private variable $\text{wait}$. When a message is produced in $\text{sndMsg}$ during some round, the boolean $\text{snd}$ is set to $\text{true}$ in that round, and reset to $\text{false}$ in the next round. Further, private variable $\text{wait}$ is set to $\text{true}$, to denote that $\text{sndSpec}$ is waiting for an $\text{ack}$. When $\text{wait}$ is $\text{true}$ and $\text{sndSpec}$ receives an $\text{ack}$, $\text{wait}$ is reset to $\text{false}$. The $\text{rcvSpec}$ module has external variables $\text{sndMsg}$ and $\text{snd}$ interface variable $\text{ack}$, and a boolean private variable $\text{pending}$. When $\text{rcvSpec}$ receives a message (indicated by the $\text{sndSpec}$ setting $\text{snd}$ to $\text{true}$), it records the event by setting $\text{pending}$ to $\text{true}$. It then waits for an arbitrary amount of time, and then resets $\text{wait}$ to $\text{false}$, simultaneously sending an $\text{ack}$. The implementation of the protocol is given by $\text{Impl} := \text{sndImpl} \parallel \text{rcvImpl}$, where $\text{sndImpl}$ and $\text{rcvImpl}$ are modules shown in Figure 3.6 and Figure 3.7 respectively. Module $\text{sndImpl}$ differs from $\text{sndSpec}$ in its response to an incorrect $\text{ack}$. If $\text{sndImpl}$ receives an $\text{ack}$ when it is not waiting for one, it just goes into an error state, and starts sending arbitrary messages. In a similar situation, $\text{sndSpec}$ just ignores the offending $\text{ack}$. Thus $\text{sndImpl} \not\preceq_B \text{sndSpec}$. Similarly, $\text{rcvImpl}$ differs from $\text{rcvSpec}$, if it receives a message when $\text{pending}$ is $\text{true}$. Consequently $\text{rcvImpl} \not\preceq_B \text{rcvSpec}$. However, the following refinements hold: (1) $(\text{sndImpl} \parallel \text{rcvSpec}) \preceq_B \text{sndSpec}$, and (2) $(\text{rcvImpl} \parallel \text{sndSpec}) \preceq_B \text{rcvSpec}$. The assume-guarantee rule we prove in this section concludes $(\text{sndImpl} \parallel \text{rcvImpl}) \preceq_B (\text{sndSpec} \parallel \text{rcvSpec})$ from (1) and (2).

Below, we present the formal statement and proof of our assume-guarantee rule
module sndSpec

external ack:bool

interface sndMsg:msgType, snd:bool

private wait:bool

atom controls wait, sndMsg reads ack, wait, sndMsg

init

[] true -> wait’ := false

update

[] ~wait-> sndMsg’ := nondet; wait’ := true
[] wait & ack-> wait’ := false
[] ~wait->

endatom

atom controls snd reads wait awaits wait

init

[] true -> snd’ := false

update

[] ~wait & wait’ -> snd’ := true
[] default-> snd’ := false

endatom

endmodule

Figure 3.4: Specification of Sender
module rcvSpec

    interface ack:bool
    external snd:bool
    private pending:bool

atom controls pending reads pending, snd
    init
        [] true -> pending' := false
    update
        [] ¬pending & snd-> pending' := true
        [] pending-> pending' := false
        [] pending->
endatom

atom controls ack reads pending awaits pending
    init
        [] true -> ack' := false
    update
        [] pending & ¬pending' -> ack' := true
        [] default-> ack' := false
endatom

endmodule

Figure 3.5: Specification of Receiver
module sndImpl
  
  external ack:bool

  interface sndMsg:msgType, wait:bool
  
  private wait, error:bool

  atom controls wait, sndMsg, error reads ack, wait, sndMsg, error

  init
  [] true -> wait’ := false; error’ := false

  update
  [] ~wait-> sndMsg’ := nondet; wait’ := true
  [] wait & ack-> wait’ := false
  [] ~wait->
  [] ~wait & ack-> error’ := true
  [] error’ -> sndMsg’ := nondet; wait’ := true

  endatom

  atom controls snd readswait awaitswait

  init
  [] true -> snd’ := false

  update
  [] ~wait & wait’ -> snd’ := true
  [] default-> snd’ := false

  endatom

endmodule

Figure 3.6: Implementation of the sender
module rcvImpl

    interface ack:bool
    external snd:bool

    private pending, error:bool

    atom controls pending, error reads pending, snd, error
    init
        [] true -> pending' := false; error' := false
    update
        [] ~pending & snd-> pending' := true
        [] pending -> pending' := false
        [] pending & snd-> error' := true
        [] pending ->
    endatom

    atom controls ack reads pending awaits pending, error
    init
        [] true -> ack' := false
    update
        [] pending & ~pending' -> ack' := true
        [] error' -> ack' := true
        [] default-> ack' := false
    endatom

endmodule

Figure 3.7: Implementation of Receiver
for branching refinement. The proof is by induction over partial trace-trees.

**Theorem 3.1** Let $P, Q, P', Q'$ be modules such that $P \parallel Q$ and $P' \parallel Q'$ exist. If $P \parallel Q \preceq_B P', P' \parallel Q \preceq_B Q'$, and every external variable of $P' \parallel Q'$ is an observable variable of $P \parallel Q$, then $P \parallel Q \preceq_B P' \parallel Q'$.

**Proof:** It is checked that $P' \parallel Q'$ is refinable by $P \parallel Q$. We need to show that every trace-tree of $P \parallel Q$ is also a trace-tree of $P' \parallel Q'$. We do this by carrying out an induction over partial trace-trees.

In the following, $\lambda \cdot [x \rightarrow u]$ is a used to denote a function that evaluates to $u$ at $x$ and behaves like $\lambda$ at all other inputs.

Given a module $R$, a subset $X \subseteq \text{obs}X_R$ of the observable variables is *awaits-closed* if, whenever $y \succ R x$ and $y \in X$, then $x \in X$. For an awaits-closed set $X \subseteq \text{obs}X_R$, a pair $(\langle t, \lambda \rangle, x)$ is an $X$-partial trace-tree of $R$ if (1) $x$ is a leaf node of $t$, (2) $(t \setminus \{x\}, \lambda)$ is a trace-tree of $R$, (3) $\lambda(x)$ is a valuation to variables in $X$, and (4) there exists a valuation $v \in \Gamma_R$ that $v$ and $\lambda(x)$ agree on the valuations to variables in $X$, and $(t, \lambda \cdot [x \rightarrow v])$ is a trace-tree of $R$. Intuitively, partial trace-trees are obtained by revealing only a subset of the valuations to observable variables at some leaf node of a trace-tree (see Figure 3.8 for an example.) In the following, for notational simplicity, we omit explicit projections on partial trace-trees. For example, we write that $(\langle t, \lambda \rangle, x)$ is an $X$-partial trace-tree of $R$ even if $X \nsubseteq \text{obs}X_R$, when the projection of node labels to $\text{obs}X_R$ results in an $(X \cap \text{obs}X_R)$-partial trace-tree of $R$. The following facts about partial trace-trees can be checked easily:

1. If $R \preceq_S S$ and $X$ is awaits-closed for $R$, then $X$ is awaits-closed for $S$. If $R \preceq_S S$ and $T$ is an $X$-partial trace-tree of $R$, then $T$ is an $X$-partial trace-tree of $S$. Thus, inclusion of trace-trees is equivalent to inclusion of partial trace-trees.
2. The partial trace-trees of a compound module are determined from the partial trace-trees of the components: $T$ is a $X$-partial trace-tree of $R \parallel S$ iff it is a $X$-partial trace-tree of both $R$ and $S$.

3. If $((T, \lambda), x)$ is an $X$-partial trace-tree of $R$ and $Y \cap \text{intf} X_R = \emptyset$, and $u$ is a valuation to variables in $Y$, then $((T, \lambda \cdot [x \rightarrow (\lambda(x) \cup u)]), x)$ is an $(X \cup Y)$-partial trace-tree of $P$. This property is due to the nonblocking nature of reactive modules.

Let $X_0, \ldots, X_{k-1}$ be a partitioning of $\text{obs} X_P \parallel Q$ into disjoint subsets such that (1) each $X_i$ either contains only external variables of $P \parallel Q$ or contains only interface variables of $P$ or only interface variables of $Q$, and (2) $x \not\in P \parallel Q$ and $x \in X_i$ then $y \in X_j$ for some $j < i$. Let $Y_0 = \emptyset$, and for $0 \leq i < k$, $Y_{i+1} = Y_i \cup X_i$. Each such set $Y_i$ is awaiting-closed. Let $T$ be the set of pairs $((t, \lambda), x)$ such that $(t, \lambda(x))$ is a $Y_j$-partial trace-tree of $P \parallel Q$ for some $0 \leq j \leq k$. Define an ordering $< \text{ over } T$: if $(t, \lambda(x))$ is a $Y_j$-partial trace-tree with $0 \leq j < k$, and $(t, \lambda'(x))$ is a $Y_{j+1}$-partial trace-tree where (1) $\lambda'(x)$ and $\lambda(x)$ agree on values of variables in $Y_j$, and (2) $\lambda'(y) = \lambda(y)$, for all other $y \in t$, then $(t, \lambda(x)) < (t, \lambda'(x))$; and if $(t, \lambda)$ is a $Y_k$-partial trace-tree (note that a $Y_k$-partial trace-tree is also a trace-tree), and $x \in t$, and $x_n \not\in t$, such that $t \cup \{x_n\}$ is a valid tree, then $(t, \lambda) < ((t \cup \{x_n\}, \lambda \cdot [x_n \rightarrow \emptyset]), x_n)$. Clearly, the ordering $<$ is well founded. By well-founded induction with respect to $<$, we prove that every partial trace-tree in $T$ is a partial trace-tree of $P' \parallel Q'$.

The null trace-tree $\langle \emptyset, \lambda \rangle$ is by definition a partial trace-tree of both $P \parallel Q$ and $P' \parallel Q'$. Let $((t, \lambda), x)$ in $T$ be a partial trace-tree. We distinguish two cases and treat them separately:

- **Case 1.** $(t, \lambda)$ is a $Y_0$-partial trace-tree of $P \parallel Q$: Since $(t - \{x\}, \lambda) < ((t, \lambda), x)$, we have that $(t - \{x\}, \lambda)$ is a trace-tree of $P' \parallel Q'$ by induction hypothesis. Now, due to nonblocking assumption $((t, \lambda), x)$ is a $Y_0$-partial trace-tree of $P' \parallel Q'$.

- **Case 2.** $(t, \lambda)$ is a $Y_{j+1}$-partial trace-tree of $P \parallel Q$ for $j \geq 0$: Let $u$ be the valuations to the variables $Y_j$ in $\lambda(x)$. Since $(t, \lambda \cdot [x \rightarrow u], x) < ((t, \lambda), x)$, by induction hypothesis $(t, \lambda \cdot [x \rightarrow u], x)$ is a $Y_j$-partial trace-tree of $P' \parallel Q'$. By the property (2) of partial trace-trees, $(t, \lambda \cdot [x \rightarrow u], x)$ is a $Y_j$-partial trace-tree of both $P'$ and $Q'$. Note that $X_{j+1} = Y_{j+1} \setminus Y_j$ contains interface variables of at most one of $P$ and $Q$. Without loss of generality, let us assume that $X_{j+1} \subseteq \text{intf} X_P$. Thus
$X_{j+1} \cap \inf X_{Q'} = \emptyset$. By property (3) of partial trace-trees, $(\langle t, \lambda \rangle, x)$ is a $Y_{j+1}$-partial trace-tree of $Q'$. Hence, $(\langle t, \lambda \rangle, x)$ is a $Y_{j+1}$-partial trace-tree of $P \parallel Q'$. Since $P \parallel Q' \preceq_S P'$, by property (1) of partial trace-trees, $(\langle t, \lambda \rangle, x)$ is a partial trace-tree of $P'$. Again, by property (2) of partial trace-trees, $(\langle t, \lambda \rangle, x)$ is a partial trace-tree of $P' \parallel Q'$.

**Abstraction Modules.** To make use of Proposition 3.6 or Theorem 3.1 we need the specification and implementation to be structured in a similar way i.e., we assume that our implementation contains two components $P$ and $Q$, and the specification has corresponding components $P'$ and $Q'$. However, most often, this is not the case. Due to lack of detail, the specification often does not supply values to several of the implementation’s internal variables. According to the following simple property of the refinement relation, however, we can arbitrarily “enrich” the specification by composing it with new modules.

**Proposition 3.7** [Abstraction modules] For all modules $P$, $P'$, and $A$, if $P \preceq_L P' \parallel A$ and $P'$ is refinable by $P$, then $P \preceq_L P'$.

**Proof:** Follows from Proposition 2.1.

So, before applying the assume-guarantee rule, we may add modules to the specification and prove $P \preceq P' \parallel A_1 \cdots \parallel A_k$ instead of $P \preceq P'$. The new modules $A_1, \ldots, A_k$ are called abstraction modules, as they usually give high-level descriptions for some implementation components. In Chapter 5 we generalize abstraction modules to abstract constraints, and in Chapter 6 we present a concrete case study, in which abstract constraints and assume-guarantee reasoning are used extensively.
Chapter 4

Fair Branching Refinement

Refinement, as we have discussed so far, verifies only the safe behaviors of implementations against specifications. It is well known that a safe refinement implies “nothing bad happens” [AL93] (for example, “two processors cannot access the bus at the same time.”) Sometimes, it is necessary to check refinements that imply “something good happens” [AL93] (for example, “whenever there is a request, it is eventually followed by a response.”) Under certain circumstances, we need what are called fairness assumptions on system descriptions, and extensions of refinement to account for fairness.

To motivate the need for fairness, recall the specification module arbSpec of the arbiter from Chapter 3, Figure 3.1. Figure 4.1 shows an implementation wrongImpl, which looks obviously incorrect. Irrespective of the requests to the arbiter, wrongImpl stays in a state with variable state set to free, and thus neither processor is ever granted the bus. However, it is seen that wrongImpl $\preceq_L$ arbSpec (in fact, wrongImpl $\preceq_B$ arbSpec as well.) The specification arbSpec allowed implementations that wait for any number of rounds before granting a request. However, it also allows implementations like wrongImpl, which never grant at all, irrespective of getting requests. This is a classical problem in concurrency theory. The standard solution is to enhance the system modeling language with fairness conditions, which classify infinite runs of the system into fair and unfair runs.

4.1 Fairness and fair refinement

With fairness conditions added to the specification and implementation models, we are concerned with fair refinements. As one would expect, fair refinements could be
module wrongImpl
  external req1, req2: bool
interface grant1, grant2: bool
private state: {free, grt1, grt2 }
atom controls state reads state awaits req1, req2
init
  [] true \rightarrow state' := free
update
  [] true \rightarrow state' := free
endatom
atom controls grant1 awaits state
init update
  [] state' = grt1 \rightarrow grant1' := true
  [] default \rightarrow grant1' := false
endatom
atom controls grant2 awaits state
init update
  [] state' = grt2 \rightarrow grant2' := true
  [] default \rightarrow grant2' := false
endatom
endmodule

Figure 4.1: Incorrect implementation of arbiter
viewed from linear and branching perspectives. The linear framework of trace-containment generalizes naturally to fair trace-containment: \( S \) fairly trace-contains \( I \) iff it is possible to fairly generate by \( S \) every sequence of observations that can be fairly generated by \( I \). Robustness with respect to LTL, and PSPACE-completeness extend to the fair case.

It is not so obvious how to generalize the branching framework of simulation to account for fairness. Indeed, several proposals can be found in the literature. The definition suggested by Grumberg and Long [GL94], and used among others by [ASB+94, KV96], rests on the motivation that \( S \) fairly simulates \( I \) iff every Fair-\( \forall \)CTL* formula that holds for \( S \) holds also for \( I \) (the universal path quantifier of Fair-\( \forall \)CTL* ranges over fair computations only.) This definition, however, is neither robust (Fair-\( \forall \)CTL induces a weaker preorder [ASB+94], and Fair-\( \forall \)AFMC, as we show here, induces a stronger one) nor can it be checked efficiently (it is complete for PSPACE [KV96].) Following [Hoj96], we call the Grumberg-Long version of fair simulation \( \exists \)-simulation, because it can be defined as simulation where each fair computation of \( I \) is related to some fair computation of \( S \). In manual verification, by Lynch and others [LT87, Lyn96], usually a stronger notion of fair simulation is used, which we call \( \forall \)-simulation (see also [DHWT91]): for each fair computation of \( I \), every related computation of \( S \) is required to be fair.\(^1\) Again, this definition is neither robust (no logical characterization is known) nor can it be checked efficiently (it is NP-complete [Hoj96].) While both \( \exists \)-simulation and \( \forall \)-simulation are sufficient conditions for fair trace-containment, they do not provide any computational advantage (indeed, algorithms for checking \( \exists \)-simulation use subroutines for checking fair trace-containment.)

We introduce a new definition of \textit{fair simulation}, and argue for its theoretical and practical merits and its advantages over existing definitions. In order to define fair simulation without losing the locality that makes simulation useful, we go back to the basis of the branching-time approach and view simulation as a generalization of tree-containment to \textit{fair tree-containment}: we define that \( S \) fairly simulates \( I \) iff it is possible to fairly embed in the unrolling of \( S \) every tree of observations that can be fairly embedded in the unrolling of \( I \), where a tree embedding is fair if all infinite paths are mapped onto fair computations. This definition falls strictly between \( \exists \)-simulation and \( \forall \)-simulation.

Consider the implementation and specification appearing in Figure 4.2. The struc-\(^1\)

\( ^1 \)Using a similar proof technique, Lamport and others [Lam83, AL91] suggest a restricted, functional version of simulation, called refinement mapping. There, every computation of \( I \) is related to exactly one computation of \( S \); thus, \( \exists \)-simulation coincides with \( \forall \)-simulation.
tutes representing them are augmented with Büchi fairness constraints. In order to be fair, an infinite computation of $\mathcal{I}$ must visit the set $\{i_3, i_4\}$ infinitely often, and an infinite computation of $\mathcal{S}$ must visit the set $\{s_3, s'_4\}$ infinitely often. It is easy to see that $\mathcal{S}$ 3-simulates $\mathcal{I}$.

Indeed, the relation that maps each state in $\mathcal{I}$ to the set of states in $\mathcal{S}$ that agree with its observation is an 3-simulation. Nevertheless, the infinite tree generated by unwinding $\mathcal{I}$ cannot be fairly embedded in an unwinding of $\mathcal{S}$. To see this, note that every occurrence of $s_2$ in any embedding must have both $s_3$ and $s_4$ as successors. Similarly, every occurrence of $s'_2$ must have both $s'_3$ and $s'_4$ as successors. Consequently, any embedding must have an infinite unfair computation.

![Figure 4.2: Fair simulation is stronger than 3-simulation](image_url)

Consider now the implementation and specification appearing in Figure 4.3. Here, the infinite fair computations of $\mathcal{I}$ are those that visit $i_6$ infinitely often, and the infinite fair computations of $\mathcal{S}$ are those that visit $s_6$ infinitely often. Clearly, we can fairly embed in $\mathcal{S}$ the tree generated by unwinding $\mathcal{I}$. Hence, $\mathcal{S}$ fairly simulates $\mathcal{I}$. Still, $\mathcal{S}$ does not ∀-simulate $\mathcal{I}$. To see this, note that any candidate relation for ∀-simulation must relate $i_6$ to both $s_4$ and $s_5$. Then, however, the observations along the unfair computation $s_1 \cdot s_2 \cdot (s_4 \cdot s_5)^\omega$ of $\mathcal{S}$ agree with the observations of the fair computation $i_1 \cdot i_2 \cdot i_6^\omega$ of $\mathcal{I}$.

The definition of fair simulation as fair tree-containment is equivalent to an alternative, local definition that is based on games. It is well-known that $\mathcal{S}$ simulates $\mathcal{I}$ iff in a game of the protagonist $\mathcal{S}$ against the adversary $\mathcal{I}$, the protagonist can match every move of the adversary by moving to a state with the same observation. Then, $\mathcal{S}$ fairly simulates $\mathcal{I}$ iff the protagonist has a strategy such that in the limit, after $\omega$ moves, if the adversary
produces a fair computation of $\mathcal{I}$, then the protagonist produces a fair computation of $\mathcal{S}$. Consider again $\mathcal{I}$ and $\mathcal{S}$ of Figure 4.2, with the adversary starting at $i_1$ and the protagonist starting at $s_1$. We show that the adversary can produce a fair computation of $\mathcal{I}$ that the protagonist cannot match. The adversary first moves to $i_2$ and then uses the following strategy: if the protagonist replies with a move to $s_2$, then the adversary makes its next move to $i_4$, forcing the protagonist to reply with a move to $s_4$. If, on the other hand, the protagonist replies with a move to $s'_2$, the adversary makes its next move to $i_3$, forcing the protagonist to reply with a move to $s'_3$. Keeping this strategy, the adversary produces a fair computation (all computations of $\mathcal{I}$ are fair), while the protagonist, irrespective of the strategy used, produces an unfair computation, which never visits $s_3$ or $s'_3$. Hence, $\mathcal{S}$ does not fairly simulate $\mathcal{I}$. By contrast, recall, $\mathcal{S}$ 3-simulates $\mathcal{I}$: while in 3-simulation the protagonist can make use of information about the future moves of the adversary in order to produce a fair computation, in fair simulation the strategy of the protagonist must depend on the past and current moves only.

We argue that our definition of fair simulation is a suitable extension of simulation to fairness, as it preserves many of the appealing properties of simulation:

- Based on the locality in the game-theoretic definition of fair simulation, for two structures $\mathcal{I}$ and $\mathcal{S}$ with weak (Büchi) or strong (Streett) fairness constraints, it can be checked in time polynomial in $\mathcal{I}$ and $\mathcal{S}$ whether $\mathcal{S}$ fairly simulates $\mathcal{I}$. The algorithm, which employs tree automata, is presented in Section 4.4.

- Since fair simulation captures fair tree-containment, it allows a logical characteriza-
CHAPTER 4. FAIR BRANCHING REFINEMENT

...tion: $S$ fairly simulates $I$ iff every Fair-$\forall$AFMC formula that holds in $S$ holds also in $I$. This is shown in Section 4.5.

- Fair simulation implies fair trace-containment, and thus provides an efficiently-computable sufficient condition for checking fair trace-containment. There is evidence that most practical specifications $\forall$-simulate their implementations. Since fair simulation is implied by $\forall$-simulation, the fair simulation condition can be used as an efficient check to verify distributed protocols that have been verified using $\forall$-simulation [Lam83, LS93, Lyn96].

- In the degenerate case of vacuous fairness constraints, fair simulation coincides with simulation. In the degenerate case of deterministic systems, fair simulation coincides with fair trace-containment.

We note that in process algebra, several other preorders and equivalences on state-transition systems have been extended to account for fairness, including failure preorders [BKO87] and testing preorders [Hen87, EB95, NC95]. From an algorithmic point of view, these preorders are closely related to (fair) trace-containment, and the problems of checking them are complete for PSPACE.

4.2 Definitions

For most of this chapter (except Section 4.7), we deal with systems at the level of Kripke structures. Recall the definition of Kripke structures from Chapter 2, Definition 2.2.

Single initial states

For the purposes of this chapter, we assume that a Kripke structure has a single initial state. The single initial state assumption is for technical simplicity. Problems on Kripke structures with multiple initial states can be easily transformed into problems on Kripke structures with single initial states. For example, let $K_1 = \langle \Gamma, W_1, \hat{W}_1, R_1, L_1 \rangle$, and $K_2 = \langle \Gamma, W_2, \hat{W}_2, R_2, L_2 \rangle$ be two Kripke structures with the same observation alphabet. Suppose we wish to check if $K_1 \preceq_B K_2$. Let $K'_1 = \langle \Gamma', W'_1, \{\hat{w}_1\}, R'_1, L'_1 \rangle$ with the following components:

- $\Gamma' = \Gamma \cup \{\gamma\}$, such that $\gamma \not\in \Gamma$. 

CHAPTER 4. FAIR BRANCHING REFINEMENT

• $W'_1 = W'_1 \cup \{\hat{w}_1\}$, such that $\hat{w}_1 \notin W_1$.

• $R'_1 = R_1 \cup \{(\hat{w}_1, w) \mid w \in \hat{W}\}$.

• $L'_1(\hat{w}_1)$ is defined to be $\gamma$. For all $w \in W_1$, $L'_1(w) = L_1(w)$.

Intuitively, $K'_1$ is obtained from $K_1$ by adding a new initial state $\hat{w}_1$ with a distinguished observation $\gamma$, and adding transitions from $\hat{w}_1$ to every state in $W$. Let $K'_2 = \langle \Gamma', W'_2, \{\hat{w}_2\}, R'_2, L'_2 \rangle$ be obtained similarly. Then, it is checked that $K_1 \preceq_B K_2$ iff $K'_1 \preceq_B K'_2$.

For the rest of this chapter, whenever $K = \langle \Gamma, W, \hat{W}, R, L \rangle$ is a Kripke structure, we assume that $\hat{W} = \{\hat{w}\}$. Since there is a single initial state, we drop the set notation for $\hat{W}$ and write $K = \langle \Gamma, W, \hat{w}, R, L \rangle$.

**Fair Kripke structures**

We need to enhance Kripke structures to include fairness conditions. Let $K = \langle \Gamma, W, \hat{w}, R, L \rangle$ be a Kripke structure. Recall that, for a state $w \in \Sigma$, a $w$-run of $K$ is a finite or infinite sequence $\overline{w} = w_0 \cdot w_1 \cdot w_2 \cdots$ of states $w_i \in \Sigma$ such that $w_0 = w$ and $R(w_i, w_{i+1})$ for all $i \geq 0$. We write $\inf(\overline{w})$ for the set of states that occur infinitely often in $\overline{w}$.

A **fairness constraint** for $K$ is a function that maps every run of $K$ to the binary set $\{\text{fair}, \text{unfair}\}$. We consider three kinds of fairness constraints:

• The **vacuous** constraint maps every run of $K$ to $\text{fair}$.

• A **Büchi** constraint $F$ is specified by a set $F_B \subseteq \Sigma$ of states. Then, for a run $\overline{w}$ of $K$ we have $F(\overline{w}) = \text{fair}$ iff $\inf(\overline{w}) \cap F_B \neq \emptyset$.

• A **Streett** constraint $F$ is specified by a set $F_S \subseteq 2^W \times 2^W$ of pairs of state sets. Then, for a run $\overline{w}$ of $K$ we have $F(\overline{w}) = \text{fair}$ iff for every pair $\langle l, r \rangle \in F_S$, if $\inf(\overline{w}) \cap l \neq \emptyset$ then $\inf(\overline{w}) \cap r \neq \emptyset$.

A fair structure $K = \langle K, F \rangle$ consists of a structure $K$ and a fairness constraint $F$ for $K$. The fair structure $K$ is a Büchi structure if $F$ is a Büchi constraint, and $K$ is a Streett structure if $F$ is a Streett constraint. In particular, every Büchi structure is also a Streett structure. For a state $w \in \Sigma$, a fair $w$-run of $K$ is either a finite $w$-run of $K$ or an infinite $w$-run $\overline{w}$ of $K$ such that $F(\overline{w}) = \text{fair}$. A fair run of $K$ is a fair $\hat{w}$-run, for some initial state $\hat{w} \in \hat{W}$. A fair trace of $K$ is a trace of $K$ that is witnessed by a fair run of $K$. 
CHAPTER 4. FAIR BRANCHING REFINEMENT

In the following, we consider two structures $K_1 = \langle \Gamma, W_1, \dot{w}_1, R_1, L_1 \rangle$ and $K_2 = \langle \Gamma, W_2, \dot{w}_2, R_2, L_2 \rangle$ over the same alphabet, and two fair structures $\mathcal{K}_1 = \langle K_1, F_1 \rangle$ and $\mathcal{K}_2 = \langle K_2, F_2 \rangle$.

Trace-containment and fair trace-containment

Recall that the structure $K_2$ trace-contains the structure $K_1$ if every trace of $K_1$ is also a trace of $K_2$ (or, equivalently, if every finite trace of $K_1$ is also a finite trace of $K_2$.) The problem of checking if $K_2$ trace-contains $K_1$ is complete for PSPACE [SM73].

The fair structure $\mathcal{K}_2$ fairly trace-contains the fair structure $\mathcal{K}_1$ if every fair trace of $\mathcal{K}_1$ is also a fair trace of $\mathcal{K}_2$. For vacuous constraints $F_1$ and $F_2$, fair trace-containment coincides with trace-containment. For Büchi or Streett constraints $F_1$ and $F_2$, the problem of checking if $\mathcal{K}_2$ fairly trace-contains $\mathcal{K}_1$ is complete for PSPACE [SVW87, Saf88].

Simulation

In Chapter 3, we defined when a transition constraint simulates another. In this section, we restate the definition of simulation with respect to Kripke structures.

A binary relation $S \subseteq W_1 \times W_2$ is a simulation of $K_1$ by $K_2$ if the following two conditions hold [Mil71]:

1. If $S(w_1, w_2)$, then $L_1(w_1) = L_2(w_2)$.
2. If $S(w_1, w_2)$ and $R_1(w_1, w'_1)$, then there is a state $w'_2 \in W_2$ such that $R_2(w_2, w'_2)$ and $S(w'_1, w'_2)$.

The structure $K_2$ simulates the structure $K_1$ if there is a simulation $S$ of $K_1$ by $K_2$ such that $S(\dot{w}_1, \dot{w}_2)$. The problem of checking if $K_2$ simulates $K_1$ can be solved in time $O(|R_1| \cdot |W_2| + |R_2| \cdot |W_1|)$ [BP95, HHK95]. If $K_2$ simulates $K_1$, then $K_2$ trace-contains $K_1$. If $K_1$ and $K_2$ are both deterministic, then simulation coincides with trace-containment.

The following three alternative definitions of simulation are equivalent to the definition above.

The game-theoretic view. Consider a two-player game whose positions are pairs $\langle w_1, w_2 \rangle \in W_1 \times W_2$ of states. The initial position is $\langle \dot{w}_1, \dot{w}_2 \rangle$. The game is played between an adversary and a protagonist and it proceeds in a sequence of rounds. In each round, if $\langle w_1, w_2 \rangle$ is the current position, first the adversary updates the first component $w_1$ to any $R_1$-successor $w'_1$,
and then the protagonist updates the second component $w_2$ to some $R_2$-successor $w'_2$ such that $L_1(w'_1) = L_2(w'_2)$. If no such $w'_2$ exists, then the protagonist loses. If the game proceeds ad infinitum, for $\omega$ rounds, then the adversary loses. It is easy to see that $K_2$ simulates $K_1$ iff the protagonist has a winning strategy.

The tree-containment view. Due to Proposition 3.2, $K_2$ simulates $K_1$ iff every trace-tree of $K_1$ is also a trace-tree of $K_2$ (or, equivalently, iff every finite trace-tree of $K_1$ is also a finite trace-tree of $K_2$).

The temporal-logic view. The three branching-time logics $\forall$CTL, $\forall$CTL*, and $\forall$AFMC are the fragments of CTL, CTL*, and the alternation-free $\mu$-calculus that do not contain existential path quantifiers [BBL92, GL94]. It is well-known that $K_2$ simulates $K_1$ iff for every formula $\psi$ of $\forall$CTL (or $\forall$CTL* or $\forall$AFMC), if $K_2$ satisfies $\psi$, then $K_1$ satisfies $\psi$. It follows that simulation is the weakest preorder that preserves any of these three logics.

Previous definitions of fair simulation

In the literature, we find several extensions of simulation that account for fairness constraints. In particular, the following two extensions have been studied and used extensively.

$\exists$-simulation [GL94]. A binary relation $S \subseteq W_1 \times W_2$ is an $\exists$-simulation of $\mathcal{K}_1$ by $\mathcal{K}_2$ if the following two conditions hold:

1. If $S(w_1, w_2)$, then $L_1(w_1) = L_2(w_2)$.

2. If $S(w_1, w_2)$, then for every fair $w_1$-run $\bar{w} = u_0 \cdot u_1 \cdot u_2 \cdots$ of $\mathcal{K}_1$, there is a fair $w_2$-run $\bar{w}' = u'_0 \cdot u'_1 \cdot u'_2 \cdots$ of $\mathcal{K}_2$ such that $\bar{w}'$ $S$-matches $\bar{w}$; that is, $|\bar{w}'| = |\bar{w}|$ and $S(u_i, u'_i)$ for all $0 \leq i \leq |\bar{w}|$.

Clearly, every $\exists$-simulation of $\mathcal{K}_1$ by $\mathcal{K}_2$ is a simulation of $K_1$ by $K_2$. The fair structure $\mathcal{K}_2$ $\exists$-simulates the fair structure $\mathcal{K}_1$ if there is an $\exists$-simulation $S$ of $\mathcal{K}_1$ by $\mathcal{K}_2$ such that $S(\hat{w}_1, \hat{w}_2)$.

For Büchi or Streett constraints $F_1$ and $F_2$, the problem of checking if $\mathcal{K}_2$ $\exists$-simulates $\mathcal{K}_1$ is complete for PSPACE [KV96]. $\exists$-simulation is the weakest preorder that preserves Fair-$\forall$CTL*, where the universal path quantifiers range over the fair runs only:
\( \mathcal{K}_2 \equiv \text{-simulates} \mathcal{K}_1 \) iff for every formula \( \psi \) of \( \text{Fair-}\forall\text{CTL}^* \), if \( \mathcal{K}_2 \) satisfies \( \psi \), then \( \mathcal{K}_1 \) satisfies \( \psi \) [GL94]. By contrast, \( \equiv \text{-similarity} \) is not the coarsest abstraction that preserves \( \text{Fair-}\forall\text{CTL} \): there are two Büchi structures \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \), such that \( \mathcal{K}_1 \) satisfies every \( \forall\text{CTL} \) formula satisfied by \( \mathcal{K}_2 \), but \( \mathcal{K}_2 \) does not \( \equiv \text{-simulate} \ \mathcal{K}_1 \) [ASB+94]. In addition, \( \equiv \text{-similarity} \) does not preserve \( \text{Fair-}\forall\text{AFMC} \): as we show in Section 4.5, there are two Büchi structures \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \), and a \( \text{Fair-}\forall\text{AFMC} \) formula \( \psi \), such that \( \mathcal{K}_2 \equiv \text{-simulates} \ \mathcal{K}_1 \), and \( \mathcal{K}_2 \) satisfies \( \psi \), but \( \mathcal{K}_1 \) does not satisfy \( \psi \).

\( \forall \text{-simulation} \) [LT87, DHWT91]. A binary relation \( S \subseteq W_1 \times W_2 \) is a \( \forall \text{-simulation} \) of \( \mathcal{K}_1 \) by \( \mathcal{K}_2 \) if the following two conditions hold:

1. \( S \) is a simulation of \( \mathcal{K}_1 \) by \( \mathcal{K}_2 \).

2. If \( S(w_1, w_2) \), then for every fair \( w_1\text{-run} \overline{w} \) of \( \mathcal{K}_1 \) and every \( w_2\text{-run} \overline{w}' \) of \( \mathcal{K}_2 \), if \( \overline{w}' \) \( S \)-matches \( \overline{w} \), then \( \overline{w}' \) is a fair \( w_2\text{-run} \) of \( \mathcal{K}_2 \).

If \( S \) is a \( \forall \text{-simulation} \) of \( \mathcal{K}_1 \) by \( \mathcal{K}_2 \), then \( S(w_1, w_2) \) implies \( L_1(w_1) = L_2(w_2) \) (since \( S \) has to be a simulation, by definition.) Further, if \( S(w_1, w_2) \), and \( \overline{w} \) is a fair-\( w_1 \) run of \( \mathcal{K}_1 \), then there exists a \( w_2 \) \( \text{run} \overline{w}' \) of \( \mathcal{K}_2 \) that \( S \)-matches \( \overline{w} \). In addition, such a \( \text{run} \overline{w}' \) must also be fair for \( \mathcal{K}_2 \), by definition. Thus, every \( \forall \text{-simulation} \) of \( \mathcal{K}_1 \) by \( \mathcal{K}_2 \) is an \( \equiv \text{-simulation} \) of \( \mathcal{K}_1 \) by \( \mathcal{K}_2 \). The fair structure \( \mathcal{K}_2 \forall \text{-simulates} \) the fair structure \( \mathcal{K}_1 \) if there is a \( \forall \text{-simulation} \) \( S \) of \( \mathcal{K}_2 \) by \( \mathcal{K}_1 \) such that \( S(w_1, w_2) \).

For vacuous constraints \( F_1 \) and \( F_2 \), \( \forall \text{-similarity} \) coincides with similarity. For Büchi or Streett constraints \( F_1 \) and \( F_2 \), the problem of checking whether \( \mathcal{K}_2 \forall \text{-simulates} \mathcal{K}_1 \) is NP-complete [Hoj96]. \( \forall \text{-simulation} \) is widely used for proving abstraction hierarchies of distributed protocols [Lyn96]. In practice, Condition 2 is often replaced by a stronger condition that relates the two fairness constraints: for example, if \( F_1 \) and \( F_2 \) are both Büchi constraints, then a sufficient condition for 2. is that if \( S(w_1, w_2) \) and \( w_1 \in F_1 \), then \( w_2 \in F_2 \). Particularly popular is a functional version of simulation: the simulation \( S \) is a refinement mapping if whenever \( S(w_1, w_2) \) and \( S(w_1, w'_2) \), then \( w_2 = w'_2 \) [AL91]. If \( S \) is a refinement mapping, then \( S \) is a \( \forall \text{-simulation} \) iff \( S \) is an \( \equiv \text{-simulation} \). There is no known logical characterization for \( \forall \text{-simulation} \).
Our definition of fair simulation

Recall the simulation game of the protagonist $K_2$ against the adversary $K_1$. A strategy $\tau$ for the protagonist is a partial function from $(W_1 \times W_2)^* \times W_1$ to $W_2$: if the game so far has produced the sequence $\pi \in (W_1 \times W_2)^*$ of positions, and the adversary moves to $w$, then the strategy $\tau$ instructs the protagonist to move to $w' = \tau(\pi, w)$, thus resulting in the new position $\langle w, w' \rangle$. Given a finite or an infinite sequence $\overline{w} = w_0 \cdot w_1 \cdot w_2 \cdots$ of states $w_i \in W_1$, the outcome $\tau[\overline{w}] = w'_0 \cdot w'_1 \cdot w'_2 \cdots$ of the strategy $\tau$ is the finite or the infinite sequence of states $w'_i \in W_2$ such that $|\tau[\overline{w}]| = |\overline{w}|$ and $w'_i = \tau(w_0, w'_0, w_1, w'_1, \ldots, w_{i-1}, w'_{i-1}, w_i)$ for all $i \geq 0$. A binary relation $S \subseteq W_1 \times W_2$ is a fair simulation of $K_1$ by $K_2$ if the following two conditions hold:

1. If $S(w_1, w_2)$, then $L_1(w_1) = L_2(w_2)$.

2. There exists a strategy $\tau$ such that if $S(w_1, w_2)$ and $\overline{w}$ is a fair $w_1$-run of $K_1$, the outcome $\tau[\overline{w}]$ is a fair $w_2$-run of $K_2$ and $\tau[\overline{w}]$ $S$-matches $\overline{w}$. We say that $\tau$ is a witness to the fair simulation $S$.

Clearly, every fair simulation of $K_1$ by $K_2$ is an $\exists$-simulation of $K_1$ by $K_2$, and every $\forall$-simulation of $K_1$ by $K_2$ is a fair simulation of $K_1$ by $K_2$. The fair structure $K_2$ fairly simulates the fair structure $K_1$ if there is a fair simulation $S$ of $K_1$ by $K_2$ such that $S(\hat{w}_1, \hat{w}_2)$.

In Section 4.4, we suggest an algorithm for checking whether two fair structures are fairly similar. The algorithm reduces the fair-similarity problem to the nonemptiness problem of tree automata. Known results about tree automata [Rab70, PR89] then imply that in Condition 2 above, if a required strategy exists, then there exists a finite-state strategy; that is, a strategy produced by a finite-state machine. Moreover, for Büchi structures, there exists a memoryless strategy; that is, a strategy that decides its next move based only on the current position and the current move of the adversary. On the other hand, for Streett structures, there may not exist a memoryless strategy. To see this, consider the two Streett structures shown below: the infinite fair runs of $I$ are those that visit $i_2$ infinitely often, and the infinite fair runs of $S$ are those that visit both $s_2$ and $s'_2$ infinitely often (i.e., $S$ has the Streett constraint $\{(\{s_1\}, \{s_2\}), (\{s_1\}, \{s'_2\})\}$). In order to satisfy Condition 2, the protagonist must visit both $s_2$ and $s'_2$ infinitely often. Hence, it cannot follow a memoryless strategy.
CHAPTER 4. FAIR BRANCHING REFINEMENT

\[
I:
\begin{array}{c}
  \circ \quad i_1 \\
    \quad a \\
  \quad i_2 \\
\end{array}
\quad S:
\begin{array}{c}
  \circ \quad s_1 \\
    \quad a \\
  \quad s_2 \\
\end{array}
\]

Figure 4.4: Fair simulation does not have memoryless strategy

As we demonstrated in Section 4.1, fair similarity falls strictly between ∃-similarity and ∀-similarity. Hence the following two propositions.

**Proposition 4.1** For all fair structures \(K_1\) and \(K_2\), if \(K_2\) fairly simulates \(K_1\), then \(K_2\) ∃-simulates \(K_1\). There are two Büchi structures \(K_1\) and \(K_2\) such that \(K_2\) ∃-simulates \(K_1\), but \(K_2\) does not fairly simulate \(K_1\).

**Proposition 4.2** For all fair structures \(K_1\) and \(K_2\), if \(K_2\) ∀-simulates \(K_1\), then \(K_2\) fairly simulates \(K_1\). There are two Büchi structures \(K_1\) and \(K_2\) such that \(K_2\) fairly simulates \(K_1\), but \(K_2\) does not ∀-simulate \(K_1\).

Fair trace containment is weaker than ∃-simulation [GL94]. However, for deterministic structures, all the definitions of fair similarity collapse and coincide with trace containment.

**Proposition 4.3** For all deterministic structures \(K_1\) and \(K_2\) the following four statements are equivalent: (1) \(K_2\) fairly trace contains \(K_1\). (2) \(K_2\) ∃-simulates \(K_1\). (3) \(K_2\) fairly simulates \(K_1\). (4) \(K_2\) ∀-simulates \(K_1\).

**Proof:** Following Proposition 4.1 and Proposition 4.2, we only need to prove that (1) implies (4). Suppose \(K_2\) fairly trace contains \(K_1\). Then, for every fair run \(\bar{w} = u_0 \cdot u_1 \cdot u_2 \cdots\) of \(K_1\), there is a fair run \(\bar{w}' = u'_0 \cdot u'_1 \cdot u'_2 \cdots\) of \(K_2\) such that \(L_1(\bar{w}) = L_2(\bar{w}')\). Define a partial function \(f : W_1 \rightarrow 2^{W_2}\), such that

\[
f(s) = \{ s' \mid \text{there exist runs } \bar{w} = u_0 \cdot u_1 \cdot u_2 \cdots \text{ of } K_1, \text{ and } \bar{w}' = u'_0 \cdot u'_1 \cdot u'_2 \cdots s' \text{ of } K_2 \text{ such that } L_1(\bar{w}) = L_2(\bar{w}')\}
\]

Consider the relation

\[
S = \{(s, s') \mid s' \in f(s)\}.
\]
Since \( K_2 \) is deterministic, for every run \( \overline{w} = u_0 \cdot u_1 \cdot u_2 \cdots \) of \( K_1 \), there is a unique \( K_2 \) run \( \overline{w}' = u'_0 \cdot u'_1 \cdot u'_2 \cdots \) of \( K_2 \) such that for all \( i \geq 0 \), we have \( u'_i \in f(u_i) \). Further, by definition of \( f \), we have that \( \overline{w}' \) is a fair \( K_2 \) run whenever \( \overline{w} \) is a fair \( K_1 \) run. Thus, \( S \) is a \( \forall \)-simulation. Further, \( S \) contains the pair of initial states \((\hat{w}_1, \hat{w}_2)\). Thus, \( K_2 \) \( \forall \)-simulates \( K_1 \).

It is also worth noting that for vacuous fairness constraints, fair similarity, \( \exists \)-similarity, and \( \forall \)-similarity all coincide with similarity. As with similarity, there are three alternative definitions of fair similarity that are equivalent to the definition above.

The game-theoretic view. If the simulation game is played for \( \omega \) rounds, then the protagonist wins. In that case, the adversary produces an infinite run of \( K_1 \) and the protagonist produces an infinite run of \( K_2 \). In a fair game, the winning condition is modified as follows: if the game is played for \( \omega \) rounds, then the protagonist wins iff either the adversary does not produce a fair run of \( K_1 \), or the protagonist produces a fair run of \( K_2 \).

It is easy to see that \( K_2 \) fairly simulates \( K_1 \) iff the protagonist has a winning strategy in the fair game.

The tree-containment view. Given a fair structure \( K = \langle K, F \rangle \), a fair run-tree of \( K \) is a run-tree \( \langle t, \lambda \rangle \) of \( K \) such that every path of \( t \) generates a fair run of \( K \). A fair trace-tree of \( K \) is a trace-tree of \( K \) that is witnessed by a fair run-tree of \( K \). The following proposition gives a fully abstract tree semantics to fair simulation.

**Proposition 4.4** A fair structure \( K_2 \) fairly simulates a fair structure \( K_1 \) iff every fair trace-tree of \( K_1 \) is also a fair trace-tree of \( K_2 \).

**Proof:** Suppose \( K_2 \) fairly simulates \( K_1 \). That is, there exists a strategy \( \tau \) and a relation \( S \subseteq W_1 \times W_2 \) such that \( S(\hat{w}_1, \hat{w}_2) \), and conditions (1) and (2) of the definition are satisfied. Consider a fair trace-tree \( \langle t, \lambda_1' \rangle \) of \( K_1 \), and its witnessing run-tree \( \langle t, \lambda_1 \rangle \). Using \( \tau \), we can construct a run-tree \( \langle t, \lambda_2 \rangle \) by defining \( \lambda_2 \) inductively as follows: (1) \( \lambda_2(\epsilon) = \hat{w}_2 \), and (2) for every node \( n_0 n_1 n_2 \cdots n_i \in t \), the label \( \lambda_2(n_0 n_1 n_2 \cdots n_{i-1} n_i) \) is given by

\[
\tau(\hat{w}_1, \hat{w}_2, \lambda_1(n_0), \lambda_2(n_0), \ldots, \lambda_1(n_0 \cdots n_{i-1}), \lambda_2(n_0 \cdots n_{i-1}), \lambda_1(n_0 \cdots n_{i-1} n_i)).
\]

Note that \( \langle t, \lambda_2 \rangle \) is a fair run-tree of \( K_2 \), because \( \tau \) is a witness to a fair simulation. We can now construct the fair trace-tree \( \langle t, \lambda_2 \circ L_2 \rangle \) of \( K_2 \), which is identical to \( \langle t, \lambda_1 \rangle \).

Conversely, suppose every fair trace-tree of \( K_1 \) is also a fair trace-tree of \( K_2 \). Construct a maximal fair run-tree \( \langle t, \lambda_1 \rangle \) of \( K_1 \) (i.e., every fair run of \( K_1 \) is a path in
The corresponding trace tree $\langle t, \lambda_1 \circ L_1 \rangle$ is also a trace tree of $\mathcal{K}_2$. Hence, it has a witnessing fair run-tree $\langle t, \lambda_2 \rangle$ of $\mathcal{K}_2$. Consider the relation

$$S = \{ (w_1, w_2) \mid \text{for some } x \in t, w_1 = \lambda_1(x) \text{ and } w_2 = \lambda_2(x) \}$$

We claim that $S$ is a fair simulation from $\mathcal{K}_1$ to $\mathcal{K}_2$. Indeed, we can construct the required strategy $\tau$, inductively as follows: (1) $\tau(\hat{w}_1) = \hat{w}_2$, and (2) for every run $\hat{w}_1 \cdot \lambda_1(n_0) \cdot \lambda_1(n_0n_1) \cdots \lambda_1(n_0n_1 \cdots n_i)$, we have

$$\tau(\hat{w}_1, \hat{w}_2, \lambda_1(n_0), \lambda_2(n_0), \ldots, \lambda_1(n_0n_1 \cdots n_{i-1}n_i)) = \lambda_2(n_0n_1n_2 \cdots n_i)$$

Since $\langle t, \lambda_2 \rangle$ is a fair run-tree of $\mathcal{K}_2$, the strategy $\tau$ is a witnessing strategy for $S$.

**The temporal-logic view.** In Section 4.5, we show that $\mathcal{K}_2$ fairly simulates $\mathcal{K}_1$ iff for every formula $\psi$ of Fair-$\forall$AFMC, if $\mathcal{K}_2$ satisfies $\psi$, then $\mathcal{K}_1$ satisfies $\psi$. It follows that fair similarity is the coarsest abstraction that preserves the fair universal alternation-free $\mu$-calculus.

### 4.3 Init simulations

In this section, we consider weak versions of $\exists$-similarity, $\forall$-similarity, and fair similarity, where the $S$-matching restricted to fair runs that start at the initial states of $\mathcal{K}_1$ and $\mathcal{K}_2$. We refer to the weak versions as init-$\exists$-similarity, init-$\forall$-similarity, and init-fair similarity, and investigate two properties of such init-similarities:

- **Monotonicity:** given a relation $S$ that is an init-$\exists$-simulation (init-$\forall$-simulation, init-fair simulation) and a finite simulation $S' \supseteq S$, is $S'$ guaranteed to be an init-$\exists$-simulation (init-$\forall$-simulation, init-fair simulation) as well?

- **Init-sensitivity:** does the existence of an init-$\exists$-simulation (init-$\forall$-simulation, init-fair simulation) from $\mathcal{K}_1$ to $\mathcal{K}_2$ guarantee the existence of an $\exists$-simulation ($\forall$-simulation, fair simulation)?

As we shall see in this section, fair simulation enjoys both monotonicity and init-sensitivity. This leads us, as detailed in Section 4.4, to an efficient algorithm for checking fair simulation between two fair structures. In addition, the lack of either monotonicity or init-sensitivity for init-$\exists$-simulation and init-$\forall$-simulation gives some intuition for why
they are computationally hard to check. We first define init-$\exists$-simulation, init-$\forall$-simulation and init-fair simulation formally. As in the previous section, we consider two structures over the same alphabet $K_1 = \langle T, W_1, \dot{w}_1, R_1, L_1 \rangle$ and $K_2 = \langle T, W_2, \dot{w}_2, R_2, L_2 \rangle$, and two fair structures $K_1 = \langle K_1, F_1 \rangle$ and $K_2 = \langle K_2, F_2 \rangle$.

**init-$\exists$-simulation** A binary relation $S \subseteq W_1 \times W_2$ is an *init-$\exists$-simulation* of $K_2$ by $K_1$ if the following three conditions hold:

1. $S(\dot{w}_1, \dot{w}_2)$.
2. If $S(w_1, w_2)$, then $L_1(w_1) = L_2(w_2)$.
3. For every fair run $\bar{w}$ of $K_1$ there exists an $S$-matching fair run $\bar{w}'$ of $K_2$.

**init-$\forall$-simulation** A binary relation $S \subseteq W_1 \times W_2$ is an *init-$\forall$-simulation* of $K_2$ by $K_1$ if the following two conditions hold:

1. $S$ is a simulation of $K_2$ by $K_1$.
2. For every fair run $\bar{w}$ of $K_1$ and every run $\bar{w}'$ of $K_2$, if $\bar{w}'$ $S$-matches $\bar{w}$, then $\bar{w}'$ is a fair run of $K_2$.

**init-fair simulation** A binary relation $S \subseteq W_1 \times W_2$ is an *init-fair simulation* of $K_2$ by $K_1$ if the following three conditions hold:

1. $S(\dot{w}_1, \dot{w}_2)$.
2. If $S(w_1, w_2)$, then $L_1(w_1) = L_2(w_2)$.
3. There exists a strategy $\tau$ such that for every fair run $\bar{w}$ of $K_1$, the outcome $\tau[\bar{w}]$ is a fair run of $K_2$ and $\tau[\bar{w}]$ $S$-matches $\bar{w}$.

We first consider monotonicity of init-similarities.

**Proposition 4.5** For all fair structures $K_1 = \langle K_1, F_1 \rangle$ and $K_2 = \langle K_2, F_2 \rangle$, if $S$ is an init-$\exists$-simulation (init-fair simulation) of $K_1$ by $K_2$, and $S' \supseteq S$ is a simulation of $K_1$ by $K_2$, then $S'$ is also an init-$\exists$-simulation (init-fair simulation) of $K_1$ by $K_2$.
CHAPTER 4. FAIR BRANCHING REFINEMENT

Proof: Let $S$ be an init-∃-simulation from $K_1$ to $K_2$. Given a fair run $w$ of $K_1$ we know that there exists an $S$-matching fair run $w'$ of $K_2$. Since $w'$ also $S'$-matches $w$, we have that $S'$ is an init-∃-simulation as well, and we are done.

Similarly, if $S$ is an init-fair simulation, there is a witnessing strategy $τ$ such that given a fair run $w$ of $K_1$, the outcome $τ[w]$ is a fair run of $K_2$ and $τ[w]$ $S$-matches $w$. Since $τ[w]$ $S'$-matches $w$, we have that $S'$ is an init-fair simulation as well, with the same strategy $τ$ as witness.

Proposition 4.6 There are two Büchi structures $K_1$ and $K_2$, an init-∀-simulation $S$ from $K_1$ to $K_2$, and a simulation $S' \supseteq S$ from $K_1$ to $K_2$, such that $S'$ is not an init-∀-simulation from $K_1$ to $K_2$.

Proof: Consider the structures in Figure 4.4, and replace the Streett constraint of $S$ with the Büchi constraint $\{s_2\}$. Thus, an infinite run of $S$ is fair iff it visits the state $s_2$ infinitely often. Consider the relations $S = \{(i_1, s_1), (i_2, s_2)\}$ and $S' = \{(i_1, s_1), (i_2, s_2), (i_2, s_2)'\}$. Both $S$ and $S'$ are simulations from $I$ to $S$, and $S' \supseteq S$. There is a single infinite fair run for $I$, namely $(i_1 \cdot i_2)^\omega$. For this run, there is only one $S$-matching run of $S$, namely $(s_1 \cdot s_2)^\omega$, which is fair. However, the run $(i_1 \cdot i_2)^\omega$ has the $S'$-matching run $(s_1 \cdot s_2)'^\omega$, which is not fair. Thus, while $S$ is an init-∀-simulation from $I$ to $S$, the relation $S'$ is not. Next, we consider init-sensitivity of init-similarities.

Proposition 4.7 For all fair structures $K_1 = \langle K_1, F_1 \rangle$ and $K_2 = \langle K_2, F_2 \rangle$, $K_2$ init-fair simulates (init-∀-simulates) $K_1$ iff $K_2$ fair simulates (∀-simulates) $K_1$.

Proof: Let $S$ be an init-fair simulation from $K_1$ to $K_2$, with strategy $τ$ as the witness. We define

$$S' = \{ (w, w') | (w, w') \in S \text{ and there exists a finite run } w = w_1 \cdots w \text{ of } K_1, \text{ with } τ[w] = w_2 \cdots w' \}.$$ 

Intuitively, $S'$ is the subset of pairs from $S$ that are used by the protagonist in the fair simulation game. It is easy to see that $S'$ is a fair simulation from $K_1$ to $K_2$. Similarly, let $S$ be an init-∀-simulation from $K_1$ to $K_2$. We define

$$S' = \{ (w, w') | (w, w') \in S \text{ and there exist finite runs } w = w_1 \cdots w' \text{ of } K_1 \text{ and } w' = w_2 \cdots w \text{ of } K_2, \text{ such that } w' S - \text{matches } w \}.$$
<table>
<thead>
<tr>
<th>Init Simulation</th>
<th>Monotonicity</th>
<th>Init-sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists$-simulation</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>$\forall$-simulation</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>fair simulation</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 4.1: Properties of Init Simulations

It is easy to see that $S'$ is a $\forall$-simulation from $K_1$ to $K_2$.

Before investigating the init-sensitivity of $\exists$-similarity, we first note the following strong relation between init-$\exists$-similarity and fair trace containment.

**Proposition 4.8** For all fair structures $K_1 = \langle K_1, F_1 \rangle$ and $K_2 = \langle K_2, F_2 \rangle$, $K_2$ fairly trace contains $K_1$ iff $K_2$ init-$\exists$-simulates $K_1$.

**Proof:** It is easy to see that init-$\exists$-simulation implies fair trace containment. To prove the other direction, assume $K_2$ fairly trace contains $K_1$. Then, we claim that the relation

$$S = \{(w_1, w_2) | w_1 \in W_1, w_2 \in W_2, \text{ and } L_1(w_1) = L_2(w_2)\}$$

is an init-$\exists$-simulation. Indeed, for every fair run of $K_1$, we can produce an $S$-matching fair run of $K_2$.

Finally, Proposition 4.8 and the well known fact that trace containment is strictly weaker than $\exists$-simulation [GL94], lead to the following proposition.

**Proposition 4.9** There are two Büchi structures $K_1$ and $K_2$, such that $K_2$ init-$\exists$-simulates $K_1$, but $K_2$ does not $\exists$-simulate $K_1$.

It follows (see Table 1) that only fair simulation enjoys both monotonicity and init-sensitivity. Checking if a given relation is an $\exists$-simulation is PSPACE-complete [KV96]. Checking if a given relation is a $\forall$-simulation can be done in PTIME, but because of lack of monotonicity, finding a good candidate relation is hard (NP-complete [Ho96].) In the next section, we exploit the monotonicity and init sensitivity of fair simulation to obtain an efficient algorithm for checking if there is a fair simulation from one fair structure to another.
CHAPTER 4. FAIR BRANCHING REFERENCE

4.4 Checking fair simulation

Given two structures $K_1 = \langle \Gamma, W_1, \hat{w}_1, R_1, L_1 \rangle$ and $K_2 = \langle \Gamma, W_2, \hat{w}_2, R_2, L_2 \rangle$, and two fair structures $K_1 = \langle K_1, F_1 \rangle$ and $K_2 = \langle K_2, F_2 \rangle$, we present an automata-based algorithm that checks, in time polynomial in $K_1$ and $K_2$, whether $K_2$ fairly simulates $K_1$.

A maximal simulation is a binary relation $\hat{S} \subseteq W_1 \times W_2$, such that (1) $\hat{S}$ is a simulation from $K_1$ to $K_2$ and (2) for every simulation $S$ from $K_1$ to $K_2$, we have $S \subseteq \hat{S}$. The following proposition reduces the problem of checking if there is a fair simulation from $K_1$ to $K_2$, to checking if the (unique) maximal simulation from $K_1$ to $K_2$ is an init-fair simulation from $K_1$ to $K_2$.

Proposition 4.10 For all fair structures $K_1 = \langle K_1, F_1 \rangle$ and $K_2 = \langle K_2, F_2 \rangle$, if $\hat{S}$ is the maximal simulation from $K_1$ to $K_2$, then $K_2$ fairly simulates $K_1$ iff $S$ is an init-fair simulation from $K_1$ to $K_2$.

Proof:

Due to init-sensitivity (Proposition 4.7), we know $K_2$ fairly simulates $K_1$ iff $K_2$ init-fairly simulates $K_1$.

Suppose $S$ is an init-fair simulation from $K_1$ to $K_2$. Since $S$ has to be a simulation from $K_1$ to $K_2$ as well, and $\hat{S}$ is the maximal simulation, we have $S \subseteq \hat{S}$. Further, due to monotonicity (Proposition 4.5), $\hat{S}$ has to be an init-fair simulation as well.

The maximal simulation from $K_1$ to $K_2$ can be constructed in time $O(|R_1| \cdot |W_2| + |R_2| \cdot |W_1|)$ [BP95, HHK95]. Hence, it is left to find an algorithm that efficiently checks, given a relation $S \subseteq W_1 \times W_2$, if $S$ is an init-fair simulation from $K_1$ to $K_2$. For this purpose, consider the structure $K_S = \langle \Gamma_S, W, \hat{w}, R, L \rangle$, with the following components:

- $\Gamma_S = W_1 \cup W_2$. Thus, each state of $K_S$ is labeled by a state of $K_1$ or $K_2$.
- $W = \langle S \times \{a\} \rangle \cup \langle W_1 \times W_2 \times \{p\} \rangle$. Thus, there are two types of states: adversary-states, in which the $W_1$-component is related by $S$ to the $W_2$-component, and protagonist-states, which are not restricted. We regard the states of $K_S$ as positions in a game, with the adversary moving in adversary-states and the protagonist moving in protagonist-states.
- $\hat{w} = \langle \hat{w}_1, \hat{w}_2, a \rangle$. This is the initial game position.
• $R = \{ (\langle w_1, w_2, a \rangle, \langle w_1', w_2', p \rangle) | R_1(w_1, w_1') \} \cup \{ (\langle w_1, w_2, p \rangle, \langle w_1', w_2', a \rangle) | R_2(w_2, w_2') \}$. Thus, the adversary and the protagonist alternate moves. The adversary moves along transitions that correspond to transitions of $K_1$ and the protagonist moves along transitions that correspond to transitions of $K_2$. Since adversary-states consist only of pairs in $S$, the protagonist must reply to each move of the adversary to a state $\langle w_1', w_2, p \rangle$ with a move to a state $\langle w_1', w_2', a \rangle$ for which $S(w_1', w_2')$.

• We label an adversary-state by its $W_1$-component and we label a protagonist-state by its $W_2$-component; that is, $L(\langle w_1, w_2, a \rangle) = \{ w_1 \}$ and $L(\langle w_1, w_2, p \rangle) = \{ w_2 \}$.

We say that a run $\bar{w}$ of $K_S$ satisfies a fairness constraint $F$ if $F(L(\bar{w})) = \text{fair}$. The protagonist wins the game on $K_S$ if (1) whenever the game position is a protagonist-state, the protagonist can proceed with a move, and (2) whenever the game produces an infinite run of $K_S$, either the run does not satisfy $F_1$ or it satisfies both $F_1$ and $F_2$. Then, the protagonist has a winning strategy in this game iff $S$ is an init-fair simulation from $K_1$ to $K_2$.

The problem of checking the existence of a winning strategy (and the synthesis of such a strategy [Rab70, PR89]) can be reduced to the nonemptiness problem for tree automata. We first define finite automata over words and trees formally.

**Word automata.** An automaton on (finite or infinite) words is a quintuple $\mathcal{A} = \langle \Gamma, Q, q, \delta, \alpha \rangle$, with the following components:

• A finite alphabet $\Gamma$.

• A finite set $Q$ of states.

• An initial state $q \in Q$.

• A transition function $\delta : Q \times \Gamma \rightarrow 2^Q$.

• An acceptance condition $\alpha$, which is a partial function that maps sequences from $Q^* \cup Q^\omega$ to the set $\{ \text{accepting}, \text{rejecting} \}$.

A run of $\mathcal{A}$ on an input word $\bar{\gamma} = \gamma_0 \cdot \gamma_1 \cdot \gamma_2 \cdots$ is a word $\bar{q} = q_0 \cdot q_1 \cdot q_2 \cdots$, such that $q_0 = q$ and $q_{i+1} \in \delta(q_i, \gamma_i)$. A run $\bar{q}$ is accepting iff $\alpha(\bar{q}) = \text{accepting}$. The automaton accepts $\bar{\gamma}$ iff there exists an accepting run of $\mathcal{A}$ on $\bar{\gamma}$. The set of all words accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$, the language of $\mathcal{A}$. 


CHAPTER 4. FAIR BRANCING REFERENCE

Tree automata. An automaton on (finite or infinite) trees is a hextuple \( A = (\Gamma, Q, \tilde{q}, \mathcal{D}, \delta, \alpha) \), where \( \Gamma, Q, \tilde{q} \), and \( \alpha \) are as in word automata, and in addition, we have:

- A finite set \( \mathcal{D} \subset \mathbb{N} \) of possible branching degrees.
- A transition function \( \delta : Q \times \Gamma \times \mathcal{D} \rightarrow 2^{Q^*} \), such that for every \( q \in Q \), \( \gamma \in \Gamma \), and \( k \in \mathcal{D} \), we have that \( \delta(q, \gamma, k) \subseteq 2^{Q^k} \). Thus, both \( \gamma \) and \( k \) are arguments of the transition function.

A run of \( A \) on an input \( \Gamma \)-labeled tree \( \langle t, \lambda \rangle \) is a \( Q \)-labeled tree \( \langle t, \theta \rangle \) such that \( \theta(\epsilon) = \hat{q} \) and for every \( x \in t \), we have that

\[
\langle \theta(x \cdot 0), \theta(x \cdot 1), \ldots, \theta(x \cdot (\text{deg}(x) - 1)) \rangle \in \delta(\theta(x), \lambda(x), \text{deg}(x))
\]

If, for instance, \( \theta(0) = q_0 \), \( \lambda(0) = \gamma \), \( \text{deg}(0) = 2 \), and \( \delta(q_0, \sigma, 2) = \{ \langle q_1, q_2 \rangle, \langle q_4, q_5 \rangle \} \), then either \( \theta(0 \cdot 0) = q_1 \) and \( \theta(0 \cdot 1) = q_2 \), or \( \theta(0 \cdot 0) = q_4 \) and \( \theta(0 \cdot 1) = q_5 \). A run \( \langle t, \theta \rangle \) is accepting iff \( \alpha(\theta(\rho)) = \text{accepting} \) for all paths \( \rho \subseteq t \). An automaton \( A \) accepts \( \langle t, \lambda \rangle \) iff there exists an accepting run of \( A \) on \( \langle t, \lambda \rangle \). The set of all trees accepted by \( A \) is denoted by \( L(A) \), the language of \( A \).

We now return to the problem of deciding whether the protagonist has a winning strategy in the game played in \( K_S \). We construct two tree automata:

1. The tree automaton \( A_S \) accepts all infinite \((W_1 \cup W_2)\)-labeled trees that can be obtained by unrolling \( K_S \) and pruning from it subtrees that have as a root an adversary-state so that each protagonist state has exactly one successor. The intuition is that each tree accepted by \( A_S \) corresponds to a strategy of the protagonist.

The automaton \( A_S \) has \(|W_1| \cdot |W_2| \) states, and it has a vacuous acceptance condition. Formally, \( A_S = (\Gamma_s, W, \tilde{w}, \mathcal{D}, \delta, \alpha_s) \), where \( A_S \) has the same input alphabet as the observation alphabet of \( K_S \), same states and initial state as \( K_S \). The set of possible branching degrees is given by

\[ \mathcal{D} = \{ k \mid k \text{ is the outdegree of an adversary state} \} \cup \{ 1 \} \]

The transition function \( \delta \) is defined by:

- \( \delta(\langle w_1, w_2, a \rangle, w_1, k) = \{ \langle \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \ldots, \mathbf{p}_k \rangle \} \), where \( k \) is the out-degree of \( \langle w_1, w_2, a \rangle \) and \( R(\langle w_1, w_2, a \rangle, \mathbf{p}_i) \) for \( i = 1, 2, \ldots, k \).
CHAPTER 4. FAIR BRANCHING REFINEMENT

59

- \( \delta\left( \langle w_1, w_2, p \rangle, w_2, 1 \right) = \{ \langle a_1 \rangle, \langle a_2 \rangle, \langle a_3 \rangle, \ldots, \langle a_k \rangle \} \), where \( k \) is the out-degree of \( \langle w_1, w_2, p \rangle \) and \( R(\langle w_1, w_2, p \rangle, a_i) \) for \( i = 1, 2, \ldots, k \).

The acceptance condition \( \alpha_s \) maps all runs of \( \mathcal{A} \) to accepting.

2. The tree automaton \( \mathcal{A}_F \) accepts all infinite trees labeled by \( W_1 \cup W_2 \) in which all paths that satisfy \( F_1 \) satisfy \( F_2 \) as well. In order to define \( \mathcal{A}_F \), we first construct a deterministic word automaton \( \mathcal{A}_W \). The automaton \( \mathcal{A}_W \) accepts words over \( W_1 \cup W_2 \) satisfying the LTL formula \( \psi = (\phi_1 \rightarrow \phi_2) \), where \( \phi_1 \) and \( \phi_2 \) are LTL formulas corresponding to \( F_1 \) and \( F_2 \) respectively. We use usual notation to describe LTL formulas. For example, a run \( \overline{w} \) satisfies \( \square \Diamond p \) if \( p \) is true in infinitely many states of \( \overline{w} \), and it satisfies \( \Diamond \square p \) if \( p \) is true in all states of some infinite suffix of \( \overline{w} \). The definition of \( \mathcal{A}_W \) depends on the type of the fairness conditions of \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \):

- When \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \) are Büchi structures, \( \psi \) has the form \( \square \Diamond l \rightarrow \square \Diamond r \), where \( l = F_1 \subseteq W_1 \) and \( r = F_2 \subseteq W_2 \). Then,

\[
\mathcal{A}_W = (\Gamma_S, \{q_0, q_l, q_r\}, q_0, \delta, \{(q_l, q_r)\}),
\]

where for every \( \varsigma \in \Gamma_S \), we have:

\[
\delta(q_0, \varsigma) = \delta(q_l, \varsigma) = \delta(q_r, \varsigma) = \begin{cases} 
q_l & \text{If } \varsigma \in l. \\
q_r & \text{If } \varsigma \in r. \\
q_0 & \text{Otherwise.} 
\end{cases}
\]

Note that the acceptance condition for \( \mathcal{A}_W \) is a Streett condition \( \{(q_l, q_r)\} \), which states that a run is accepting if it either visits \( q_l \) only finitely many times, or visits \( q_r \) infinitely many times.

- When \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \) are Streett structures with

\[
F_1 = \{ \langle t_1^1, r_1^1 \rangle, \langle t_2^1, r_2^1 \rangle, \ldots, \langle t_{|F_1|}^1, r_{|F_1|}^1 \rangle \}, \text{ and}
\]

\[
F_2 = \{ \langle t_1^2, r_1^2 \rangle, \langle t_2^2, r_2^2 \rangle, \ldots, \langle t_{|F_2|}^2, r_{|F_2|}^2 \rangle \},
\]

the formula \( \psi \) has the form

\[
\bigwedge_{i=1}^{|F_1|} (\square \Diamond t_i^1 \rightarrow \square \Diamond r_i^1) \rightarrow \bigwedge_{j=1}^{|F_2|} (\square \Diamond t_j^2 \rightarrow \square \Diamond r_j^2).
\]
We can rewrite $\psi$ to have $|F_1| + 1$ disjuncts as:

$$
\left( \bigvee_{i=1}^{|F_1|} \neg (\Box \Diamond l_i^1 \rightarrow \Box \Diamond r_i^1) \right) \lor \bigwedge_{j=1}^{|F_2|} (\Box \Diamond l_j^2 \rightarrow \Box \Diamond r_j^2).
$$

With each of the first $|F_1|$ disjuncts $\neg (\Box \Diamond l_i^1 \rightarrow \Box \Diamond r_i^1)$, we can associate an automaton

$$
\mathcal{A}_i = \langle \Gamma, \{q_0, q_l, q_r\}, q_0, \delta, \{\{q_0, q_l, q_r\}, \{q_i\}, \{q_r\}, \{\}\}\rangle,
$$

where for every $s \in \Gamma_S$, we have:

$$
\delta(q_0, s) = \delta(q_l, s) = \delta(q_r, s) = \begin{cases} 
q_l & \text{if } s \in l_i^1, \\
q_r & \text{if } s \in r_i^1, \\
q_0 & \text{Otherwise.}
\end{cases}
$$

Note that the acceptance condition for $\mathcal{A}_i$ is a Streett condition with two Streett pairs. Consider any accepting run of $\mathcal{A}_i$. The first Streett pair $\langle\{q_0, q_l, q_r\}, \{q_i\}\rangle$ ensures that the run visits $q_l$ infinitely many times. The second Streett pair $\langle\{q_r\}, \{\}\rangle$ ensures that the run visits $q_r$ only finitely many times.

For the last disjunct we can construct an automaton $\mathcal{A}(|F_1| + 1)$ with $|F_2|$ states and $|F_2|$ Streett pairs. We can then take the union of the $|F_1| + 1$ automata and obtain a deterministic Streett automaton with $3|F_1| \times |F_2|$ states and $(2\cdot|F_1| + |F_2|)$ Streett pairs.

Since the word automaton $\mathcal{A}_W$ is deterministic, it can be transformed into a tree automaton $\mathcal{A}_F$, with branching degrees $\mathcal{D}$. Formally, if $\mathcal{A}_W = \langle \Gamma, Q, \hat{q}, \delta, \alpha \rangle$, then $\mathcal{A}_F = \langle \Gamma, Q, \hat{q}, \mathcal{D}, \delta', \alpha \rangle$, where $\delta'(q, s, k) = \langle q', q', ..., q' \rangle$ whenever $k \in \mathcal{D}$, and $\delta(q, s) = q'$.

Since $\mathcal{A}_W$ is deterministic, a tree $\langle t, \lambda \rangle$ is in $\mathcal{L}(\mathcal{A}_F)$ iff all branching degrees of $t$ are in $\mathcal{D}$, and for every path $\rho$ in $\langle t, \lambda \rangle$, we have $\lambda(\rho) \in \mathcal{L}(\mathcal{A}_W)$ [KSV96].

Note that $\mathcal{A}_F$ has the same number of states and Streett pairs as $\mathcal{A}_W$.

It is easy to see that the protagonist has a winning strategy iff the intersection of the Streett automata $\mathcal{A}_S$ and $\mathcal{A}_F$ is nonempty. To check the latter, we define and check the nonemptiness of the product automaton $\mathcal{A}_S \times \mathcal{A}_F$. Since $\mathcal{A}_S$ has a vacuous acceptance condition, the product automaton is a Streett automaton with the same number of pairs as $\mathcal{A}_F$. Finally, since checking the nonemptiness of a Streett tree automaton with $n$ states and $f$ pairs requires time $O(n^{(2f + 1)} \cdot f!)$ [KV98], the theorem below follows.
Theorem 4.1  Given two fair structures $\mathcal{K}_1$ and $\mathcal{K}_2$ with state sets $W_1$ and $W_2$, transition relations $R_1$ and $R_2$, and fairness constraints $F_1$ and $F_2$, we can check whether $\mathcal{K}_2$ fairly simulates $\mathcal{K}_1$ in time:

- $O(|R_1| \cdot |W_2| + |R_2| \cdot |W_1| + (|W_1| \cdot |W_2|)^3)$, for Büchi structures.
- $O(n^{(2f+1)} \cdot f!)$, where $n = |W_1| \cdot |W_2| \cdot (3|F_1| + |F_2|)$, and $f = 2 \times |F_1| + |F_2|$, for Street structures.

4.5 A logical characterization of fair simulation

We show that fair simulation characterizes the distinguishing power of the fair universal fragment of the alternation free $\mu$-calculus (Fair-$\forall$AFMC); that is, for every two fair structures $\mathcal{K}_1$ and $\mathcal{K}_2$, every Fair-$\forall$AFMC formula that is satisfied in $\mathcal{K}_2$ is satisfied also in $\mathcal{K}_1$ iff $\mathcal{K}_2$ fairly simulates $\mathcal{K}_1$. For technical convenience, we consider $\exists$AFMC, the dual, existential fragment of the alternation-free $\mu$-calculus.

Syntax and semantics of Fair-$\exists$AFMC

The syntax of $\exists$AFMC is defined with respect to a set $P$ of propositions and a set $V$ of variables. We first define the syntax of the entire existential $\mu$-calculus ($\exists$-MC.) A formula of $\exists$-MC is one of the following:

- $\text{true}$, $\text{false}$, $p$, or $\neg p$, for $p \in P$.
- $y$, for $y \in V$.
- $\varphi_1 \lor \varphi_2$ or $\varphi_1 \land \varphi_2$, where $\varphi_1$ and $\varphi_2$ are $\exists$-MC formulas.
- $\exists \varphi$, where $\varphi$ is an $\exists$-MC formula.
- $\mu y. f(y)$ or $\nu y. f(y)$, where $f(y)$ is a $\exists$-MC formula. All occurrences of the variable $y$ in $\mu y. f(y)$ and $\nu y. f(y)$ are bound.

A $\exists$-MC formula is alternation free if for all $y \in V$, there are respectively no occurrences of $\nu$ ($\mu$) in any syntactic path from an occurrence of $\mu y$ ($\nu y$) to an occurrence of $y$. For example, the formula $\mu x. (p \lor \mu y. (x \lor \exists \varphi))$ is alternation free, and the formula
$\mu x. (p \lor \nu y. (x \lor \exists y \psi))$ is not alternation free. The existential alternation free $\mu$-calculus ($\exists$AFMC) is the set of all $\exists$-MC formulas that are alternation free.

The semantics of $\exists$AFMC is defined for formulas without free occurrences of variables. We interpret $\exists$AFMC formulas over fair structures, thus obtaining the logic Fair-$\exists$AFMC. While there is no obvious interpretation of the full alternation-free $\mu$-calculus over fair structures, $\exists$AFMC does not admit any switching between universal and existential path quantifiers within a fixed-point calculation. This enables us to limit all fixed-point calculations to fair paths. Before we give the formal semantics of Fair-$\exists$AFMC, let us note that since AFMC is a “local” logic, in the sense that its only temporal operator is next, and fairness is a global property, any attempt to add fairness to AFMC would result in something that may seem unsatisfactory. We believe that our definition is the most natural semantics one can get by adding fairness to AFMC. In particular, as shall now be below, when we restrict attention to fair-$\exists$AFMC formulas that have equivalences of CTL, our semantics of fair-$\exists$AFMC coincides with the one for fair-$\exists$CTL.

The closure $cl(\psi)$ of a Fair-$\exists$AFMC formula $\psi$ is the least set of formulas that satisfies the following conditions:

- $true \in cl(\psi)$ and $false \in cl(\psi)$.
- $\psi \in cl(\psi)$.
- If $\varphi_1 \land \varphi_2$ or $\varphi_1 \lor \varphi_2$ is in $cl(\psi)$, then $\varphi_1 \in cl(\psi)$ and $\varphi_2 \in cl(\psi)$.
- If $\exists \varphi \in cl(\psi)$, then $\varphi \in cl(\psi)$.
- If $\mu y. f(y) \in cl(\psi)$, then $f(\mu y. f(y)) \in cl(\psi)$.
- If $\nu y. f(y) \in cl(\psi)$, then $f(\nu y. f(y)) \in cl(\psi)$.

Each Fair-$\exists$AFMC formula $\psi$ specifies a set of “obligations” — a subset of formulas in $cl(\psi)$ — that need to be satisfied. The witness to the satisfaction of a formula is a tree called a sat-tree. Formally, given a fair structure $K = \langle K, F \rangle$ with $K = \langle T, W, w, R, L \rangle$, and a Fair-$\exists$AFMC formula $\psi$, a sat-tree $\langle t, \lambda \rangle$ of $K$ for $\psi$ is a $(W \times cl(\psi))$-labeled tree $\langle t, \lambda \rangle$ that satisfies the following conditions:

- $\lambda(e) = \langle \hat{w}, \psi \rangle$. Thus, the root of the tree, which corresponds to the initial obligation, is labeled by the initial state of $K$ and $\psi$ itself.
CHAPTER 4. FAIR BRANCHING REFINEMENT

- If $\lambda(x) = \langle w, \text{false} \rangle$, then $\text{deg}(x) = 0$.

- If $\lambda(x) = \langle w, \text{true} \rangle$ and $w$ has no successors in $K$, then $\text{deg}(x) = 0$.

- If $\lambda(x) = \langle w, \text{true} \rangle$ and $w$ has successors in $K$, then $\text{deg}(x) = 1$ and $\lambda(x_0) \in \{\langle w', \text{true} \rangle \mid R(w, w') \}$.

- If $\lambda(x) = \langle w, p \rangle$, where $p \in P$, then $\text{deg}(x) = 1$. If $p \in L(w)$, then $\lambda(x_0) = \langle w, \text{true} \rangle$, otherwise $\lambda(x_0) = \langle w, \text{false} \rangle$.

- If $\lambda(x) = \langle w, \neg p \rangle$, where $p \in P$, then $\text{deg}(x) = 1$. If $p \in L(w)$, then $\lambda(x_0) = \langle w, \text{false} \rangle$, otherwise $\lambda(x_0) = \langle w, \text{true} \rangle$.

- If $\lambda(x) = \langle w, \varphi_1 \lor \varphi_2 \rangle$, then $\text{deg}(x) = 1$ and $\lambda(x_0) \in \{\langle w_1, \varphi_1 \rangle, \langle w_2, \varphi_2 \rangle \}$.

- If $\lambda(x) = \langle w, \varphi_1 \land \varphi_2 \rangle$, then $\text{deg}(x) = 2$, $\lambda(x_0) = \langle w, \varphi_1 \rangle$, and $\lambda(x_1) = \langle w, \varphi_2 \rangle$.

- If $\lambda(x) = \langle w, \exists \top \varphi \rangle$, then $\text{deg}(x) = 1$ and $\lambda(x_0) \in \{\langle w', \varphi \rangle \mid R(w, w') \}$.

- If $\lambda(x) = \langle w, \forall y. f(y) \rangle$, then $\text{deg}(x) = 1$ and $\lambda(x_0) = \langle w, f(\forall y. f(y)) \rangle$.

- If $\lambda(x) = \langle w, \mu y. f(y) \rangle$, then $\text{deg}(x) = 1$ and $\lambda(x_0) = \langle w, f(\mu y. f(y)) \rangle$.

Consider, for example, the fair structure $I$ from Figure 4.2 and the Fair-$\exists$AFMC formula

$$\varphi = \nu z. (a \land \exists z (b \land \exists z (c \land \exists z z) \land \exists z (d \land \exists z z)))$$

Intuitively, a state of $I$ belongs to the set defined by the variable $z$ if $z$ is the largest set that satisfies the above property. In addition, as $\varphi$ is a Fair-$\exists$AFMC formula, it is required that all runs of $I$ that are embedded in $z$ are fair. A sat-tree of $I$ for $\varphi$ is presented in Figure 4.5 (in the figure, we use the following abbreviations for formulas in $cl(\varphi)$: $\varphi_4 = d \land \exists z \varphi$; $\varphi_3 = c \land \exists z \varphi$; $\varphi_2 = b \land \exists z \varphi_3 \land \exists z \varphi_4$; and $\varphi_1 = a \land \exists z \varphi_2$).

Consider a sat-tree $\langle t, \lambda \rangle$ of $K$ for $\psi$. If $\langle t, \lambda \rangle$ contains no node labeled $\langle w, \text{false} \rangle$, then it provides a witness to the satisfaction of all local obligations induced by $\psi$. In addition, we have to make sure that least fixed-point obligations are not propagated forever, and that existential obligations are satisfied along fair runs of $K$. Formally, the sat-tree $\langle t, \lambda \rangle$ of $K$ for $\psi$ is convincing if the following three conditions hold:
1. The sat-tree $\langle t, \lambda \rangle$ contains no node labeled $\langle w, \text{false} \rangle$. Thus, all local obligations induced by $\psi$ are satisfied.

2. For all infinite paths $\rho$ of $\langle t, \lambda \rangle$, the projection of $\lambda(\rho)$ on the $\text{cl}(\psi)$-component contains only finitely many occurrences of formulas of the form $\mu y.f(y)$. Thus, no least fixed-point obligations are propagated forever.

3. For all infinite paths $\rho$ of $\langle t, \lambda \rangle$, the projection of $\lambda(\rho)$ on the $W$-component satisfies the fairness constraint $F$ of $K$. Thus, all existential obligations are satisfied along fair runs.

Then, the fair structure $K$ satisfies the Fair-$\exists$AFMC formula $\psi$, written $K \models \psi$, if there exists a convincing sat-tree of $K$ for $\psi$. For example, the sat-tree from Figure 4.5 is convincing. This is because (1) it contains no nodes labeled $\langle w, \text{false} \rangle$, (2) it contains no least fixed-point obligations, and (3) for all infinite paths, the projection of the labels to the states of $I$ satisfies the fairness constraint of $I$. Hence, $I \models \varphi$.

Our definition of Fair-$\exists$AFMC is very similar to the automata-theoretic characterization of the alternation-free $\mu$-calculus. Indeed, a convincing sat-tree of $K$ for $\psi$ can be
viewed as a run of an alternating tree automaton for $\psi$ on $\mathcal{K}$ [BVW94]. We also note that for the Fair-$\exists$AFMC formulas that correspond to the existential fragment of Fair-CTL, our definition coincides with the usual semantics for Fair-CTL [CES86].

Fair simulation and Fair-$\exists$AFMC

Before we show that fair simulation and Fair-$\exists$AFMC induce the same relation on fair structures, we demonstrate that this is not the case for $\exists$-simulation. Consider again the fair structures from Figure 4.2. We saw that the Fair-$\exists$AFMC formula $\varphi$ is satisfied in $\mathcal{I}$. On the other hand, it is easy to check that although $\mathcal{S}$ $\exists$-simulates $\mathcal{I}$, the formula $\varphi$ is not satisfied in $\mathcal{S}$.

Theorem 4.2 For all fair structures $\mathcal{K}_1$ and $\mathcal{K}_2$, the following are equivalent:

1. $\mathcal{K}_2$ fairly simulates $\mathcal{K}_1$.

2. For every formula $\psi$ of Fair-$\exists$AFMC, if $\mathcal{K}_1 \models \psi$ then $\mathcal{K}_2 \models \psi$.

Proof: Assume first that $\mathcal{K}_2$ fairly simulates $\mathcal{K}_1$ and $\mathcal{K}_1 \models \psi$. Then, there exists a convincing sat-tree $\langle t, \lambda \rangle$ of $\mathcal{K}_1$ for $\psi$. Let $\langle t, \lambda' \rangle$ be the fair trace-tree of $\mathcal{K}_1$ induced by $\langle t, \lambda \rangle$. By Proposition 4.4, this trace-tree is also a fair trace-tree of $\mathcal{K}_2$. Thus, there exists a fair run-tree of $\mathcal{K}_2$ that witnesses $\langle t, \lambda' \rangle$ and which can be used to construct a convincing sat-tree of $\mathcal{K}_2$ for $\psi$. It follows that $\mathcal{K}_2 \models \psi$.

Assume now that $\mathcal{K}_2$ does not fairly simulate $\mathcal{K}_1$. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be tree automata that accept all fair trace-trees of $\mathcal{K}_1$ and $\mathcal{K}_2$, respectively. By Proposition 4.4, the language of $\mathcal{A}_1$ is not contained in the language of $\mathcal{A}_2$. Hence, by [Rab70], there is a regular tree (i.e., a tree with only finitely many distinct subtrees) that is accepted by $\mathcal{A}_1$ and not accepted by $\mathcal{A}_2$. This tree can be encoded by a Fair-$\exists$AFMC formula $\psi$ such that $\mathcal{K}_1 \models \psi$ and $\mathcal{K}_2 \not\models \psi$.

4.6 Fair similarity quotients

Suppose we are given a fair structure $\mathcal{K}$. We are interested in finding a fair structure $\mathcal{K}'$ that fairly simulates $\mathcal{K}$, and is smaller than $\mathcal{K}$. There are several advantages in finding such a $\mathcal{K}'$. First, due to Theorem 4.2, for every Fair-$\exists$AFMC formula $\psi$ (and in particular, for every Fair-$\forall$CTL formula), if $\mathcal{K}' \models \psi$, then $\mathcal{K} \models \psi$ as well. Thus $\mathcal{K}'$ is a property
preserving abstraction of $\mathcal{K}$, for properties expressed in Fair-$\forall$AFMC. Second, suppose $\mathcal{K}$ is one component in a large system. We can replace $\mathcal{K}$ by its abstraction $\mathcal{K}'$. By repeating this for each component in the system, we can obtain significant computational savings while checking for satisfaction of Fair-$\forall$AFMC formulas.

Let $K = \langle T, W, \hat{w}, R, L \rangle$ be a structure, and let $\mathcal{K} = \langle K, F \rangle$ be a fair structure with a Büchi fairness constraint $F$. Note that $\mathcal{K}$ fairly simulates itself. Let $S \subseteq W \times W$ be a maximal fair simulation of $\mathcal{K}$ by $\mathcal{K}$ i.e., no superset of $S$ is a fair simulation. Define a binary relation $E \subseteq W \times W$ as follows: $E(w, w')$ iff $S(w, w')$ and $S(w', w)$. It is easy to see that $E$ is reflexive, symmetric, and transitive. Thus, $E$ is an equivalence relation. The structure $K' = \langle T, W', \hat{w}', R', L' \rangle$ is defined as follows:

- The state set $W' = W/E$, the equivalence classes of $W$ with respect to $E$. We denote the equivalence class of state $w \in W$ by $[w]$.
- The initial state $\hat{w}' = [\hat{w}]$.
- The transition relation $R' = \{([w], [w']) \mid R(w, w')\}$.
- The labeling function $L'$ is given by $L'([w]) = L(w)$. Note that $L'$ is well defined.

The fair structure $\mathcal{K}' = \langle K', F' \rangle$, where $F' = \{[w]|w] \cap F \neq \{\}\}$, is called the fair-similarity abstraction of $\mathcal{K}$.

**Proposition 4.11** For any fair Büchi structure $\mathcal{K}$, and its fair-similarity abstraction $\mathcal{K}'$, we have that $\mathcal{K}'$ fairly simulates $\mathcal{K}$.

**Proof:** Consider the relation $S \subseteq W \times W'$, given by $S = \{(w, [w]) \mid w \in W\}$. Let $\tau$ be a strategy that maps any sequence $u_0 \cdot [u_0] \cdot w_1 \cdot [w_1] \cdots s$, where $u_0 \cdot w_1 \cdots s$ is a run of $K$, to $[s]$. Then, $S$ is a fair simulation from $\mathcal{K}$ to $\mathcal{K}'$, with $\tau$ as the witnessing strategy.

**4.7 Assume-guarantee rule for fair simulation**

We would like to obtain a proof rule for fair simulation between modules built using parallel composition, along the lines of the proof rule in Chapter 3, Section 3.5.

We first need to be able to add fairness constraints at the module level. A fair module is a pair $\mathcal{P} = \langle P, F^P \rangle$, where $P$ is a reactive module, and $F^P$ is a fairness condition.
CHAPTER 4. FAIR BRANCHING REFINEMENT

The fairness condition $F^P$ for $\mathcal{P}$ is a function that maps every infinite run of $P$ to the binary set $\{\text{fair}, \text{unfair}\}$. For the purposes of this section, we assume that $F^P$ is either a Büchi or Streett fairness condition. A higher level specification of fairness using weakly fair and strongly fair actions can be found in [AH96].

A fair run of $\mathcal{P}$ is either a finite run of $P$ or an infinite run $\bar{s}$ of $P$ such that $F^P(\bar{s}) = \text{fair}$. A fair trace of $\mathcal{P}$ is a trace of $P$ that is witnessed by a fair run of $\mathcal{P}$. A fair run-tree of $\mathcal{P}$ is a run-tree $\langle \tau, \lambda \rangle$ of $P$ such that for every labelled path $\rho$ of $\tau$ is a fair run of $\mathcal{P}$. A fair trace-tree of $\mathcal{P}$ is a trace-tree of $P$ that is witnessed by a fair run-tree of $\mathcal{P}$.

A fair module $\mathcal{P} = \langle P, F^P \rangle$ has an associated fair Kripke structure $\mathcal{K}_\mathcal{P} = \langle K, F^\mathcal{P} \rangle$. Note that $\mathcal{P}$ and $\mathcal{K}_\mathcal{P}$ are semantically equivalent (i.e., they have the same set of fair traces and fair trace-trees.)

Fair simulation between modules. Let $\mathcal{P} = \langle P, F^P \rangle$ and $\mathcal{Q} = \langle Q, F^Q \rangle$ be two fair modules, such that $Q$ is refinable by $P$. Then we say that $\mathcal{P} \preceq_B \mathcal{Q}$ if for every fair trace-tree $T$ of $\mathcal{P}$, the projection $[T]_Q$ is a fair trace-tree of $\mathcal{Q}$. We can also give equivalent definitions of $\preceq_B$ based on fair simulation relation, game theory, and ∀AFMC (as in Section 4.2.)

Since all finite runs are fair by definition, we have the following proposition.

**Proposition 4.12** Consider two fair modules $\mathcal{P} = \langle P, F^P \rangle$ and $\mathcal{Q} = \langle Q, F^Q \rangle$. If $\mathcal{P} \preceq_B \mathcal{Q}$, then $P \preceq_B Q$.

Fair composition. Let $\mathcal{P} = \langle P, F^P \rangle$ and $\mathcal{Q} = \langle Q, F^Q \rangle$ be two fair modules. The composition of $\mathcal{P}$ and $\mathcal{Q}$, denoted by $\mathcal{P} \parallel \mathcal{Q}$, exists if $P \parallel Q$ exists, and is defined to be the fair module $\mathcal{K} = \langle K, F^K \rangle$, where

- $K = P \parallel Q$,
- $F^K(\bar{s}) = \text{fair}$ iff both $F^P(\bar{s}_P) = \text{fair}$ and $F^Q(\bar{s}_Q) = \text{fair}$, where $\bar{s}_P$ is the projection of $\bar{s}$ on $P$ and $\bar{s}_Q$ is the projection of $\bar{s}$ on $Q$.

Intuitively, a run $\bar{s}$ of $\mathcal{K}$ is fair if (1) the projection of $\bar{s}$ on $P$ is fair with respect to $F^P$, and (2) the projection of $\bar{s}$ on $Q$ is fair with respect to $F^Q$.

**Proposition 4.13** Consider fair modules $\mathcal{P}$ and $\mathcal{Q}$ such that the composition $P \parallel Q$ exists. Then $\bar{s}$ is a fair trace of $\mathcal{P} \parallel \mathcal{Q}$ iff (1) $[\bar{s}]_P$ is a fair trace of $\mathcal{P}$, and (2) $[\bar{s}]_Q$ is a fair trace of $\mathcal{Q}$.
Proposition 4.14 Consider two fair modules \( \mathcal{P} \) and \( \mathcal{Q} \) such that the composition \( \mathcal{P} \parallel \mathcal{Q} \) exists. Then \( T \) is a fair trace-tree of \( \mathcal{P} \parallel \mathcal{Q} \) iff (1) \([T]_P\) is a fair trace-tree of \( \mathcal{P} \), and (2) \([T]_Q\) is a fair trace-tree of \( \mathcal{Q} \).

Let \( \text{Safe}(\mathcal{P}) \) be the fair module obtained by replacing the fairness constraint of \( \mathcal{P} \) with the trivial fairness constraint that maps every infinite run to \text{fair}. We now present the assume-guarantee proof rule for fair simulation. The same rule was proved for fair trace containment in [AH96], with an essentially similar proof (we just use trace-trees instead of traces to get the proof below.)

Theorem 4.3 Let \( \mathcal{P}, \mathcal{Q}, \mathcal{P}' \) and \( \mathcal{Q}' \) be fair modules such that \( \mathcal{P} \parallel \mathcal{Q} \) and \( \mathcal{P}' \parallel \mathcal{Q}' \) exist. Suppose that \( \mathcal{P} \parallel \text{Safe}(\mathcal{Q}') \preceq_B \mathcal{P}' \), \( \mathcal{P}' \parallel \mathcal{Q} \preceq_B \mathcal{Q}' \), and every input of \( \mathcal{P}' \parallel \mathcal{Q}' \) is either an input or output of \( \mathcal{P} \parallel \mathcal{Q} \). Then, \( \mathcal{P} \parallel \mathcal{Q} \preceq_F \mathcal{P}' \parallel \mathcal{Q}' \).

Proof: Let \( \mathcal{P} = \langle P, F^P \rangle, \mathcal{Q} = \langle Q, F^Q \rangle; \mathcal{P}' = \langle P', F^{P'} \rangle \), and \( \mathcal{Q}' = \langle Q', F^{Q'} \rangle \). From Proposition 4.12, we know that \( P \parallel Q' \preceq_B P' \) and \( P' \parallel Q \preceq_B Q' \). Consequently, by Theorem 3.1 we know that \( P \parallel Q \preceq_B P' \parallel Q' \). Therefore, for every trace-tree \( T \) of \( P' \parallel Q' \) we have that \([T]_P \parallel Q\) is a trace-tree of \( P \parallel Q \). Hence, if \( T \) is a trace-tree of \( P \parallel Q \), then \([T]_{P'} \parallel Q\) is a trace-tree of \( P' \parallel Q' \). It remains to be proved that if \( T \) is a fair trace-tree of \( P \parallel Q \), then \([T]_{P'} \parallel Q\) is a fair trace-tree of \( P' \parallel Q' \).

For notational simplicity, we omit explicit projections in the following. Suppose \( T \) is a fair trace-tree of \( P \parallel Q \). From Proposition 4.14, \( T \) is a fair trace-tree of both \( P \) and \( Q \). Further, we know that \( T \) is a trace-tree of \( P' \parallel Q' \). Thus, \( T \) is a fair trace-tree of \( \text{Safe}(Q') \). Again, by Proposition 4.14, \( T \) is a fair trace-tree of \( P \parallel \text{Safe}(Q') \). Since \( P \parallel \text{Safe}(Q') \preceq_F P' \), we conclude that \( T \) is a fair trace-tree of \( P' \). Therefore, from Proposition 4.14, since \( T \) is a fair trace-tree of both \( P' \) and \( Q' \), it is a fair trace-tree of \( P' \parallel Q' \). Since \( P' \parallel Q \preceq_F Q' \), \( T \) is a fair trace-tree of \( Q' \). Finally from Proposition 4.14, since \( T \) is a fair trace-tree of both \( P' \) and \( Q' \), it is a fair trace-tree of \( P' \parallel Q' \).

Note the asymmetry in the antecedent of Theorem 4.3. The first requirement \( \mathcal{P} \parallel \text{Safe}(Q') \preceq_F \mathcal{P}' \) cannot be weakened to \( \mathcal{P} \parallel Q' \preceq_B \mathcal{P}' \). The asymmetry is required to break the circularity.
Chapter 5

Refinement with Sampling

So far, whenever we do refinement checking, our specifications are forced to operate on the same time scale as the implementation. This may lead to unnatural specifications and inefficiencies in verification. In this chapter, we introduce a novel methodology for decomposing refinement proofs of temporally abstract specifications, which specify implementation requirements only at certain sampling instances in time. Our new assume-guarantee rule allows separate refinement maps for specifying functionality and timing. Support for sampling and the generalized assume-guarantee rule have been implemented in the model checker Mocha and successfully applied to verify the VGI multiprocessor dataflow chip with 6 million transistors.

5.1 Motivation for sampling

Specifications are typically less detailed than the implementation. For example, the specification of an adder might simply state that the output is the sum of the two inputs, whereas the implementation might be a gate-level adder circuit, which operates at the detail of individual bits. Nonetheless, common notions of correctness require specifications to operate in “lock-step” with the implementation: every possible computation step of the implementation must be matched by an admissible computation step of the specification. If the natural time scale of the specification is less detailed than that of the implementation—for example, if the gate-level adder requires several clock cycles to compute a sum—then the specification often is “slowed down” by stuttering, even for perfectly synchronous designs. A prominent example of this occurs in pipeline verification, where the Instruction
Set Architecture (ISA) specification usually is slowed down by introducing a nondeterministic stall signal to stretch its time scale to match that of the pipeline [HQR98, McM98]. Instead of slowing down the specification, we pursue the alternative of “speeding up” the implementation. For this purpose, we use an operator called Sample, which samples the behavior of the implementation at appropriately defined sampling instants.²

If $P$ is a system description, and $\phi$ is a condition on the variables of $P$, then \textit{Sample $P$ at $\phi$} describes the same system at a more abstract temporal level: a single transition of \textit{Sample $P$ at $\phi$} aggregates as many transitions of $P$ as are required to satisfy the condition $\phi$, and hides the intermediate steps. For example, if $P$ is a gate-level description of an ALU, and $\phi$ signals the completion of an arithmetic operation, then \textit{Sample $P$ at $\phi$} is an operation-level description of the ALU. Mathematically, the semantics of \textit{Sample $P$ at $\phi$} is defined as a two-level hierarchy of transition systems: each transition of the upper-level (e.g., operation-level) transition system abstracts an entire lower-level (e.g., gate-level) transition system. Then, by nesting \textit{Sample} operators we obtain multi-level hierarchies of transition systems. The structuring of a state space into a multi-level transition hierarchy makes possible the exhaustive exploration of very large state spaces. This is because after the traversal of a level-$n$ transition, the computed reach set for the corresponding level-$(n-1)$ transition system represents hidden intermediate steps and can be removed from memory.

Our motivation for sampling arose specifically from the attempt of verifying a 96-processor V(ideo) G(raphics) I(mage) chip designed by the Infopad project at the University of California, Berkeley [STUR98]. There, the specification consists of ISAs for the individual processors and FIFO buffers that abstract the point-to-point communication protocols, which interact subtly with the processor pipelines. Since the implementation contains levelsensitive latches and different parts of the circuit are active at high vs. low phases of the clock, sampling must be used to match the implementation time scale with the specification time scale. In this chapter, we examine techniques for exploring state spaces of systems constructed using the \textit{Sample} operator. We illustrate the computational advantages of sampling in state-space exploration. However, even with these advantages, the VGI is still far beyond the scope of exhaustive search. Hence, we needed to generalize a compositional verification methodology to accommodate the \textit{Sample} operator.

²The \textit{Sample} operator is similar, but not identical, to the \textit{Next} operator of Reactive Modules [AH96]: while \textit{Next} changes the time scale of a module and its environment, \textit{Sample} does not constrain the environment, which therefore may offer multiple inputs between sampling instances.
In Chapter 3, we discussed the use of witness modules to reduce a refinement problem to projection refinement. We also discussed scalable approaches to refinement checking using assume-guarantee reasoning, and the use of abstract constraints to bridge the "semantic gap" between the specification and the implementation. In this chapter, we generalize the assume-guarantee method to accommodate the sampling operator. If implementation and specification operate at the same time scale, witness modules generate values for hidden specification variables at each step. However, if a single macro-step of the specification corresponds to several micro-steps of the implementation, it is necessary to provide witness modules that operate at the micro-step level. The purpose of such a witness is to generate the correct value for the specification signal to be witnessed and to maintain that value until the next sampling instance. Dually, if implementation and specification operate at the same time scale, refinement constraints provide abstract definitions for implementation variables at each step. If one specification step corresponds to several implementation steps, then it no longer suffices for the refinement constraints to supply values for the implementation variables at the rate of the specification—at sampling instances—but additional constraints need to be provided between sampling instances. Providing different refinement constraints at the macro and micro levels enables a separation of concerns: while macro-level constraints (at sampling instances) tend to describe the functional behavior of an implementation variable, micro-level constraints (between sampling instances) tend to describe its timing behavior. This separation of functionality and timing is particularly natural for collections of synchronous blocks that communicate asynchronously.

We develop the theory and methodology to carry out assume-guarantee reasoning when specifications are abstract in both space (fewer variables/components) and time (fewer observation points.) The crux of the theory lies in the ability to distribute the Sample operator over the parallel composition of implementation components using micro-level refinement constraints. The resulting assume-guarantee proof rule produces refinement obligations both at the macro level and at the micro level, which are then discharged by our model checker Mocha [AHM+98]. While micro-level refinement obligations can be handled using traditional model checking, we had to enhance Mocha to discharge macro-level refinement obligations. We have used this methodology successfully in the verification of the VGI chip. We found several subtle bugs which were unknown to the designers.

Speeding up the implementation using Sample has significant computational advantages when compared to earlier approaches that slow down the specification using stut-
tering. The sampled implementation typically has a smaller state space than the original implementation. If the refinement checking uses explicit state enumeration, we can use a secondary stack to explore and discard the implementation states between two sampling instances. If we use symbolic search using binary decision diagrams (BDDs), the sampled state space can be typically represented using a smaller BDD [AHR98].

**Example.** We consider a design that computes the Greatest Common Divisor (GCD) of two numbers. We will start with a synchronous specification shown in Figure 5.1(a). Given two inputs \( a \) and \( b \), the module \( GCDSpec1 \) computes the GCD of \( a \) and \( b \) and places the result in the output \( r \). The boolean input \( validin \) asserts that the inputs are valid in the current round and the boolean output \( validout \) asserts that the output is valid in the current round. Module \( GCDSpec1 \) operates synchronously, with a delay of one round. If inputs \( a \) and \( b \) are given in the current round, then the output is available at \( r \) in the next round.

We refine our specification and add more spatial and temporal detail on how the GCD is computed. We use Euclid’s algorithm to compute the GCD:

\[
GCD (a, b)
\]

\{ Given positive non-zero integers \( a \) and \( b \), compute \( GCD(a, b) \) \}

1. \( \textbf{if} \ (a = b) \ \textbf{return} \ (a) \);  
2. \( \textbf{if} \ ((a = 1) \ \text{or} \ (b = 1)) \ \textbf{return} \ (1) \);  
3. \( \text{small} := \min(a, b) \);
Figure 5.2: GCD Implementation
(4) \( \text{big} := \max(a, b) \);
(5) \textbf{return} \(GCD(\text{small}, \text{big} - \text{small})\);

The resulting refinement \textit{GCDSpec2} (Figure 5.1(b)) has three modules: \textit{IntlS}, \textit{DoneS}, and \textit{CompS}. Given two numbers, the \textit{DoneS} module decides if the \textit{GCD} is computed trivially (if the numbers are equal, or one of the numbers is 1.) If so, it sends the result, otherwise, it resends the numbers in increasing order. Suppose \textit{small} and \textit{big} are sent by the \textit{DoneS} module. The \textit{CompS} module responds by sending \textit{small} and \textit{big} - \textit{small}.

The \textit{IntlS} module takes data inputs from both \textit{CompS} and the environment and feeds the data to the \textit{DoneS} module. Referring back to Euclid’s algorithm, the \textit{DoneS} module implements lines (1) through (4), the \textit{CompS} module computes the arguments for the recursive call in line (5), and the \textit{IntlS} module handles routing of data between the recursive calls. The modules \textit{IntlS}, \textit{DoneS}, and \textit{CompS} communicate with each other using point-to-point communication links. Valid bits (\textit{valid}1, \textit{valid}2 and \textit{valid}3) are used to validate the presence of meaningful data on these links. For instance, if \textit{IntlS} wants to send two numbers to \textit{DoneS}, it places the numbers in \textit{aout} and \textit{bout}, and sets \textit{valid}1 to \textit{true}. Each of these communications is assumed to complete in one round.

While \textit{GCDSpec1} requires only one round to compute the \textit{GCD}, the module \textit{GCDSpec2} requires multiple rounds depending on the data inputs. We add an additional variable \textit{inprogress} in module \textit{GCDSpec2} and set it to \textit{true} whenever a \textit{GCD} computation is in progress. Using this variable and the \textit{Sample} operator in Section 5.2, we will formally state how \textit{GCDSpec2} refines \textit{GCDSpec1}.

Our final level of refinement uses a single physical broadcast channel for communication between the modules. Time-division multiplexed access (TDMA) is used to share the channel. Communication in the channel is conducted in units called \textit{frames}. A frame is divided, in time, into several \textit{time-slots}. Each module is allocated one or more \textit{time-slots} to send data. There is a \textit{Beacon} module that signals the beginning of a frame. Each module has its local counter that is synchronized on the \textit{Beacon} module’s signal. Once the \textit{frame} starts, each module sends data in its allocated \textit{time-slots}. A valid bit sent on the channel indicates if the data being sent in the current \textit{time-slot} is valid. The allocation of \textit{time-slots} to modules is done statically at configuration time, and stays fixed thereafter. Thus, every module knows the identity of the sender in each \textit{time-slot}. Figure 5.2(a) shows the block
diagram of the implementation Impl. In our example, a frame is divided into 6 slots. The
figure also shows the allocation of time-slots within the frame to individual modules —the
first two time-slots are given to the Intf module, the next two to the Done module, and
the last two to the Comp module. The Intf, Done, and Comp modules are intended to
have the same functionality as the specification modules IntfS, DoneS, and CompS from
GCDSpec2. However, while the communication between modules in GCDSpec2 happens in
a single round through point-to-point links, the communication between modules in Impl is
through a shared channel, and takes several rounds. Let sync be a variable of the module
Beacon that is set to true whenever the Beacon module sends the synchronizing signal. We
will use sync and the Sample operator to relate Impl to GCDSpec2 in Section 5.2.

5.2 The Sample operator

Spatial and temporal hierarchies. As we seek to enhance the applicability of model
checking to complex designs, we are faced with the so-called state-explosion problem: the
size of the state space grows exponentially with the size of the system description, making
exhaustive state-space exploration infeasible. Consequently, to use the model checking
tools effectively, one needs to focus on a critical system component. Since the behavior
of an individual component typically depends on its interaction with other components, a
component cannot be analyzed in isolation; rather, for a meaningful analysis, all relevant
aspects of the surrounding system need to be identified. This process, called abstraction,
is usually performed in an informal, manual fashion, and requires considerable expertise.
Indeed, it is not uncommon that a successful verification or falsification run, using a few
seconds of CPU time, depends on months of manual labor for constructing abstractions
that are neither too coarse to invalidate the correctness requirements, nor too detailed to
exhaust the tool capacities.

One goal of our research is to systematize and, whenever possible, automate the
construction of useful abstractions. Our approach is to provide the designer, within the
system description language, with operators for writing mental design abstractions into
the system description. The judicious use of such operators is called design for verifica-
bility, because it simplifies—and in some cases, eliminates—the process of “rediscovering”
abstractions after the design is completed.

Our abstraction operators are motivated by the two main, orthogonal structuring
principles for designs: (1) spatial aggregation together with hiding of spatial details, and
(2) temporal aggregation together with hiding of temporal details. Type-(1) abstractions
enable the design of components at different levels of spatial granularity: an ALU can be
designed by aggregating gates, then used as an atomic block (after hiding internal gates and
wires) in the design of a processor. Type-(2) abstractions enable the design of components
at different levels of temporal granularity: an arithmetic operation can be designed by
aggregating bit operations, then used as an atomic step (after hiding intermediate results)
in the design of an instruction.

Spatial, type-(1) abstractions can be written into a system description using, for
aggregation, the parallel composition (Definition 2.5) of subsystems and, for hiding [AH96],
the existential quantification of variables. Appropriate operators are provided by most
system description languages. They are exploited heavily, both to facilitate the description
of complex systems, and to obtain heuristics for coping with state explosion. For instance,
in symbolic state-space exploration using BDDs, instead of building a single transition
relation for the entire system, one typically maintains a set of transition relations, one for
each component [TSL+90].

By contrast, most system description languages do not provide operators for defining temporal, type-(2) abstractions. The Sample operator is intended to serve this purpose.

Definition 5.1 [Sample operator] Let $A$ be a transition constraint and $\varphi$ be a predicate
on the primed and unprimed observable variables of $A$. Then $B = \text{(Sample } A \text{ at } \varphi)$ is
a transition constraint. The private and observable variables of $B$ are the private and
observable variables of $A$. The initial predicate of $B$ is equal to the initial predicate of $A$.
The update predicate of $B$ is true at the pair of states $(s, t)$ iff there is a sequence of states
$s_0, s_1, \ldots, s_n$, such that (1) $s = s_0$, (2) $t = s_n$, (3) $s_i \rightarrow_A s_{i+1}$ for $i = 0, 1, \ldots, n - 1$, and
(4) $\varphi(s_i, s_{i+1})$ is false for $i = 0, 1, \ldots, n - 2$ and true for $i = n - 1$.

Informally, $B$ updates from state $s$ to $t$ if there exists a sequence of rounds of $A$
starting at $s$ and ending at $t$, such that the final round satisfies $\varphi$, and none of the
intermediate rounds satisfy $\varphi$. Given a trajectory of $A$, we use the term sampling instants
to refer to the instants in the trajectory where $\varphi$ is true.

We note that the Sample operator differs from the Next operator of [AH96] in two ways:
(1) Sample operates on transition constraints, whereas Next operates on modules,
and (2) Sample does not constrain the environment between sampling instants, whereas
Next constrains the environment to not change between sampling instants. In the rest of this chapter, whenever we write \( B = (\text{Sample} \ A \ at \ \varphi) \), we assume that \( \varphi \) is a predicate on the primed and unprimed observable variables of \( A \).

**Example.** Recall the GCD computation example from Section 5.1. Module \( \text{GCDSpec1} \) is the high-level specification. The intermediate-level specification and implementation modules are composed as follows:

\[
\begin{align*}
\text{GCDSpec2} &= \text{Intf}S \parallel \text{Done}S \parallel \text{Comp}S \\
\text{Impl} &= \text{Beacon} \parallel \text{Channel} \parallel \text{Intf} \parallel \text{Done} \parallel \text{Comp}
\end{align*}
\]

We wish to relate the intermediate specification \( \text{GCDSpec2} \) to the high level specification \( \text{GCDSpec1} \). Recall that \( \text{GCDSpec1} \) computes GCD in one round, whereas \( \text{GCDSpec2} \) takes multiple rounds. Also recall that \text{inprogress} is a variable of \( \text{GCDSpec2} \) that is set to \text{true} when \( \text{GCDSpec2} \) is doing the GCD computation. If we sample the behaviors of \( \text{GCDSpec2} \) during the instances where \text{inprogress} is \text{false}, the sampling should conform to the behaviors allowed by \( \text{GCDSpec1} \). Using the \text{Sample} operator, we can express this requirement as:

\[(\text{Sample} \ \text{GCDSpec2} \ at \ (\neg \text{inprogress}')) \preceq_L \text{GCDSpec1}\]

We also wish to relate \( \text{GCDSpec2} \) to the final implementation \( \text{Impl} \). Recall that every communication in \( \text{GCDSpec2} \) happens in a single round, whereas every communication in \( \text{Impl} \) takes several rounds (as many rounds it takes to transmit a frame) to complete. Recall that the \text{Beacon} module has a variable \text{sync} that is set to \text{true} when it sends the synchronization signal. Though \( \text{Impl} \) and \( \text{GCDSpec2} \) operate at different time scales, if we consider any trace of \( \text{Impl} \) and sample only the instants where \text{sync} is \text{true}, the resulting subsequence should be a trace of \( \text{GCDSpec2} \). Using the \text{Sample} operator, we can express this requirement as:

\[(\text{Sample} \ \text{Impl} \ at \ \text{sync}') \preceq_L \text{GCDSpec2}\]

**Properties of Sample.** The \text{Sample} operator has several appealing properties. First, the refinement relation between two transition constraints is maintained by application of \text{Sample}, as given by the following proposition.

**Proposition 5.1** [Sampled Refinement] Consider two transition constraints \( A \) and \( B \), where \( A \preceq_L B \). If \( \varphi \) is a predicate on the observable variables of \( B \), then \( (\text{Sample} \ A \ at \ \varphi) \preceq_L (\text{Sample} \ B \ at \ \varphi) \).
CHAPTER 5. REFINEMENT WITH SAMPLING

Proof: Consider any trace $\gamma$ of Sample $A$ at $\varphi$, with $\sigma = s_0, s_1, \ldots, s_n$ as the witnessing run. Then there must exist run $\sigma = t_0, t_1, \ldots, t_m$ of $A$ such that $\sigma$ is equal to the subsequence of states in $\sigma$ obtained by including $t_i$ if $i = 0$ or $\varphi(t_{i-1}, t_i)$. Since $A \preceq_L B$, there must exist a run $\sigma' = t'_0, t'_1, \ldots, t'_m$ of $B$, such that $[\sigma]_B = [\sigma']_B$. Finally, corresponding to $\sigma'$, there exists a run $\sigma'$ of (Sample $B$ at $\varphi$) such that $[\gamma]_B = [\sigma]_B = [\sigma']_B$.

The converse of the above proposition does not hold. It is easy to create modules $A$ and $B$ such that $A \not\preceq_L B$, but (Sample $A$ at $\varphi$) $\preceq_L$ (Sample $B$ at $\varphi$). The next proposition asserts the distributivity of Sample with respect to the parallel-composition operator.

Proposition 5.2 [Distributivity of Sample] If $A$, $B$ are any two transition constraints and $\varphi$ is a predicate on the observable variables common to $A$ and $B$, then (Sample $(A \parallel B)$ at $\varphi$) $\preceq_L$ (Sample $A$ at $\varphi$) $\parallel$ (Sample $B$ at $\varphi$).

Proof: Let $C = (\text{Sample } (A \parallel B) \text{ at } \varphi)$, $D = (\text{Sample } (A) \text{ at } \varphi)$, and $E = (\text{Sample } (B) \text{ at } \varphi)$. By definition of Sample and $\parallel$ operators, every initial state of $C$ is an initial state of $D \parallel E$. Suppose $s \to_C t$. Then we know that there is a sequence of states $s_0, s_1, \ldots, s_n$, such that

1. $s = s_0$, 
2. $t = s_n$, 
3. $s_i \to_{A \parallel B} s_{i+1}$ for $i = 0, 1, \ldots, n-1$, and 
4. $\varphi(s_i, s_{i+1})$ is false for $i = 0, 1, \ldots, n-2$ and true for $i = n-1$. Since $s_i \to_{A \parallel B} s_{i+1}$ implies $s_i \to_A s_{i+1}$ and $s_i \to_B s_{i+1}$, we have $s \to_D t$ and $s \to_E t$ as well. Thus $s \to_{D \parallel E} t$.

In practice, the traces of (Sample $A$ at $\varphi$) $\parallel$ (Sample $B$ at $\varphi$) form a large superset of the traces of (Sample $(A \parallel B)$ at $\varphi$). It is desirable to constrain the observable variables of $A$ and $B$ while distributing the Sample operator over parallel composition. We can strengthen the above proposition in the presence of suitable transition constraints $T_A$ and $T_B$ on the observable variables of $A$ and $B$, respectively. The resulting proposition, given below, will be used in the next section to carry out assume-guarantee reasoning between different time scales.

Proposition 5.3 [Constrained distributivity of Sample] Consider transition constraints A, B, $T_A$ and $T_B$, such that every variable of $T_A$ is an observable variable of $A$, and every variable of $T_B$ is an observable variable of $B$. If $A \parallel B \preceq_L T_A \parallel T_B$ and $\varphi$ is a predicate on the observable variables common to $A$ and $B$, then (Sample $(A \parallel B)$ at $\varphi$) $\preceq_L$ (Sample $(A \parallel T_A)$ at $\varphi$) $\parallel$ (Sample $(B \parallel T_B)$ at $\varphi$).

Proof: The proof is similar in spirit to the proof of Proposition 5.2. Let $C = (\text{Sample } (A \parallel B) \text{ at } \varphi)$, $D = (\text{Sample } (A \parallel T_A) \text{ at } \varphi)$, and $E = (\text{Sample } (B \parallel T_B) \text{ at } \varphi)$. By definition
of \textit{Sample} and \parallel operators, every initial state of \( C \) is an initial state of \( D \parallel E \). Suppose \( s \rightarrow_C t \). Then we know that there is a sequence of states \( s_0, s_1, \ldots, s_n \), such that (1) \( s = s_0 \), (2) \( t = s_n \), (3) \( s_i \rightarrow_A \parallel_B s_{i+1} \) for \( i = 0, 1, \ldots, n-1 \), and (4) \( \varphi(s_i, s_{i+1}) \) is false for \( i = 0, 1, \ldots, n-2 \) and true for \( i = n-1 \). Since every variable of \( T_A \) is an observable variable of \( A \), and every variable of \( T_B \) is an observable variable of \( B \), and \( A \parallel_B \preceq_L T_A \parallel_T B \), we have \([s_i]_{T_A} \parallel T_B \rightarrow T_A \parallel T_B [s_{i+1}]_{T_A} \parallel T_B \) for \( i = 0, 1, \ldots, n-1 \). Since \( s_i \rightarrow_A \parallel_B s_{i+1} \) and \([s_i]_{T_A} \parallel T_B \rightarrow T_A \parallel T_B [s_{i+1}]_{T_A} \parallel T_B \) imply \( s_i \rightarrow_A T_A s_{i+1} \) and \( s_i \rightarrow_B \parallel T_B s_{i+1} \), we have \( s \rightarrow_D t \) and \( s \rightarrow_E t \) as well. Thus \( s \rightarrow_D \parallel_E t \). \hfill \Box

5.3 Symbolic exploration of hierarchical state spaces

The reason why the \textit{Sample} operator can be exploited by model checking can be seen as follows. If the semantics of a transition constraint is viewed as a state-transition graph, with each round corresponding to a transition, then the sampled constraint \( B = \text{Sample } A \text{ at } \varphi \) may have many fewer states than the flat constraint \( A \). In particular, the \textit{Sample} operator removes all states of \( A \), which are reached by transitions where the sampling predicate \( \varphi \) is not true; these states are called \textit{transient}. When exploring the state space of \( A \), both transient and non-transient reachable states need to be stored as they are computed. By contrast, when exploring the state space of \( B \), transient states need to be computed in order to find non-transient successor states, but once these states are found, the transient states need not be stored. Moreover, some of the variables that are not history-free in \( A \) no longer influence (non-transient) successor states in \( B \) and thus in effect become history-free. This results in further savings in memory for storing states.

The savings are particularly pronounced when sampled constraints are composed. Consider two constraints \( C \) and \( D \), and two sampled constraints \( C' = (\text{Sample } C \text{ at } \varphi) \) and \( D' = (\text{Sample } D \text{ at } \psi) \). The sampled composition \( C' \parallel D' \) is an adequate model for the composed systems if the sampling predicates \( \varphi \) and \( \psi \) characterize the synchronization points of the two components; that is, if \( C \) interacts with \( D \) whenever \( \varphi \) becomes true, and \( D \) interacts with \( C \) whenever \( \psi \) becomes true. The possible interleavings of transitions of \( C \) and \( D \) may lead to an explosion of transient states of \( C \parallel D \) (states in which either \( \varphi \) or \( \psi \) is false), which can be avoided by exploring instead \( C' \parallel D' \). In other words, if the interaction between two component systems can be restricted, then some of the state-explosion problem may be avoided. In complex systems with many components but well-defined interactions
between the components, the computational savings, both in time and memory, can be enormous.

In order to carry out BDD based symbolic reachability analysis (Algorithm 2.5), we need BDD representations of the initial and update predicates. In the following, we first define the initial and update predicates of composite and sampled transition constraints from the initial and update predicates of the components. Then we present a nested-search algorithm that explores the state space of a hierarchical transition constraint efficiently. The nested-search algorithm uses an implicit, algorithmic representation of the update predicate of a hierarchical constraint for image computation.

5.3.1 Explicit definition of hierarchical constraints

Recall that a primitive transition constraint has an initial predicate and an update predicate. For hierarchical transition constraints, the initial and update predicates are defined inductively.

Parallel composition. If $A$ and $B$ are two compatible transition constraints, then the composition $C = A \parallel B$ is the transition constraint with the initial and update predicates constructed inductively as:

1. $I_C(X_C') = I_A(X_A') \land I_B(X_B')$, and $U_C(X_C, X_C') = U_A(X_A, X_A') \lor U_B(X_B, X_B')$.

Sampling. Consider the constraint $E = Sample D at \varphi$. The initial predicate of $E$ is equal to the initial predicate of $D$. Let $X = X_E = X_D$. The update predicate of $E$ is constructed as follows: For all natural numbers $i \geq 0$, define

- $U^0(X, X') = U_D(X, X') \land \varphi(X, X')$, and $L^0(X, X') = U_D(X, X') \land \neg \varphi(X, X')$.
- $U^{i+1}(X, X') = U^i(X, X') \lor(\exists Y)(L^i(X, Y) \land U_D(Y, X') \land \varphi(Y, X'))$, and $L^{i+1}(X, X') = L^i(X, X') \lor(\exists Y)(L^i(X, Y) \land U_D(Y, X') \land \neg \varphi(Y, X'))$

Then the update predicate of $E$ is equal to the limit of the sequence $U^0, U^1, U^2, \ldots$ The reachable state set of a transition constraint can be computed by iterated application of the update predicate. For this purpose, it is theoretically possible to construct, using the above definitions, a BDD for the symbolic update predicate of a hierarchical module. In practice, however, during the construction the intermediate BDDs often blow up and results in memory overflow. For parallel composition, it is a common trick to leave the
update predicate conjunctively decomposed and represent it as a set of BDDs, rather than computing their conjunction as a single BDD [TSL+90]. Early quantification heuristics are then used to efficiently compute the image of a state-set under a conjunctively partitioned update predicate. For sampling, we propose a similar approach.

5.3.2 Efficient computation with implicit update predicates

For symbolic reachability, it suffices to represent the symbolic update predicate of a module not directly, as a BDD, but implicitly, as an algorithm that given a state set, computes the set of successor states. This algorithm can then be iterated for reachability analysis and more general verification problems. Given a constraint $A$, recall that the single-step successor function of $A$ is a function $R_A^1$ from state sets of $A$ to state sets of $A$. Let $\sigma$ be a set of states of $A$. Then $R_A^1(\sigma)$ is the set of all states $t$ of $A$ such that there is a state $s \in \sigma$ with $U_A(s, t)$; that is, $R_A^1(\sigma)$ is the image of $\sigma$ under the update predicate $U_A$. It is straightforward to compute $R_A^1(\sigma)$ for flat constraints. For complex modules, the single-step successor function is computed recursively. In order to carry out this recursive computation, we need for each constraint $A$, a function that computes the update predicate restricted to a set of states. Given a set of states $\sigma$, $U_A^1(\sigma)$ is the update predicate of $A$ restricted to the states in $\sigma$. Formally $U_A^1(\sigma) = \{(s, t) | U_A(s, t) \text{ and } s \in \sigma\}$.

**Parallel composition.** Consider the constraint $C = A \parallel B$ and a set $\sigma$ of states of $A$. Let $U_A^1(\sigma)$ and $U_B^1(\sigma)$ be the restricted update predicates with respect to $\sigma$ for $A$ and $B$, respectively. Then

$$U_C^1(\sigma) = U_A^1(\sigma) \land U_B^1(\sigma), \text{ and } R_C^1(\sigma) = \{t | \exists s.(s, t) \in U_C^1(\sigma)\}.$$

**Sampling.** Consider the constraint $C = \text{Sample A at } \varphi$, and a set $\sigma$ of states of $C$. Let $R_A^1(\sigma)$ be the image of $\sigma$ for $A$, and let $U_A^1(\sigma)$ be the update predicate of $A$ restricted to $\sigma$. Then $R_C^1(\sigma)$ and $U_C^1(\sigma)$ are computed by the nested-search procedure described in Algorithm 5.3.2.

**Algorithm 5.3.2**

\{Given constraint $C = \text{Sample A at } \varphi$, function to generate single-step successor $R_A^1$, function to generate restricted update predicate $U_A^1$, and state set $\sigma$, compute $R_C^1(\sigma)$ and $U_C^1(\sigma)$ \}

$\text{SecondLevelReachSet} := \sigma$
CHAPTER 5. REFINEMENT WITH SAMPLING

\[ \text{SecondLevelUpdate} := \{ (s,s) \mid s \in \sigma \} \]
\[ \text{FirstLevelUpdate} := \{ \} \]
\[ \text{StepLevelUpdate} := U^1_A(\sigma) \]
\[ \text{loop} \]
\[ \text{SecondLevelUpdate} := \text{SecondLevelUpdate} \cup \]
\[ \{ (s,t) \mid \exists s'.(s,s') \in \text{SecondLevelUpdate} \land (s',t) \in \text{StepLevelUpdate} \land \neg \varphi(s',t) \} \]
\[ \text{FirstLevelUpdate} := \text{FirstLevelUpdate} \cup \]
\[ \{ (s,t) \mid \exists s'.(s,s') \in \text{SecondLevelUpdate} \land (s',t) \in \text{StepLevelUpdate} \land \varphi(s',t) \} \]
\[ \text{SecondLevelReachSet} := \text{SecondLevelReachSet} \cup \{ t \mid \exists s.(s,t) \in \text{StepLevelUpdate} \land \neg \varphi(s,t) \} \]
\[ \text{StepLevelUpdate} := U^1_A(\text{SecondLevelReachSet}) \]
\[ \text{until} \ (\text{SecondLevelUpdate} \ \text{does not change}) \]
\[ U^1_C(\sigma) := \text{FirstLevelUpdate} \]
\[ R^1_C(\sigma) := \{ t \mid \exists s.(s,t) \in U^1_C(\sigma) \} \]
\[ \text{return}(U^1_C(\sigma), R^1_C(\sigma)) \]

In contrast to a BDD for \( U_C(X, X') \), which directly represents the transition relation of constraint \( C \), the recursive algorithm for computing the function \( R^1_C \) implicitly represents the same information. In practice, a mixture of direct symbolic representation of transition relations (for small modules) and implicit image computation (for complex modules) will be more efficient.

5.4 Assume-guarantee refinement with Sample

We generalize the methodology for assume-guarantee style refinement checking given in [HQR98] to accommodate the Sample operator.

Witness modules. Recall that projection refinement can be verified using a transition-invariant check (Proposition 3.4), whose complexity is linear in the state spaces of both \( A \) and \( B \). In order to check refinement, it is sufficient to first find a witness module and then check projection refinement (Proposition 3.5.) If the sample operator needs to be applied to the implementation to relate it to the specification, the witness could be composed either before or after applying the Sample operator.

Proposition 5.4 [Sampled Witnesses] Consider two transition constraints \( A \) and \( B \), and a predicate \( \varphi \) on the variables of \( A \) such that \( B \) is refinable by \( A \). Let \( W \) be a module such that the interface variables of \( W \) include the private variables of \( B \), and are disjoint from
the observable variables of $A$. Then (1) $B^u$ is projection refifiable by $(\text{Sample } (A\parallel W) \text{ at } \varphi)$, and (2) $(\text{Sample } (A\parallel W) \text{ at } \varphi) \preceq_L B^u$ implies $(\text{Sample } A \text{ at } \varphi) \preceq_L B$.

**Proof:** The proof is similar to that of Proposition 3.5.

Proposition 5.4 is a generalization of the earlier Proposition 3.5. The latter can be obtained by setting $\varphi$ to true in Proposition 5.4.

**Assume-guarantee reasoning.** The state space of a transition constraint may be exponential in the size of its description. Consequently, even checking projection refinement may not be feasible. However, typically both the implementation $A$ and the specification $B$ consist of the parallel composition of several transition constraints, in which case it may be possible to reduce the problem of checking if $A \preceq_L B$ to several subproblems that involve smaller state spaces. The assume-guarantee rules allows us to conclude $A \preceq_L B$ as long as each component of the specification $B$ is refined by the corresponding components of the implementation $B$ within a suitable environment. The apparent circularity in the environment assumptions is resolved by an induction over time. If we operate with modules, the acyclic await dependencies of legal modules breaks this circularity. Since we want to carry out assume-guarantee reasoning with the Sample operator, which gives transition constraints, not modules, we impose a well-founded order on the specification components [McM98] to break the circularity. See [Qad99] for how Proposition 5.5 is related to the assume-guarantee rules for modules given in Chapter 3.

**Proposition 5.5** [Assume-Guarantee [McM98]] Let $A = A_1 \parallel A_2 \parallel \ldots \parallel A_n$ and $B = B_1 \parallel B_2 \parallel \ldots \parallel B_m$ be transition constraints. Let $\prec$ be a well-founded order on the components of $B$, let $Z(B_i) = \{B_j \mid B_j \prec B_i\}$, and let $Z^C(B_i) = \{B_j \mid B_j \notin Z(B_i)\}$. For each $B_i$, let $C_i$ be some composition of transition constraints from $A$, let $D_i$ be some composition of transition constraints from $Z(B_i)$, and let $E_i$ be some composition of transition constraints from $Z^C(B_i)$. If $C_i^\tau \parallel D_i^\tau \parallel E_i^{\tau-1} \preceq_L B_i^\tau$ for all $1 \leq i \leq m$ and $\tau \in \mathbb{N}$, then $A \preceq_L B$.

Proposition 5.5 produces proof obligations of the form $C_i^\tau \parallel D_i^\tau \parallel E_i^{\tau-1} \preceq_L B_i^\tau$. For each specification component $B_i$, the corresponding implementation component that is intended to implement the functionality specified by $B_i$ is $C_i$, and the constraining environments are $D_i$ and $E_i$. This obligation can be discharged by a reachability computation on $C_i\parallel D_i\parallel E_i$. At each stage of the reachability one has to merely check that every transition possible in $C_i\parallel D_i$ is also allowable in $B_i$. Note that while the transition constraint $E_i$ is used to
constraint the reachable states of $C_i \parallel D_i$, it is not used to constrain the transition invariant check at each step.

While discharging the obligation for $B_i$ according to Proposition 5.5, we would like to keep the state spaces to be small. Thus, it is preferable to choose most components of the constraining environment from the specification. Unfortunately, due to lack of detail, the specification does not have sufficiently abstract definitions of internal variables of the implementation. For all transition constraints $A$, $B$, and $F$, if $A \preceq_L B \parallel F$ and $B$ is refinable by $A$, then $A \preceq_L B$. Thus, we can arbitrarily “enrich” the specification by composing it with new transition constraints. Before applying the assume-guarantee rule, we may add components to the specification and prove $A \preceq_L B \parallel F_1 \parallel \cdots \parallel F_k$ instead of $A \preceq_L B$. The new transition constraints $F_1, \ldots, F_k$ are called abstract constraints, and they usually give high-level descriptions for some implementation variables, in order to provide a sufficient supply of abstract components while applying Proposition 5.5. While witness modules are introduced “on the left” of a refinement relation, abstract constraints are introduced “on the right.” Note that abstract constraints are generalizations of abstraction modules from Proposition 3.7.

Suppose the left side of a refinement relation is of the form $\text{Sample}(A \parallel B) \varphi$. It is not directly possible to apply the assume-guarantee rule from Proposition 5.5 in such cases. However, we can distribute the Sample operator with respect to parallel composition using Proposition 5.2. In practice, Proposition 5.2 tends to provide abstractions that are too coarse to be useful. To see why, imagine $A$ and $B$ as modules, each constraining the other’s inputs. By distributing the Sample inside the parallel composition, $B$ is allowed to constrain the inputs to $A$ only at the sampling instants. The inputs to $A$ are essentially unconstrained between sampling instants. Symmetrically, the inputs to $B$ are constrained at sampling instants by $A$ and left unconstrained between sampling instants. In several common situations, the interactions between $A$ and $B$ can be orthogonalized into (1) functionality, and (2) timing of the communication protocol used for the interaction. The functionality determines values at the sampling instants, whereas the timing determines how these values propagate between sampling instants.

Suppose $T_A$ and $T_B$ are transition constraints that specify how the inputs to $A$ and $B$ behave between sampling instants. Then, we can use Proposition 5.3 to distribute the Sample operator, while constraining the inputs to $A$ by $T_A$ and the inputs to $B$ by $T_B$. Thus, we get the following generalization of the assume-guarantee rule, with the Sample
operator.

**Theorem 5.1 [Assume-Guarantee with Sample]** Let \( A = A_1 || A_2 || \ldots || A_n \) and \( B = B_1 || B_2 || \ldots || B_m \) be transition constraints. Let \( T_i \) be a transition constraint on the observable variables of \( A_i \), for \( 1 \leq i \leq n \). Let \( \prec \) be a well-founded order on the components of \( B \), and let \( T, Z \) and \( Z^C \) be defined as follows: \( T = \{ T_1, \ldots, T_n \} \), \( Z(B_i) = \{ B_j \mid B_j \prec B_i \} \), and \( Z^C(B_i) = \{ B_j \mid B_j \not\in Z(B_i) \} \). For each \( B_i \), let \( C_i \) be some composition of transition constraints from \( A \), let \( U_i \) be some composition of transition constraints from \( T \), let \( D_i \) be some composition of transition constraints from \( Z(B_i) \), and let \( E_i \) be some composition of transition constraints from \( Z^C(B_i) \). If \( A \preceq_L T_1 || \ldots || T_n \), and \( \psi \) is any predicate such that \( (\text{Sample } (C_i || U_i) \text{ at } \psi^\tau) || D_i^\tau || E_i^{\tau-1} \preceq_L B_i^\tau \) for all \( 1 \leq i \leq m \) and \( \tau \in \mathbb{N} \), then \( (\text{Sample } A \text{ at } \psi) \preceq_L B \).

In the above theorem, note that the antecedent \( A \preceq_L T_1 || \ldots || T_n \) can itself be discharged by traditional assume-guarantee reasoning (Proposition 5.5.)

### 5.5 Refinement proof

Recall the high-level specification \( GCDSpec1 \), intermediate specification \( GCDSpec2 \), and implementation \( Impl \) from the previous sections. As stated in Section 5.2, the refinements we would like to verify are:

\[
(\text{Sample } GCDSpec2 \text{ at } (\neg\text{inprogress'}) \preceq_L GCDSpec1 \\
(\text{Sample } Impl \text{ at } sync') \preceq_L GCDSpec2)
\]

In this section, we will focus on how to carry out the second refinement, which relates the intermediate specification \( GCDSpec2 \) to the implementation \( Impl \). We first observe that \( GCDSpec2 \) is not projection refinable by \( Impl \), due to the presence of private variables in \( GCDSpec2 \) that represent point-to-point communication channels. The module \( Impl \) has a single channel that is shared by all modules using TDMA. The specification, while more abstract in time, provides individual point-to-point channels for communication. Each round of \( GCDSpec2 \) corresponds to multiple rounds of \( Impl \), during which a frame is transmitted. It is possible to relate the values that appear on the shared implementation channel, at particular time-slots during the communication of a frame, to values that appear on particular point-to-point channels in the specification. For instance, the values of the specification
variables $a_{\text{out}}$ and $b_{\text{out}}$ at the end of each round are expected to be equal to the values occurring in time-slots 1 and 2 of the frame. We can write a witness module $\text{IntfW}$ that looks at the implementation channel during the transmission of the frame, collects the values at time-slots 1 and 2, and assigns them to $a_{\text{out}}$ and $b_{\text{out}}$, respectively. If the values in time-slots 1 and 2 are valid, then $\text{valid}_1$ is set to true. Further, the values assigned to $a_{\text{out}}$, $b_{\text{out}}$, and $\text{valid}_1$ are retained till the end of the frame. Similar witness modules $\text{DoneW}$ and $\text{CompW}$ can be written. All these witnesses take inputs from the channel as shown in Figure 5.2(b). Let $\text{ImplW}$ be the module given by

$$\text{ImplW} = \text{Impl} \parallel \text{IntfW} \parallel \text{DoneW} \parallel \text{CompW}.$$ 

Note that the witnesses do not interfere with the channel—they merely observe the values on the channel. Due to Proposition 5.4, it suffices to check that $(\text{Sample ImplW at sync'}) \preceq_L \text{GCDSpec2}$. Recall $\text{GCDSpec2} = \text{IntfS} \parallel \text{DoneS} \parallel \text{CompS}$. We use the order $\text{DoneS} \prec \text{CompS}$ $\prec \text{IntfS}$ and apply Theorem 5.1. Let us consider the component $\text{CompS}$ of $\text{GCDSpec2}$. The component of $\text{Impl}$ that is intended to implement the functionality of $\text{CompS}$ is $\text{Comp}$. We wish to check if:

$$\forall \tau \geq 0, (\text{Sample Comp at sync'})^\tau \preceq_L \text{CompS}^\tau.$$ 

This check fails because the outputs of module $\text{CompS}$, namely, $a_{\text{in}}$, $b_{\text{in}}$ and $\text{valid}_3$ are not present in module $\text{Comp}$. Adding the witness module and appropriately constraining it, we obtain the obligation:

$$\forall \tau \geq 0, (\text{Sample (Comp} \parallel \text{CompW} \parallel \text{Channel} \parallel \text{Beacon) at sync'})^\tau \preceq_L \text{CompS}^\tau.$$ 

This still fails, because we have not constrained the inputs of $\text{Comp}$. In this obligation, the specification $\text{CompS}$ looks at the inputs $\text{small}$, $\text{big}$, and $\text{valid}_2$ in every round and produces corresponding outputs $a_{\text{in}}$, $b_{\text{in}}$ and $\text{valid}_3$ in the next round. The implementation $\text{Comp}$ anticipates two values at time-slots 3 and 4 (which correspond to $\text{small}$ and $\text{big}$, respectively) of every frame. If these inputs are valid, then $\text{Comp}$ generates values in time-slots 5 and 6 (which correspond to $a_{\text{in}}$ and $b_{\text{in}}$ respectively) of the next frame. The module $\text{Comp}$ makes the following assumptions:

1. The inputs are available at time-slots 3 and 4.
2. Either both inputs are available at a given frame, or none of the inputs are available.
CHAPTER 5.  REFINEMENT WITH SAMPLING

3. If both inputs are available, the first input (from time-slot 3) is smaller than the second input (from time-slot 4.)

In our refinement obligation, the inputs to Comp have to be constrained both at the sampling instants and between sampling instants, in order to satisfy the above assumptions. The specification component DoneS supplies the constraint at sampling instants, which ensures that assumptions 2 and 3 are satisfied. The timing assumption DoneW constrains the inputs between sampling instants and ensures that assumption 1 is satisfied. Thus we get the proof obligation

\[ \forall \tau \geq 0. \quad (Sample \ (Comp \ || \ Comp \ W \ || \ Channel \ || \ Beacon \ || \ DoneW) \ at \ sync')^\tau \]

\[ \| DoneS^\tau \ \leq_L \ \ CompS^\tau \]

Similarly, we can verify the correctness of modules Done and Intf separately. The complete refinement proof, which is a direct application of Theorem 5.1, uses the ordering DoneS < CompS < IntfS:

\[ \forall \tau \geq 0. \quad (Sample \ (Done \ || \ DoneW \ || \ Channel \ || \ Beacon \ || \ IntfW) \ at \ sync')^\tau \ \leq_L \ \ DoneS^\tau \]

\[ \forall \tau \geq 0. \quad (Sample \ (Comp \ || \ Comp \ W \ || \ Channel \ || \ Beacon \ || \ DoneW) \ at \ sync')^\tau \]

\[ \| DoneS^\tau \ \leq_L \ \ CompS^\tau \]

\[ \forall \tau \geq 0. \quad (Sample \ (Intf \ || \ IntfW \ || \ Channel \ || \ Beacon \ || \ CompW) \ at \ sync')^\tau \]

\[ \| CompS^\tau \ || \ DoneS^\tau \ \leq_L \ \ IntfS^\tau \]

\[ (Sample \ (Intf \ || \ IntfW \ || \ Done \ || \ DoneW \ || \ Comp \ || \ Comp \ W \ || \ Channel \ || \ Beacon) \ at \ sync') \ \leq_L \ \ GCDSpec2 \]

Each of the obligations above the line involves a single implementation component, possibly along with specification components, witnesses and abstract constraints to constrain the inputs. They were automatically discharged by Mocha. We thus conclude that (Sample Impl at sync') \ leq_L \ GCDSpec2.
Chapter 6

Verification of the VGI Processor

The VGI chip [STUR98] is an array of DSP processors designed to be part of a system for web-based image processing [SR97]. The VGI chip contains a total of 96 processors and has approximately 6M transistors. Of the 96 processors, 64 are 3-stage pipelined compute processors. Each compute processor has about 30,000 logic gates. Data is communicated between the processors by means of FIFO queues. No assumption is made about the relative speeds at which data is produced and consumed in the processors. Hence, to transfer data reliably an elaborate handshake mechanism is used between the sender and the receiver. In addition, the interaction between the control of the pipeline and the control of the communication unit is quite complex. In this chapter we describe how we wrote an abstract specification of the VGI chip, and carried out a refinement check between the specification and the implementation.

6.1 Overview

The VGI design was described partly in VHDL and partly in the form of circuit schematics. We translated the design into the language of Reactive Modules [AH96], which is the input language to our model checker Mocha [AHM+98]. After a number of discussions with the designers, we produced a formal specification of the design which embodies the programmer’s view of the system, also in Reactive Modules. The sheer size of the design together with the well-known state explosion problem precluded the direct use of model checking techniques to verify the implementation against the specification. Existing techniques that flatten the design hierarchy and use BDD-based state exploration [BHSV+96]
can verify designs with at most 50-60 latches reliably. Clearly, the VGI design, which has about 800 latches per compute processor, is well beyond the scope of such tools. We demonstrate how model checking can be scaled up using assume-guarantee reasoning to handle the VGI design. To the best of our knowledge, the largest design that has been ever verified using model checking has been reported by Eiriksson [Eir98]. Eiriksson's effort uses compositional techniques similar to the ones used in this paper. However, standard compositional techniques for decomposing the verification task did not readily apply for the VGI, because the implementation and specification operate on different time scales (several consecutive implementation steps realize a single specification step.) We developed novel compositional techniques for decomposing refinement proofs with variable time scales. We then applied these techniques to obtain proof obligations that were small enough to be discharged automatically by Mocha. In the process, we found several subtle bugs that were unknown to the designers. Three of these bugs will be explained in the discussion in Section 6.5.

**Step 1: formal specification.** A significant part of the verification effort was invested in producing a correct specification. Only an informal specification of the design existed in the form of English description and elaborate timing diagrams. This and the fact that no behavioral description of the design was available (the datathat was designed directly in schematic) made the task of producing the specification even more difficult.

A number of features are desirable in the specification for the VGI chip. First, the specification should be at a level of abstraction such that a high degree of confidence in its correctness can be established by informal means such as code review. More specifically, the specification should embody the view that the programmer/compiler has of the VGI chip, which is that of a dataflow architecture with a set of processing elements connected through queues. For this high-level view, every processing element behaves as if each instruction is executed atomically in one step, and the communication circuitry between the processors behaves like FIFO queues. The behavior of a programs written with this high-level view should not depend on the delay in transferring a data token from one processor to another. Such FIFO queues can be modeled using nondeterministic delay. This makes necessary the availability of nondeterminism in the specification language.

Second, the specification should have an operational as well as a mathematical semantics. Operational semantics permits the execution of specifications; mathematical semantics permits their formal verification. Executability is especially desirable in the case of the VGI processor, because the design under consideration is part of a bigger system.
CHAPTER 6. VERIFICATION OF THE VGI PROCESSOR

Provided all essential features of the design that are necessary for correct interaction with the environment have been captured by the specification, it can be used in place of the actual design for simulating the whole system.

Third, the design itself (the “implementation”) should be describable in the same language as the specification, and a refinement preorder should be available for relating the implementation and the specification. In our case, the refinement preorder must relate two different time scales. The implementation has a clock signal clk with activity on both the HIGH and LOW phases in different parts of the design. For instance, in the execute phase of the pipeline a bus carries an operand when clk is HIGH and the result when clk is LOW. But the specification does not mention clk at all. In fact, the whole computation happens in just one step. Thus, one round in the specification is equal to two rounds in the implementation, one with \( \text{clk} = \text{HIGH} \) and one with \( \text{clk} = \text{LOW} \). Therefore, our formal notion of refinement samples the implementation whenever \( \text{clk} \) is low and checks if the sampled behavior is present in the specification.

Reactive Modules, our modeling language for both specification and implementation, has all the desirable features mentioned above — mathematical semantics, executability, and support for nondeterminism and sampling.

Step 2: formal verification. Since VGI is a very big design, model checking cannot be applied directly. Previously, assume-guarantee methods have been developed for decomposing a refinement verification task into smaller proof obligations that can be discharged automatically with a model checker. In assume-guarantee reasoning the different components of the implementation are verified in isolation by making appropriate assumptions about their environments. The environment assumptions are then discharged separately. In order to keep the sizes of the individual proof obligations within the capacity limits of model checking, it is essential to specify the environment assumptions for implementation components abstractly in terms of specification signals, using so-called “abstract constraints” (refinement maps, simulation relations) [AL88, Kur94, Lyn96, AH96, McM97, HQR98].

In the case of VGI, the specification describes the behavior of the implementation only at the sampling instants. Consequently, the abstract constraints specify the values of implementation signals only at those instants. But the correct behavior of implementation components may depend on assumptions about the environment between sampling instants. Hence, for carrying out refinement-based proofs in situations where the time scales of the
CHAPTER 6. VERIFICATION OF THE VGI PROCESSOR

implementation and specification differ, we (1) introduce a new sampling operator that can sample the signal values of a module with some environment constraint between sampling instants, and (2) generalize the assume-guarantee proof rule to work with the sampling operator. Working with specifications at an abstract level of temporal granularity is not new. Previous work on dynamic switch-level circuits encountered similar situations, where it is useful to generate gate-level circuits in which the clock is abstracted out [BJ95, KSL95]; previous work on reachability checking utilized the efficient exploration of temporal abstraction hierarchies [AHR98]. However, we are not aware of any compositional refinement checks between implementations and specifications that operate at different time scales.

In order to handle the proof obligations that are generated by our new assume-guarantee rule, we extended the model checker MOCHA with the capability for dealing with the sampling operator in refinement checks. We are not aware of any other model checker that currently offers such a capability. Using the enhanced version of MOCHA we discovered several bugs in the VGI design and fixed them. In this process, we found it extremely useful to employ MOCHA as a debugging tool that supports the concurrent activities of (re)design and formal (re)verification: design insights would suggest the definition of refinement maps for model checking, and MOCHA would produce error traces that suggest corrections to the design. In this way, design and formal verification become a single activity ("formal design") that involves similar mental processes, rather than two decoupled activities, one followed by the other with little interaction.

The goal of our verification effort is a proof that the implementation of the VGI chip refines its specification. As a result, a programmer who wishes to find out how the VGI chip would behave with his program can simulate his program on the specification of the VGI chip instead. Our refinement proof guarantees that the behaviors produced by the implementation of the VGI chip conform to behaviors produced by the specification. An interesting, but different question is whether the specification correctly captures the intent of the designer. It is not possible to answer this question formally, because intentions are informal and imprecise. However, we can check if the specification satisfies specific properties expressed in temporal logics using model checking.

Outline. In Sections 6.2 and 6.3, we describe the implementation and specification of the VGI chip in more detail. In Section 6.4, we describe our notion of refinement based on a sampling operator and introduce the corresponding verification methodology based on assume-guarantee reasoning. We first show that the verification of an arbitrary
network of compute processors can be reduced to the verification of a finite set of configurations of a single processor. But even a single processor is too large to be handled by a model checker directly, and we have to decompose it further using assume-guarantee reasoning. We conclude with a discussion of the bugs found and insights gained in Section 6.5.

6.2 Design implementation

We briefly describe the architecture of the VGI chip. The VGI chip is a DSP processor array comprising 64 compute processors, 16 memory processors and 16 I/O processors, connected by a statically programmable hierarchical network. The processors are arranged in 16 clusters with 4 compute processors, 1 memory processor and 1 I/O processor in each cluster. The compute processors are connected through queues that are programmed statically. Each memory processor has a 256 × 16 storage unit, and can be statically configured as a delay line, FIFO, lookup table or random access memory. To enhance performance, the design of the compute processor uses a two phase clocking scheme, level sensitive latches of either polarity and gated clocks. Multiple VGI chips can be connected by using the I/O processors. There is a single clock variable clk for the chip. In this work, we focus on the verification of compute processors and the data communication among them.

A compute processor in the VGI chip has a datapath unit, a control unit and a 16-word instruction memory. The datapath is 16 bits wide and the instructions are 40 bits wide. There are 6 registers organized in pairs. Each pair could either be used as two general purpose registers or as a 2-place input queue. The processor operates as a 3-stage pipeline — the fetch stage (IF), the execute stage (EX) and the communicate stage (COM). The latches separating IF and EX stages are called pipelat and those separating EX and COM stages are called lout. There is feedback from the EX stage to the IF stage, and the COM stage is meant for data transfer between processors through an elaborate handshake mechanism. A block diagram of the processor is given in Figure 6.1.

Instructions in the compute processor require up to three operands. Hence, there are three data buses running through the IF stage. These buses are fed by the three register pairs. Therefore, an instruction cannot access both registers in the same register pair. The control of the IF stage is fed by the instruction stored in the mirlreg register. The IF control logic looks at these three buses and two feedback buses from the EX stage and
selects the appropriate value to latch into the pipeline latches. The EX stage has an adder, a logical unit and a booth multiply unit. The control in this stage is fed by the instruction in the mir2reg register, which is a 1-cycle delayed version of mir1reg. If any data or control tokens need to be sent to some other processor, it is latched into the lout latches.

The interconnection between the processors is programmed statically at the beginning before any computation starts. Each compute processor can be in one of a finite number of configurations depending on its input and output queues. At the input, each register pair can be configured either as a queue or as general purpose registers. At the output, each output bus can be used or unused. For instance, Figure 6.1 shows the configuration when the register pair R2-R3 is configured as a queue, and one data output queue and the control output queue are configured to send data and control tokens respectively to other processors.

The output of the ALU abus.r can be written back to the register file or sent out on one or more queues. For receiving data/control tokens, the downstream processor should have a register pair configured as a 2-place queue. Every data or control token that is computed is latched into lout. If the first send fails, then the COM stage keeps on
CHAPTER 6. VERIFICATION OF THE VGI PROCESSOR

sending the data in \texttt{lout} until the send succeeds. Signals \texttt{send} and \texttt{sendack} are used for handshake between the sender and the receiver. In the meantime, other instructions might be executing in the \texttt{EX} stage of the pipeline. The pipeline is stalled and a signal \texttt{stallpipe} asserted when \texttt{COM} stage is trying to send a token and the instruction in \texttt{EX} stage also wants to send out a token.

A data flow computation is performed by connecting the compute processors in a hierarchical network. The instruction memory of each compute processor is programmed individually, and each processor functions as an “actor” in a data flow network, consuming data and control tokens from its input and producing data and control tokens to its output. Suppose we want to configure two processors so that one sends data to another. Then, one of the register pairs of the receiver has to be configured as an input queue, and an output bus of the sender should be connected to the input bus of the receiver by programming the network. No assumption is made about the relative speeds at which data is being produced and consumed. Hence, an elaborate handshake mechanism is used to transfer data reliably between the sender and the receiver. Since an instruction can read from multiple input queues and send to multiple output queues, care must be taken to ensure that data-flow semantics is preserved. For example, if an instruction reads data from two input queues and data is available only on the first queue and the second queue is empty, then the read from the first queue has to be delayed until data is also available in the second queue. Similarly, if an instruction wants to send to two output queues and the second queue is full, then the data is sent to the first queue and the processor stalls until the second queue has space. Irrespective of the relative speeds of the processors, data should not be lost and the same data should not be sent multiple times.

We do not model the configuration logic that configures the network, and the scan logic that is used by the hardware debugger. However, our verification scheme works for an arbitrary static configuration. After these simplifications, each compute processor model has 1700 variables, of which 800 are latch variables. A complete description of the VGI implementation is a few hundred pages long. We will not describe the implementation at that level of detail here. As we describe each refinement proof obligation, we give some details about relevant components of the implementation.
6.3 Design specification

Our goal is to come up with a specification for an arbitrary (but statically) pro-
gammed network of the compute processors. We do this by writing specifications for each
component of the compute processor: (1) computation unit, (2) data output queue, and
(3) control output queue. An input queue is treated as a part of the output queue of the
upstream processor, and hence does not have a separate specification component. The module ISA is a very simple specification of the computation unit — data values read either
from a register or a queue, results are computed and output is written to the output queues
and/or written back to a register, as a result of executing an instruction. The module DataQueue is a description of a 4-place FIFO queue, where each token is a 16-bit value.
ControlQueue is similar except that each token is a 2-bit value. The specification for any
particular processor configuration can be obtained by composing the component specifications of its computation unit and its output queues. For example, the specification for the configuration shown in Figure 6.1 can be obtained by appropriately composing a module ISA, one instantiation of the module DataQueue and one instantiation of the module ControlQueue. This is shown in Figure 6.2. The register pair R2-R3 is missing in this
configuration of the specification, since they have been configured as an input queue. The dotted rectangle in the lower portion of Figure 6.2 shows auxiliary variables and abstract constraints. We defer its description to Section 6.4. In the implementation, each queue is actually distributed between the sender and the receiver. When we wrote the specification, we decided to model each queue as a single module. Further, we modeled a queue as a component in the specification of the sender. Thus, we avoided specifying the complicated interface in the implementation, between the sender and its output queue.

Let \( Q_i \) be the specification module of the appropriate configuration corresponding

Figure 6.3: Specification module with signaling details

Figure 6.4: ISA with two input and two output queues
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgMem</td>
<td>Contains the program run by the ISA</td>
</tr>
<tr>
<td>NextProgramCounter</td>
<td>Generates the next value of the program counter</td>
</tr>
<tr>
<td>CCreg</td>
<td>Manages the condition code register, using the ccwr field of the current instruction</td>
</tr>
<tr>
<td>Regfile</td>
<td>Manages the register file</td>
</tr>
<tr>
<td>IndexDecode</td>
<td>Decodes the three bit code for the source and destination registers</td>
</tr>
<tr>
<td>Illegal</td>
<td>Checks if the source and destination fields of the instruction are legal</td>
</tr>
<tr>
<td>Operand</td>
<td>Generates operands to the ALU</td>
</tr>
<tr>
<td>StallEmpty</td>
<td>Asserts stallempty signal whenever there is a read and one of the input queues is empty</td>
</tr>
<tr>
<td>InputQueueControl</td>
<td>Generates signals to control reading from the input queues</td>
</tr>
<tr>
<td>OutputQueueControl</td>
<td>Generates signals to control writing to the output queues</td>
</tr>
<tr>
<td>Alu</td>
<td>Computes result and carry from operands and operation</td>
</tr>
</tbody>
</table>

Table 6.1: Component modules of ISA

...
responds with data on \texttt{dataout} or by asserting \texttt{empty\_stnn}.

Most of the bugs that we found in this design were results of the interaction between the ISA and the input/output queues. Figure 6.4 shows a configuration where a pipeline is connected to two input queues — \texttt{input\_queue1} and \texttt{input\_queue2} — and two output queues — \texttt{output\_queue1} and \texttt{output\_queue2}. Because no assumption is made about speed at which the pipelines produce or consume data, an output queue could be full when the pipeline wants to write data, and an input queue could be empty when the pipeline wants to read data. Two signals \texttt{stallpipe} and \texttt{stallempy} are used by the pipeline to handle such situations. Suppose the pipeline wants to read from both input queues, and suppose one of the queues, say \texttt{input\_queue2}, is empty. Then the pipeline asserts \texttt{stallempy}, which in turn aborts the read from \texttt{input\_queue1} as well. Similarly, suppose that one of the output queues \texttt{output\_queue1} or \texttt{output\_queue2} is full. Then the pipeline asserts \texttt{stallpipe}, which aborts the read from the input queues. Thus, in a single round, the input queues first raise a signal if they do not have data in the first subround, and then monitor \texttt{stallpipe} and \texttt{stallempy} in a subsequent subround to decide if the read needs to be aborted. We say that the pipeline \textit{stalls} if \texttt{stallpipe} or \texttt{stallempy} is asserted. Complications arise when the pipeline goes into a stall and when it comes out of a stall.

In the ISA an instruction takes only one round to execute — in a single round, data values read either from a register or a queue, results are computed and output is written to the output queues and/or written back to a register. The pipeline takes three rounds to execute the instruction, and multiple instructions could be executing at different stages of the pipeline at any given instant. The invariant that synchronizes the operation of the specification and implementation is the following: The instruction being executed by the ISA is the instruction in the IF stage of the implementation. The specification queue is a 4-place queue because there is no assumption in the implementation about the relative speeds of production and consumption of data at the two ends of the output queue. If the receiver does not read data for a while, the sender could have executed four instructions each sending to the output queue with the fourth and fifth instructions circulating in the EX and IF stages respectively.
6.4 Refinement proof

We would like to prove that the implementation of the VGI processor refines its specification in as automatic a way as possible. Two features of the implementation make this verification task specially daunting.

- The implementation consists of a possible maximum of 64 compute processors. Each processor is quite big with around 800 latches and 1700 variables. The sheer size of the implementation precludes a direct use of model checking and makes compositional reasoning essential. In assume-guarantee reasoning, the different components of the implementation can be verified in isolation by making appropriate assumptions about their environment. These environment assumptions can then be discharged separately.

A crucial aspect of this decomposition process is the use of “abstract constraints”. We can illustrate this in an abstract setting in the following way. Consider, for example, an implementation that is the parallel composition of two modules $P$ and $Q$ and let $P'$ and $Q'$ be their respective specifications. We would like to verify the modules $P$ and $Q$ one at a time. The environment of $P$ might contain signals that are not present in the specification. Hence, we write abstract definitions of these implementation signals in terms of specification signals in the form of a module $R_P$ and use it along with $Q'$ to construct the environment $E_P = Q' \| R_P$ of $P$. A similar approach is taken for module $Q$ to generate its environment $E_Q$. Then, we can use the following proof rule.

$$
\frac{P \| E_P \; \preceq_L \; E_Q}{Q \| E_Q \; \preceq_L \; E_P} \quad \frac{P \| Q \; \preceq_L \; \frac{E_P \| E_Q \; \preceq_L \; P' \| Q'}{P \| Q}
$$

The use of circular environment assumptions as shown in the proof template above is crucial for decomposing verification tasks.

- The implementation is based on level-sensitive latches synchronized by a single clock. There are latches of both kind — transparent high and transparent low, and computation is performed in both phases of the clock in different parts of the implementation. Moreover, there are a number of gated latches, i.e., latches whose enabling signals depend on signals other than the clock. We model these phenomena through an explicit clock variable $\text{clk}$ that toggles every round. Thus, a round in the implementation corresponds to half a clock cycle. Being at a more abstract level, the specification does...
CHAPTER 6. VERIFICATION OF THE VGI PROCESSOR

not mention the clock at all and a round in the specification corresponds to two rounds of the implementation. One way to compare an implementation with a specification that operates at a coarser time scale is to sample the values of the implementation signals at appropriate time instants. We would then like to show that every sampled trace of the implementation is a trace of the specification.

Since the implementation and specification have different time scales, the abstract constraints will constrain the value of implementation signals only at the sampled time instances. But, sometimes a module in the implementation might depend on the behavior of the environment between sampling points. For example, it might be important that the environment maintains the value of a signal constant from one sampling instant to another. Therefore, the sampling operator might need to constrain the behavior of a module between sampling instants.

We describe the refinement proof in two steps. In the first step, the refinement proof of an arbitrary network of compute processors is reduced to smaller proofs that each involve only one compute processor. In the second step, the proof of as single compute processor configuration is further decomposed.

6.4.1 Step 1: Network of compute processors

Each compute processor in VGI starts computation in the positive phase of the clock and finishes the computation in the negative phase of the clock. We decided to sample at the end of each computation. Hence, the sampling predicate \( \varphi \) is \( \text{clk} = \text{LOW} \). In the rest of this section, we use \( \varphi \) to refer to \( \text{clk} = \text{LOW} \). In the previous section we showed how to obtain a specification for an arbitrary network of processors. Our goal is to verify that an arbitrary network of processors implements its corresponding specification, using refinement checking. Let \( P_1, P_2, \ldots, P_n \) be the compute processors in an arbitrary network, and let \( Q_1, Q_2, \ldots, Q_n \) be their respective specifications. The verification problem is to check that

\[
\text{Sample} \ (P_1 \parallel P_2 \parallel \cdots \parallel P_n) \ \text{at} \ \varphi \preceq_L (Q_1 \parallel Q_2 \parallel \cdots \parallel Q_n).
\]

For the correct functioning of a processor it is essential that all input signals to it change only when \( \text{clk} \) is \( \text{HIGH} \). Let \( T_i \) be a module that says that all external signals of \( P_i \) change only when \( \text{clk} \) is \( \text{HIGH} \). We use Theorem 5.1 to decompose this proof as follows:
6. VERIFICATION OF THE VGI PROCESSOR

\[
\begin{align*}
(Sample(P_i\parallel T_i) \text{ at } \varphi) & \preceq_L Q_i \text{ for all } 1 \leq i \leq n \\
\vdots & \vdots \\
Sample(P_1\parallel P_2\parallel \cdots \parallel P_n) \text{ at } \varphi & \preceq_L Q_1\parallel Q_2\parallel \cdots \parallel Q_n
\end{align*}
\]

The second antecedent says that the inputs of any processor in the network change only when \(\text{clk}\) is \text{HIGH}. Since any input to a processor has to be the output of some other processor, this antecedent can be discharged easily by proving that for all \(1 \leq i \leq n\), the outputs of \(P_i\) change only when \(\text{clk}\) is \text{HIGH}. This is an easy proof local to each processor.

In the first antecedent, there are \(n\) symmetric proof obligations, one for each \(P_i\). Each \(P_i\) can be in any one of a finite number of configurations. Moreover, if \(P_j\) and \(P_k\) are in the same configuration, then the \(j\)th and \(k\)th proofs are identical except for variable renaming.

Let \(C_{\text{VGI}}\) be the finite set of configurations for the compute processors. For \(X \in C_{\text{VGI}}\), let \(Y\) be its specification and \(T_X\) be the environment constraint that says that all inputs change only when \(\text{clk}\) is \text{HIGH}. Then, it suffices to prove that for each \(X \in C_{\text{VGI}}\), we have that \(Sample(X\parallel T_X) \text{ at } \varphi \preceq_L Y\). Thus, we decompose the proof of a 64 processor network to proofs about individual processor configurations that have 800 latches each. This is still beyond the scope of monolithic model checking. In the next section, we show how we discharged this proof for a single processor configuration, with further applications of the generalized assume-guarantee rule described earlier. We implemented support for the \(Sample\) operator in \text{Mocha}, in order to carry out this refinement check.

6.4.2 Step 2: Single processor configuration

We describe the proof of a processor configuration with one output queue. The proof decomposes very nicely as we trace the flow of data from the output to the input in the processor.

Queue lemma

The objective here is to check if the output queue behaves properly in the sense that data written into the queue is not lost, no data is written twice, and correct behavior is preserved going into and coming out of stalls (either \text{staleempty} or \text{stallpipe}.). The queue registers and the queue control logic are both physically distributed in the sending and receiving processor. Figure 6.5 shows the various components of the queue. Modules \text{hsout}, \text{lout} and \text{loutctrl} are components of the sending processor, whereas \text{hsin}, \text{lin}, \text{linctrl}, \text{reg\_d0}, \text{reg\_d0\_ctrl}, \text{reg\_d1}, \text{reg\_d1\_ctrl}, and \text{qctrl} are components of the
module QueueEnv
external clk: bool
interface read, save, stall_empty, stall_pipe, send: bool
atom controls send, read, save, stall_pipe, stall_empty
reads send, read, save, stall_pipe, stall_empty awaits clk
update
[] ~clk’ -> read’ := read; save’ := save; stall_pipe’ := stall_pipe;
    stall_empty’ := stall_empty; send’ := nondet
[] clk’ -> read’ := nondet; save’ := nondet; stall_pipe’ := nondet;
    stall_empty’ := nondet; send’ := send
endatom
endmodule

Figure 6.6: Abstract timing constraints for the queue
receiving processor. Module Clk generates a two phase clock that is fed to all modules in the implementation (we omit these connections in the figure to avoid clutter.) A state machine in the qctrl1 module maintains information about the number of elements in the queue, and where the elements are. Modules hout and hsin control the communication between the sender and receiver using a handshaking mechanism. The sender sends data from the register lout, and the receiver receives it in lin. Registers reg_d0 and reg_d1 constitute the two place queue in the receiver. Modules loutctrl, linctrl, reg_d0_ctrl and reg_d1_ctrl generate control signals for lout, lin, reg_d0 and reg_d1 respectively.

Let

\[
\text{qimpl} = hout \parallel lout \parallel loutctrl \parallel hsin \parallel lin \parallel \\
linctrl \parallel reg_d0 \parallel reg_d0_ctrl \parallel reg_d1 \parallel reg_d1_ctrl \parallel \text{Clk}
\]

\[
\text{sampled_qimpl} = \text{sample qimpl at } \sim \text{clk}
\]

We wish to check if \(\text{sampled_qimpl} \preceq_L \text{DataQueueSpec}\). For this refinement check to pass, we need to constrain the inputs to \(\text{sampled_qimpl}\), both at sampling instances, and between sampling instances. We need to write abstract constraints at both time scales. Figure 6.6 shows the abstract constraints that constrain the inputs between the sampling instances. The constraints merely state the timing assumption that certain inputs to qimpl do not change values at certain phases of the clock. For the constraining environment at sampling instances, we need abstract constraints for the variables send, stallpipe, and abus_r. These are shown in Figures 6.7, 6.8 and 6.9 respectively. We will explain these modules in the subsequent sections.

The abstract constraint for abus_r is written in terms of the two stall signals and the output of the ALU in the specification. In order to write abstract constraints for send and stallpipe, we had to add auxiliary history variables excsend, num and sendackp. The signal excsend is true whenever the the current instruction in the EX phase wants to send, and num keeps track of the number of items in the receiver’s two place input queue. The signal sendackp predicts the implementation’s sendack. The abstract constraints are generated in terms of the specification signals and the history variables as shown in the dotted rectangle in Figure 6.2.

Let

\[
\text{qrefmaps} = \text{send_ref} \parallel \text{abus_ref} \parallel \text{stallpipe_ref}
\]

Using these abstract constraints we were able to check that:

\[
(\text{sample (qimpl \parallel QueueEnv at } \sim \text{clk}) \parallel \text{qrefmaps} \preceq_L \text{DataQueueSpec})
\]
module send_ref
  external sentnn_p, stall, ex_send:  bool
interface send:  bool
private flag:  bool
atom controls flag reads send, sentnn_p, stall, flag
  update
  [] ~(send & sentnn_p) -> flag' := false
  [] send & sentnn_p & stall -> flag' := flag
  [] send & sentnn_p & ~stall -> flag' := true
endatom
atom controls send reads send, sentnn_p, stall, flag awaits ex_send
  init
  [] true -> send' := false
  update
  [] true -> send' := (send & sentnn_p) | (ex_send' & (stall | flag))
endatom
endmodule

Figure 6.7: Abstract constraint for send

send lemma

The abstract constraint for send is shown in Figure 6.7. Recall that the implementation of the queue is distributed in both the sending and receiving processor. The specification has a monolithic 4 place queue. The abstract constraint for send relates the send signal of the specification with that of the implementation. It uses the auxiliary variable \texttt{ex\_send} which is asserted whenever the EX stage of the pipeline wants to send data to \texttt{lout}. An extra private variable \texttt{flag} is used to record if the implementation is just coming out of a stall.

The module \texttt{opctrl} generates the send signal in the implementation. In order to constrain inputs to \texttt{opctrl}, another implementation module \texttt{exctrl} is also included.

Let
\[
\begin{align*}
\text{sendImpl} &= \text{opctrl} \parallel \text{exctrl} \parallel \text{Clk} \\
\text{sampledSendImpl} &= \text{sample sendImpl at } \sim\text{clk}
\end{align*}
\]
CHAPTER 6. VERIFICATION OF THE VGI PROCESSOR

As in the queue lemma, we need to constrain the inputs to `sampledSendImpl`, both at sampling instances, and between sampling instances. We need to write abstract constraints at both time scales. The abstract constraints that constrain the inputs between sampling instants have a similar flavor as the queue proof. Let

\[
\begin{align*}
\text{EnvSend} & = \text{Env.mir2} \parallel \text{Env.sentnn}_0 \\
\text{SendRefMaps} & = \text{QueueSize.d0} \parallel \text{ExecutePhaseSend.d0} \parallel \text{ISA.Stall} \parallel \\
& \quad \text{OutputQueueControl} \parallel \text{StallEmpty} \parallel \text{mir2reg_ref} \parallel \\
& \quad \text{imem_ref} \parallel \text{progmem_ref}
\end{align*}
\]

Detailed description of modules `Env.mir2`, `Env.sentnn_0`, `QueueSize.d0`, `ExecutePhaseSend.d0`, `mir2reg_ref`, `imem_ref`, and `progmem_ref` can be found in Appendix C. Modules `ISA.Stall`, `OutputQueueControl` and `StallEmpty` are part of the ISA and can be found in Appendix A. Note that the abstract constraint for `sentnn` is produced by module `QueueSize`. It sets `sentnn` to false (`sentnn` is active when false) whenever a send is being done and there is not enough space in the queue. Using these abstract constraints we were able to check that:

\[
(s_{\text{sample}}(\text{sendImpl} \parallel \text{EnvSend}) \text{ at } \sim \text{clk}) \parallel \text{SendRefMaps} \preceq_{L} \text{send.ref}
\]

**stallpipe lemma**

The abstract constraint of `stallpipe` is shown in Figure 6.8. Intuitively, it asserts `stallpipe` whenever there is a send attempted to some output queue and the send fails. However, there are subtleties in timing, depending on when the EX phase of the pipeline wants to send, and when the ISA wants to send. In the implementation, `stallpipe` is produced by `opctrl`, whose inputs are constrained by `exctrl`. This refinement check is done using the abstract constraints for `send`, and `sentnn` of all the output queues.

Let

\[
\begin{align*}
\text{stallpipeimpl} & = \text{opctrl} \parallel \text{exctrl} \parallel \text{Clk} \\
\text{EnvStall} & = \text{Env.mir2} \parallel \text{Env.sentnn}_0 \parallel \text{Env.sentnn}_1 \parallel \\
& \quad \text{Env.sentnn}_2 \parallel \text{Env.sentnn}_3 \\
\text{StallPipeRefMaps} & = \text{QueueSize.d0} \parallel \text{QueueSize.d1} \parallel \\
& \quad \text{QueueSize.d2} \parallel \text{QueueSize.c} \parallel \\
& \quad \text{ExecutePhaseSend.d0} \parallel \text{ExecutePhaseSend.d1} \parallel \\
& \quad \text{ExecutePhaseSend.d2} \parallel \text{ExecutePhaseSend.d3} \parallel \\
& \quad \text{ISA.Stall} \parallel \text{OutputQueueControl} \parallel \\
& \quad \text{StallEmpty} \parallel \text{mir2reg_ref} \parallel \text{imem_ref} \parallel \text{progmem_ref}
\end{align*}
\]

Using these abstract constraints we were able to check that:

\[
(s_{\text{sample}}(\text{stallpipeimpl} \parallel \text{EnvStall}) \text{ at } \sim \text{clk}) \parallel \text{StallPipeRefMaps} \preceq_{L} \text{StallPipe}
\]
module StallPipe  external send_0, sentnn_p_0, d0_ex_send, send_1, sentnn_p_1, d1_ex_send, send_2, sentnn_p_2, d2_ex_send, send_3, sentnn_p_3, c_ex_send: bool
interface stall_pipe: bool
atom controls stall_pipe reads send_0, sentnn_p_0, send_1, sentnn_p_1, send_2, sentnn_p_2, send_3, sentnn_p_3
awaits c_ex_send, d2_ex_send, d1_ex_send, d0_ex_send
init
[] true -> stall_pipe': := false
update
[] true -> stall_pipe': := (send_0 & sentnn_p_0 & d0_ex_send') |
(send_1 & sentnn_p_1 & d1_ex_send') |
(send_2 & sentnn_p_2 & d2_ex_send') |
(send_3 & sentnn_p_3 & c_ex_send') endatom endmodule

Figure 6.8: Abstract constraint for stallpipe

abusr lemma

The abstract constraint for abusr is shown in Figure 6.9. Since abusr is generated by the data path of the implementation, this refinement check amounts to verifying the correctness of the data path. The correctness of the refinement map for abusr is proved using a refinement map for pipelat_a.s signals, which are an input to the EX stage. We have not been able to complete this proof using Mocha. We believe that this is essentially a combinational verification problem and existing techniques geared for it can discharge it easily.

In each lemma described above, the part of the implementation under investigation was sampled at clk equal to LOW under some timing assumptions on the inputs between sampling instants. For example, in the queue lemma, it was assumed that send signal does not change value when clk changes from LOW to HIGH, and all signals at the receiver end such as read, save change values only when clk is HIGH. All such assumptions were discharged separately.

Notice the circular dependencies between the lemmas. We also did supporting refinement proofs for mirt2reg_ref, imem_ref, progmem_ref and pipelat_a.s. The cor-
module bus_r_ref
    external stall: bool; data_in: bv16
    interface bus_r: bv16
    atom controls bus_r reads bus_r, data_in, stall
    update
        [] stall -> bus_r' := bus_r
        [] ~stall -> bus_r' := data_in
    endatom
endmodule
abus_r_ref := bus_r_ref[bus_r, data_in := abus_r, d0bus]

Figure 6.9: Abstract constraint for abus_r

rectness of the constraint for pipelat_a_s is proved by using the abstract constraint for abus_r. This lemma amounts to verifying the correctness of feedback from the EX stage to the register file and the pipelat_a_s registers. In total, about 35 lemmas need to be proved.

In every lemma except the abus_r lemma, we used symmetry arguments [McM98] to reduce the datapath width to just 1 bit. In the abus_r lemma, the symmetry is broken because of arithmetic operations and hence the full datapath width of 16 bits needs to be considered. Thus, assume-guarantee reasoning provides a very clean separation between the verification of the datapath and control of the processor. It is very clear in the overall proof that the datapath width is irrelevant in verifying the control that is moving data around. This also suggests that compositional reasoning provides a formal framework under which combinational verification of the datapath and FSM verification of the control can coexist. None of the individual lemmas took more than a few minutes on a 625 MHz DEC Alpha 21164.

6.5 Discussion

In this section, we describe the bugs we found in the design. We fixed all the bugs and verified our fixes with MOCHA. Our fixes were reviewed and approved by the designers.
1. If the sending processor writes two successive values into the queue and the receiving processor waits for one cycle and then does two successive reads, the second read returns an incorrect value.

2. Suppose \texttt{stallempty} is asserted in cycle \( n \) but released in cycle \( n + 1 \). Also, suppose send to an output queue fails in cycle \( n + 1 \). Then although \texttt{stallpipe} should be asserted in cycle \( n + 2 \), it is not and as a result the instruction in \texttt{EX} stage gets corrupted.

3. A particular sequence of events involving 4 sends and 4 reads interleaved in a specific way, with a stall at a precise moment corrupts the data in the \texttt{lout} register. This results in the loss of an output token. The error trace that led to the discovery of this bug was eight cycles long.

The following execution sequence resulted in loss of data in the queue:

- Cycles 1 and 2: First the sender sends two values to an empty queue.
- Cycle 3: The sender tries to send a third value and the send fails.
- Cycle 4: A read happens from the receiving processor. Simultaneously, the sender resends the data from the previous cycle and succeeds. Since the send in the previous cycle failed and the current instruction wants to send, \texttt{stallpipe} is asserted.
- Cycle 5: A read happens in the receiving processor. Since the processor stalled in the previous cycle, no send is attempted. However, since there is no stall in the current round, the processor advances to the next instruction and the data value currently present in \texttt{lout} is overwritten.
- Cycle 6: The receiving processor performs another reads. The sending processor sends the wrong value to the receiver.
- Cycle 7: Both the sender and receiver wait. The receiver has to wait for at least one cycle before reading the value sent in the previous cycle.
- Cycle 8: The receiver reads and gets the wrong value.

We now describe the process by which we found these bugs and the insights we gained about the interaction between design and verification. We found all these bugs
CHAPTER 6. VERIFICATION OF THE VGI PROCESSOR

while doing the proof of the Queue lemma, the lemma stating the correctness of the data transfer between the sender and the receiver. Recall that we needed abstract constraints for the environment signals bus_r, send and stallpipe. Initially, we tried to write the abstract constraints based on the definitions of these signals in the implementation. But, we got error traces. We kept on strengthening the abstract constraints till we got no error trace. At this point, we had correct abstract constraints of these environment signals that we could translate down to definitions in terms of implementation signals. These design fixes were quite complicated and we actually had to do some logic design ourselves. At this point, we were using MOCHA as a debugging tool that would test our proposed fix by throwing at it all possible sequences of events. If it generated an error trace then we could look at it and refine the fix. Thus, the distinction between verifying and designing gets blurred and actually both activities proceed in parallel. We believe that design and verification are symbiotic activities in the sense that the designer’s intuition embodied in abstract constraints aids verification and the model checker aids the designer by testing his proposed solution by throwing at it all possible situations. We believe that the mental processes involved in doing verification exist when the design is being created and therefore, given the right interface to the verification tool, it is not a big burden to do “formal design”.
Chapter 7

Conclusion

The goal of our effort to increase the class and size of designs on which refinement checking (using model checking) is viable. Below, we summarize our contributions and list open problems and possible extensions to this thesis.

We started with our modeling language, transition constraints and reactive modules, in Chapter 2. In Chapter 3 we formally defined linear and branching views of refinement, and proved a new assume-guarantee rule (Theorem 3.1) for branching refinement. We had proved Theorem 3.1 earlier for Moore machines (a restricted class of modules in which there are no await dependencies across module boundaries) using simulation relations [HQRT98]. Given witnessing simulation relations $H_1$ and $H_2$ for the branching refinements $P\|Q' \preceq_B P'$ and $P'\|Q \preceq_B Q'$ respectively (where $P, Q, P'$ and $Q'$ are Moore machines), we showed in [HQRT98] how to construct a witnessing simulation relation $H$ for the refinement $P\|Q \preceq_B P'\|Q'$. A simulation relation based proof of Theorem 3.1 remains open.

In Chapter 4, we presented a novel extension of the simulation preorder for labeled transition systems to account for fairness. Our definition enjoys a fully abstract tree semantics and has a logical characterization: $S$ fairly simulates $I$ iff every fair computation tree embedded in the unrolling of $I$ can be embedded also in the unrolling of $S$ or, equivalently, iff every Fair-\forallAFMC formula satisfied by $S$ is satisfied also by $I$. The locality of the definition led us to a polynomial-time algorithm (in Section 4.4) for checking fair simulation for finite-state systems with weak and strong fairness constraints. Investigating the properties of fair simulation, we related fair simulation to the existing definitions in literature (3-simulation and \forall-simulation), and argued in favor of our definition. Below we
CHAPTER 7. CONCLUSION

list a few related issues that we have not addressed:

- In Section 4.6, we gave a technique for constructing fair-similarity abstractions of Büchi structures. Constructing fair-similarity abstractions of Streett structures remains open.

- The bisimulation relation for structures with no fairness serves as a basis for abstractions. Indeed, if two states in the same structure are bisimilar, one can merge them to a single state, resulting in a small structure that satisfies the same specifications. In order to abstract fair structures, our definition of fair simulation should be extended to fair bisimulation, and a sound merging method should be developed.

- We briefly mentioned that Fair-$\forall$CTL induces a weaker preorder [ASB+94] than both Fair-$\forall$CTL* and Fair-$\forall$AFMC. The problem of checking $\exists$-simulation, which characterizes Fair-$\forall$CTL*, is PSPACE complete (with respect to the size of the specification.) Fair simulation, which characterizes Fair-$\forall$AFMC, can be checked in polynomial time (with respect to the state spaces of the two structures.) To the best of our knowledge, the complexity of checking Fair-$\forall$CTL simulations is open, with an upper bound in PSPACE and no matching lower bound.

In Chapter 5, we defined a Sample operator to compare implementations to specifications operating at an abstract time scale. In order for this approach to be practical, we generalized assume-guarantee reasoning to handle witness modules and refinement constraints that transfer information between the two time-scales. In order to carry out the proof obligations generated by the assume-guarantee rule, we enhanced our model checker MOCHA with support for the Sample operator. In Chapter 6, we described a large real-life application of the Sample operator and the generalized assume-guarantee rule—the VGI chip.

An interesting extension, which we have not addressed, is to study refinements where $m$ steps of the implementation correspond to $n$ steps of the specification, for arbitrary $m$ and $n$. Another possible direction is to investigate verifying communication protocols, which are typically "layered." The physical layer typically operates at the bit-level, the data link layer at the level of frames, the network layer at the level of packets, and the transport layer at the level of user-defined messages. The Sample operator and the generalized assume-guarantee rule could provide a framework for verifying such systems.
CHAPTER 7. CONCLUSION

Assume-guarantee proofs require the user to manually come up with abstract constraints. Automating the construction of abstract constraints, at least partially, is an important step in making the assume-guarantee framework of MoCHA more automatic. There have been recent efforts to construct abstract constraints using games [AdAHM99]. Making such techniques scale is an important area for future work.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendix A

VGI - Instruction Set Architecture Specification

In this appendix we give the Reactive Modules description of the ISA model. The model consists of modules: ProgMem, NextProgramCounter, CReg, Regfile, IndexDecode, Illegal, Operand, StallEmpty, InputQueueControl, OutputQueueControl, and Alu. A brief description of the functionality of each module is given in Table 6.1.

-- Program Memory

module ProgMem
  external nextpc: progindex
  interface npc: bv4; ncond: bv2; src1, src2, dest: bv3;
    cin, cout: bv2; fosel: bv3; ccwr: bool;
    opcode: instrtype; l1out: bv2
  private npcarray: bv4array; ncondarray: bv2array;
    src1array, src2array, destarray: bv3array;
    cinarray, coutarray: bv2array; foselarray: bv3array;
    ccwarray: boolarray; opcodearray: instrtypearray;
    l1outarray: bv2array
  atom controls npcarray, ncondarray, src1array, src2array, destarray, cinarray,
    coutarray, foselarray, ccwarray, opcodearray, l1outarray
reads npcarray, ncondarray, src1array, src2array, destarray, cinarray, coutarray, foselarray, ccwarray, opcodearray, l1outarray

update
[] true => forall i npcarray'[i] := npcarray[i];
  forall i ncondarray'[i] := ncondarray[i];
  forall i src1array'[i] := src1array[i];
  forall i src2array'[i] := src2array[i];
  forall i destarray'[i] := destarray[i];
  forall i cinarray'[i] := cinarray[i];
  forall i coutarray'[i] := coutarray[i];
  forall i foselarray'[i] := foselarray[i];
  forall i ccwarray'[i] := ccwarray[i];
  forall i opcodearray'[i] := opcodearray[i];
  forall i l1outarray'[i] := l1outarray[i]
endatom

atom controls npc, ncond, src1, src2, dest, cin, cout, fosel,
  ccwr, opcode, l1out
reads nextpc

awaits npcarray, ncondarray, src1array, src2array, destarray, cinarray,
  coutarray, foselarray, ccwarray, opcodearray, l1outarray
update
[] true => npc' := npcarray'[nextpc]; ncond' := ncondarray'[nextpc];
  src1' := src1array'[nextpc]; src2' := src2array'[nextpc];
  dest' := destarray'[nextpc]; cin' := cinarray'[nextpc];
  cout' := coutarray'[nextpc]; fosel' := foselarray'[nextpc];
  ccwr' := ccwarray'[nextpc]; opcode' := opcodearray'[nextpc];
  l1out' := l1outarray'[nextpc]
endatom
endmodule

---------------------------------------------------------------------
-- Next Program Counter
---------------------------------------------------------------------

module NextProgramCounter
external ncond, ccreg: bv2; npc: bv4; f0bus, f1bus: bv2; stall: bool
interface nextpc: progindex

private branch: bool; temp: progindex
atom controls branch reads ccreg awaits ncond, npc, f0bus, f1bus
update
  [] ncond’ = 0 -> branch’ := npc’[0]
  [] ncond’ = 1 -> branch’ := ccreg[1]
  [] ncond’ = 2 -> branch’ := f0bus’[1]
  [] ncond’ = 3 -> branch’ := f1bus’[1]
endatom

atom controls temp awaits npc
update
  [] npc’ = 0 | npc’ = 1 -> temp’ := 0
  [] npc’ = 2 | npc’ = 3 -> temp’ := 2
  [] npc’ = 4 | npc’ = 5 -> temp’ := 4
  [] npc’ = 6 | npc’ = 7 -> temp’ := 6
  [] npc’ = 8 | npc’ = 9 -> temp’ := 8
  [] npc’ = 10 | npc’ = 11 -> temp’ := 10
  [] npc’ = 12 | npc’ = 13 -> temp’ := 12
  [] npc’ = 14 | npc’ = 15 -> temp’ := 14
endatom

atom controls nextpc reads nextpc awaits branch, temp, stall
init
  [] true -> nextpc’ := 0
update
  [] stall’ -> nextpc’ := nextpc
  [] stall’ -> nextpc’ := if branch’ then temp’ + 1 else temp’ fi
endatom

endmodule
module CCreg

external ccwr: bool; cout: bv2; abus, bbus, alout: bv16;
  regfile: regfiletype; stall: bool

interface ccreg: bv2

private ra: bv16

atom ra controls ra awaits regfile

init update
[] true -> ra' := regfile'[0]
endatom

atom ccreg controls ccreg reads ccreg awaits ccwr, abus, bbus, alout, cout, ra, stall

update
[] stall' & (ccwr' & (cout' = 0)) ->
  ccreg'[1] := (abus'[$REGWIDTH-1] & bbus'[$REGWIDTH-1]) |
  ((abus'[$REGWIDTH-1] | bbus'[$REGWIDTH-1]) & alout'[$REGWIDTH-1])
[] stall' & (ccwr' & (cout' = 1)) -> ccreg'[1] :=
  (abus'[$REGWIDTH-1] & bbus'[$REGWIDTH-1] & alout'[$REGWIDTH-1]) |
  (abus'[$REGWIDTH-1] & bbus'[$REGWIDTH-1] & alout'[$REGWIDTH-1])
[] stall' & (ccwr' & (cout' = 2)) -> ccreg'[1] := ra'[REGWIDTH-1]
[] stall' & (ccwr' & (cout' = 3)) ->
  ccreg'[1] := if (alout' = 0) then true else false fi
[] stall' -> ccreg' := ccreg
endatom
endmodule
module Regfile
external aluout, d0bus, d1bus: bv16; dest_dec: regfileindex; stall: bool
interface regfile: regfiletype
external reg1, reg2, reg3, reg4, reg5, reg6: bv16

atom controls regfile reads regfile awaits dest_dec, aluout, d0bus, d1bus, reg1, reg2, reg3, reg4, reg5, reg6, stall
init
[] true -> regfile'[0] := d0bus'; regfile'[1] := d1bus';
update
[] stall' ->
    forall i regfile'[i] := if i = 0 then d0bus' else if (i = 1) then d1bus'
    else if (i = dest_dec')
    then aluout' else regfile[i] fi fi fi
endatom
endmodule

-- Src Decoder

module IndexDecode
external x: bv3
interface x_dec: regfileindex

atom controls x_dec awaits x
init update
[] x' = 0 -> x_dec' := 0
[] x' = 1 -> x_dec' := 1
[] x' = 2 -> x_dec' := 2
[] x' = 3 -> x_dec' := 3
[] x' = 4 -> x_dec' := 4
X' = 5 -> x_dec' := 5  
X' = 6 -> x_dec' := 6  
X' = 7 -> x_dec' := 7  
endatom  
endmodule  

Src1Decode := IndexDecode[x, x_dec := src1, src1_dec]  
Src2Decode := IndexDecode[x, x_dec := src2, src2_dec]  
DestDecode := IndexDecode[x, x_dec := dest, dest_dec]  

--- Illegal Instruction Check  
---  
module Illegal  
external src1_dec, src2_dec, dest_dec: regfileindex  
interface illegal: bool  
atom controls illegal reads illegal awaits src1_dec, src2_dec, dest_dec  
update  
[] illegal -> illegal' := (src1_dec' > 0 & src1_dec' = src2_dec') |  
  (src1_dec' = 2 & src2_dec' = 3) | (src1_dec' = 3 & src2_dec' = 2) |  
  (src1_dec' = 4 & src2_dec' = 5) | (src1_dec' = 5 & src2_dec' = 4) |  
  (src1_dec' = 6 & src2_dec' = 7) | (src1_dec' = 7 & src2_dec' = 6) |  
  (src1_dec' = 1 | src2_dec' = 1 | dest_dec' = 1)  
[] illegal -> illegal' := illegal  
endatom  
endmodule  

--- Operands  
---  
module Operand  
external regfile: regfiletype; src1_dec, src2_dec: regfileindex
interface abus, bbus: bv16
atom controls abus, bbus reads regfile awaits src1_dec, src2_dec
update
[] true -> abus' := regfile[src1_dec']; bbus' := regfile[src2_dec']
endatom
endmodule

------------------------------------------------------------------
-- Stall Signals
------------------------------------------------------------------
module StallEmpty
external c0_empty_stnn, c1_empty_stnn: bool
interface stall_empty: bool
atom controls stall_empty awaits c0_empty_stnn, c1_empty_stnn
init
[] true -> stall_empty' := false
update
[] true -> stall_empty' := c0_empty_stnn' | c1_empty_stnn'
endatom
endmodule

module ISA_Stall
external stall_pipe, stall_empty: bool
interface stall: bool
atom controls stall awaits stall_pipe, stall_empty
init update
[] true -> stall' := stall_pipe' | stall_empty'
endatom
endmodule

------------------------------------------------------------------
-- Input Queue Control Signals
------------------------------------------------------------------
module InputQueueControl
  external cin, ncond: bv2; fosel: bv3
  interface c0.save, c1.save, c0.read, c1.read: bool

atom controls c0.read awaits cin, fosel, ncond
init
[] true -> c0.read' := false
update
[] cin' = 2 -> c0.read' := true
[] fosel' = 2 -> c0.read' := true
[] ncond' = 2 -> c0.read' := true
[] default -> c0.read' := false
endatom
atom controls c1.read awaits cin, fosel, ncond
init
[] true -> c1.read' := false
update
[] cin' = 3 -> c1.read' := true
[] fosel' = 3 -> c1.read' := true
[] ncond' = 3 -> c1.read' := true
[] default -> c1.read' := false
endatom
atom controls c0.save, c1.save
init update
[] true -> c0.save' := false; c1.save' := false
endatom
endmodule

--- Output Queue Control Signals

module OutputQueueControl
  external l1out, ccreg: bv2; fosel: bv3; c0bus, c1bus: bv2;
aluout: bv16; stall_pipe, stall_empty: bool
interface c_send_req, d0_send_req, d1_send_req, d2_send_req: bool;
cobus: bv2; d0bus, dibus, d2bus: bv16
atom controls c_send_req awaits fosel, stall_empty, stall_pipe
init
[] true -> c_send_req’ := false
update
[] true -> c_send_req’ := ( (fosel’ = 0) & stall_empty’ & stall_pipe’)
endatom
atom controls cobus awaits fosel, c0bus, c1bus, ccreg
update
[] fosel’ = 1 -> cobus’ := ccreg’
[] fosel’ = 2 -> cobus’ := c0bus’
[] fosel’ = 3 -> cobus’ := c1bus’
[] default -> cobus’[0] := fosel’[0]; cobus’[1] := fosel’[1]
endatom

atom controls d0_send_req, d1_send_req, d2_send_req
    awaits l1out, stall_empty, stall_pipe
init
[] true -> d0_send_req’ := false; d1_send_req’ := false; d2_send_req’ := false
update
[] true -> d0_send_req’ := (l1out’[0] & stall_empty’ & stall_pipe’);
    d1_send_req’ := (l1out’[1] & stall_empty’ & stall_pipe’);
    d2_send_req’ := false
endatom

atom controls d0bus, d1bus, d2bus awaits aluout
update
[] true -> d0bus’ := aluout’; d1bus’ := aluout’; d2bus’ := aluout’
endatom
endmodule
module Alu

external cin, ccreg, c0bus, c1bus: bv2; opcode: instrtype; abus, bbus: bv16

interface aluout: bv16

private carry_in: bool

atom controls carry_in reads ccreg awaits c0bus, c1bus, cin

update
[] cin’ = 0 -> carry_in’ := false
[] cin’ = 1 -> carry_in’ := ccreg[1]
[] cin’ = 2 -> carry_in’ := c0bus[1]
[] cin’ = 3 -> carry_in’ := c1bus[1]

endatom

atom controls aluout awaits opcode, abus, bbus, carry_in

update
[] opcode’ = ADD ->
    aluout’ := if carry_in’ then abus’ + bbus’ + 1 else abus’ + bbus’ fi
[] opcode’ = SUB ->
    aluout’ := if carry_in’ then abus’ + bbus’ else abus’ + bbus’ + 1 fi
[] opcode’ = AND ->
    aluout’ := abus’ & bbus’
[] opcode’ = NOT ->
    aluout’ := abus’

endatom

endmodule
Appendix B

VGI - Queue Specification

In this appendix, we give the Reactive Modules description of the queue specification.

-- Data Queue Specification

module DataQueueSpec
  external send_req, read, save, stall_pipe, stall_empty: bool; data_in: bv16
  interface send.ackmn, empty_stnn: bool; data_out: bv16;
    empty, full: bool; head, tail: queueindex
private q: dataqueue
private stall: bool
atom controls stall awaits stall_empty, stall_pipe
init update
[] true -> stall' := stall_empty' | stall_pipe'
endatom
atom controls empty reads tail, empty
  awaits tail, head, read, save, stall, empty_stnn
init
[] true -> empty' := true
update
[] (tail' = tail) -> empty' := false
[] (tail' = tail) & save' & read' & stall'
& empty_stnn' & (head' = tail' + 1) -> empty' := true
endatom
atom controls full awaits head, tail, empty
init update
[] true -> full' := (head' = tail' + 1) & empty'
endatom
atom controls send_acknn reads full awaits send_req
init
[] true -> send_acknn' := true
update
[] full -> send_acknn' := true
[] full -> send_acknn' := send_req'
endatom
atom controls empty_stnn reads empty awaits read
init
[] true -> empty_stnn' := read'
update
[] true -> empty_stnn' := read' | empty
[] true -> empty_stnn' := false
endatom
atom QueueWrite controls tail, q reads tail, q awaits send_acknn, data_in
init
[] true -> tail' := 3; forall i q'[i] := 0
update
[] send_acknn' -> tail' := tail + 1;
forall i q'[i] := if (i = tail + 1) then data_in' else q[i] fi
endatom
atom QueueRead controls head, data_out
    reads head, q, empty
    awaits empty_stnn, save, stall, read
init
[] true -> head' := 0
update
[] stall' & empty_stnn' & read' & empty ->
head' := if save' then head else head + 1 fi;
data_out' := q[head]
endatom
endmodule

-- Control Queue Specification

module CtrlQueueSpec
external send_req, read, save, stall_pipe, stall_empty: bool; data_in: bv2
interface send_acknn, empty_stnn, full, empty: bool;
data_out: bv2; head, tail: queueindex
private q: ctrlqueuetype
private stall: bool
atom controls stall awaits stall_empty, stall_pipe
init update
[] true -> stall' := stall_empty' | stall_pipe'
endatom
atom controls empty reads tail, empty
awaits tail, head, read, save, stall, empty_stnn
init
[] true -> empty' := true
update
[] (tail' = tail) -> empty' := false
[] (tail' = tail) & save' & read' & stall'
 & empty_stnn' & (head' = tail' + 1) -> empty' := true
endatom
atom controls full awaits head, tail, empty
init update
[] true -> full' := (head' = tail' + 1) & empty'
endatom
atom controls send_acknn reads full awaits send_req
init
APPENDIX B. VGI - QUEUE SPECIFICATION

[] true -> send_acknn' := true
update
[] full -> send_acknn' := true
[] full -> send_acknn' := send_req'
endatom

atom controls empty_stnn reads empty awaits read
init
[] true -> empty_stnn' := read'
update
[] true -> empty_stnn' := read' | empty
[] true -> empty_stnn' := false
endatom

atom QueueWrite controls tail, q reads tail, q awaits send_acknn, data_in
init
[] true -> tail' := 3; forall i q'[i] := 0
update
[] send_acknn' -> tail' := tail + 1;
   forall i q'[i] := if (i = tail + 1) then data_in' else q[i] fi
endatom

atom QueueRead controls head, data_out
   reads head, q, empty
   awaits empty_stnn, save, stall, read
init
[] true -> head' := 0
update
[] stall' & empty_stnn' & read' & empty ->
   head' := if save' then head else head + 1 fi;
   data_out' := q[head]
endatom

endmodule
d0.send_acknn, d0_empty_stnn, d0_data_out, d0_empty, d0_full, d0_head, d0_tail]
QueueSpec_d1 := DataQueueSpec[send_req, read, save, stall_pipe, stall_empty,
data_in, send_acknn, empty_stnn, data_out, empty, full, head, tail :=
d1_send_req, d1_read, d1_save, d1_stall_pipe, d1_stall_empty, d1_bus,
d1_send_acknn, d1_empty_stnn, d1_data_out, d1_empty, d1_full, d1_head, d1_tail]
QueueSpec_c := CtrlQueueSpec[send_req, read, save, stall_pipe, stall_empty,
data_in, send_acknn, empty_stnn, data_out, empty, full, head, tail :=
c_send_req, c_read, c_save, c_stall_pipe, c_stall_empty, cobus, c_send_acknn,
c_empty_stnn, c_data_out, c_empty, c_full, c_head, c_tail]
Appendix C

VGI - Abstract Constraints and Auxiliary Variables

In this appendix we give the Reactive Modules description of some of the abstract constraints and auxiliary variables used in the proof.

-- Abstract constraint for sentnn.p

module QueueSize
  external send, read, save, empty_stnn, stall_empty, stall_pipe: bool
  interface sentnn.p: bool
  private num: queueindex; incr, decr: bool

  atom controls decr awaits read, save, empty_stnn, stall_pipe, stall_empty
    init
    [] true -> decr’ := false
    update
    [] true -> decr’ := read’ & ~save’ & empty_stnn’ &
      ~stall.pipe’ & ~stall.empty’
  endatom
  atom controls incr reads num awaits decr, send
    init
APPENDIX C. VGI - ABSTRACT CONSTRAINTS AND AUXILIARY VARIABLES

[] true -> incr' := false
update
[] decr' -> incr' := send' & (num <= 2)
[] ~decr' -> incr' := send' & (num < 2)
endatom
atom controls num reads num awaits decr, incr
init
[] true -> num' := 0
update
[] incr' & ~decr' -> num' := num + 1
[] ~incr' & decr' -> num' := num - 1
endatom
atom controls sendn_p awaits send, incr
init
[] true -> sendn_p' := true
update
[] send' -> sendn_p' := ~incr'
[] ~send' -> sendn_p' := true
endatom
xendmodule

QueueSize_d0 := QueueSize[sendn_p, read, save, empty_stnn, stall_pipe, stall_empty := sendn_p.0, send_0, d0_read, d0_save, d0_empty_stnn, d0_stall.pipe, d0_stall.empty]
QueueSize_d1 := QueueSize[sendn_p, send, read, save, empty_stnn, stall_pipe, stall_empty := sendn_p.1, send_1, d1_read, d1_save, d1_empty_stnn, d1_stall.pipe, d1_stall.empty]
QueueSize_d2 := QueueSize[sendn_p, send, read, save, empty_stnn, stall_pipe, stall_empty := sendn_p.2, send_2, d2_read, d2_save, d2_empty_stnn, d2_stall.pipe, d2_stall.empty]
QueueSize_c := QueueSize[sendn_p, send, read, save, empty_stnn, stall_pipe, stall_empty := sendn_p.3, send_3, c_read, c_save, c_empty_stnn, c_stall.pipe, c_stall_empty]
APPENDIX C. VGI - ABSTRACT CONSTRAINTS AND AUXILIARY VARIABLES

-- Auxiliary variable to be asserted when EX phase sends data to lout
____________________________________________________
module ExecutePhaseSend
    external stall, send.req: bool
    interface ex_send: bool
    atom controls ex_send reads ex_send, send.req, stall
    init
    [] true -> ex_send’ := false
    update
    [] ~stall -> ex_send’ := send_req
    [] stall -> ex_send’ := ex_send
endatom
endmodule

ExecutePhaseSend_d0 := ExecutePhaseSend[send_req, ex_send := d0_send_req, d0_ex_send]
ExecutePhaseSend_d1 := ExecutePhaseSend[send_req, ex_send := d1_send_req, d1_ex_send]
ExecutePhaseSend_d2 := ExecutePhaseSend[send_req, ex_send := d2_send_req, d2_ex_send]
ExecutePhaseSend_c := ExecutePhaseSend[send_req, ex_send := c_send_req, c_ex_send]

____________________________________________________
-- Env module for constraining inputs between sampling instants
____________________________________________________
module Env_sentnn
    interface sentnn: bool
    external clk, sentnn.p: bool
    atom controls sentnn reads sentnn.p awaits clk
    update
    [] clk’ -> sentnn’ := sentnn_p
    [] ~clk’ -> sentnn’ := nondet
endatom
endmodule
Env_sentnn_0 := Env_sentnn[ sentnn, sentnn_p := sentnn_0, sentnn_p_0 ]
Env_sentnn_1 := Env_sentnn[ sentnn, sentnn_p := sentnn_1, sentnn_p_1 ]
Env_sentnn_2 := Env_sentnn[ sentnn, sentnn_p := sentnn_2, sentnn_p_2 ]
Env_sentnn_3 := Env_sentnn[ sentnn, sentnn_p := sentnn_3, sentnn_p_3 ]

-- Delayed version of sentnn

module Sentnn_d_ref
  external sentnn_p: bool
  interface sentnn_d: bool
  atom controls sentnn_d reads sentnn_p
  update
    [] true -> sentnn_d' := sentnn_p
  endatom
endmodule

Sentnn_d_ref_0 := Sentnn_d_ref[ sentnn_p, sentnn_d := sentnn_p_0, sentnn_d_0 ]
Sentnn_d_ref_1 := Sentnn_d_ref[ sentnn_p, sentnn_d := sentnn_p_1, sentnn_d_1 ]
Sentnn_d_ref_2 := Sentnn_d_ref[ sentnn_p, sentnn_d := sentnn_p_2, sentnn_d_2 ]
Sentnn_d_ref_3 := Sentnn_d_ref[ sentnn_p, sentnn_d := sentnn_p_3, sentnn_d_3 ]

-- Abstract constraint for mir1

module imem_ref external npc: bv4; ncond: bv2; src1, src2, dest: bv3;
  cin, cout: bv2; f0sel: bv3; ccwr: bool;
  opcode: instrtype; l1out: bv2;
  stall_empty, stall_pipe: bool
interface mir1: bv40
atom controls mir1
reads mir1, stall_empty, stall_pipe
awaits npc, ncond, src1, src2, dest, cin, cout, fosel, ccwr,
    opcode, liout

init
[] true -> mir1' := 0

update
[] (stall_empty | stall_pipe) -> mir1' := mir1
[] (stall_empty | stall_pipe) ->
    mir1'[39] := npc'[3]; mir1'[38] := npc'[2];
    mir1'[37] := npc'[1]; mir1'[36] := npc'[0];
    mir1'[35] := true;
    mir1'[34] := ncond'[1]; mir1'[33] := ncond'[0];
    mir1'[32] := src1'[2]; mir1'[31] := src1'[1];
    mir1'[30] := src1'[0];
    mir1'[29] := src2'[2]; mir1'[28] := src2'[1];
    mir1'[27] := src2'[0];
    mir1'[26] := false; mir1'[25] := false;
    mir1'[24] := false;
    mir1'[23] := dest'[2]; mir1'[22] := dest'[1];
    mir1'[21] := dest'[0];
    mir1'[20] := cin'[1]; mir1'[19] := cin'[0];
    mir1'[18] := cout'[1]; mir1'[17] := cout'[0];
    mir1'[16] := fosel'[2]; mir1'[15] := fosel'[1];
    mir1'[14] := fosel'[0];
    mir1'[13] := ccwr';
    mir1'[12] := false; mir1'[11] := false;
    mir1'[10] := (opcode' = ADD);
    mir1'[9] := (opcode' = ADD);
    mir1'[8] := (opcode' = NOT) | (opcode' = SUB);
    mir1'[7] := (opcode' = NOT) | (opcode' = SUB);
    mir1'[6] := (opcode' = NOT) | (opcode' = SUB);
    mir1'[5] := (opcode' = AND) | (opcode' = NOT);
    mir1'[4] := false; mir1'[3] := false;
    mir1'[2] := false; mir1'[1] := liout'[1];
APPENDIX C. VGI - ABSTRACT CONSTRAINTS AND AUXILIARY VARIABLES

```
mir1'[0] := l1out'[0]
endatom
endmodule

-- Abstract constraint for mir2reg

module mir2reg_ref
    external mir1: bv40; stall_pipe, stall_empty: bool
    interface mir2: bv24
    atom mir2 controls mir2
    reads mir2, mir1, stall_pipe, stall_empty
    init
        [] true -> mir2' := 0
    update
        [] stall_pipe | stall_empty -> mir2' := mir2
        [] (stall_pipe | stall_empty) ->
            mir2'[0] := mir1[0]; mir2'[1] := mir1[1];
            mir2'[6] := mir1[6]; mir2'[7] := mir1[7];
            mir2'[8] := mir1[8]; mir2'[9] := mir1[9];
            mir2'[14] := mir1[14]; mir2'[15] := mir1[15];
            mir2'[16] := mir1[16]; mir2'[17] := mir1[17];
            mir2'[18] := mir1[18]; mir2'[19] := mir1[19];
            mir2'[20] := mir1[20]; mir2'[21] := mir1[21];
            mir2'[22] := mir1[22]; mir2'[23] := mir1[23]
        endatom
endmodule
```
-- Abstract constraint for mir1 and mir2 between sampling instances

module Env_mir1
    interface mir1: bv40
    external clk: bool
    atom controls mir1 reads mir1 awaits clk
    update
        [] clk' -> mir1' := mir1
        [] clk' -> mir1' := nondet
    endatom
endmodule

module Env_mir2
    interface mir2: bv24
    external clk: bool
    atom controls mir2 reads mir2 awaits clk
    update
        [] clk' -> mir2' := mir2
        [] clk' -> mir2' := nondet
    endatom
endmodule

-- Abstract constraint for program memory

module program_ref
    external stall_empty, stall_pipe: bool
    interface npc: bv4; ncond: bv2; src1, src2, dest: bv3;
        cin, cout: bv2; fosel: bv3; ccwr: bool;
        opcode: instrtype; l1out: bv2
    atom controls npc, ncond, src1, src2, dest, cin, cout,
        fosel, ccwr, opcode, l1out
    reads npc, ncond, src1, src2, dest, cin, cout, fosel,
APPENDIX C. VGI - ABSTRACT CONSTRAINTS AND AUXILIARY VARIABLES

ccwr, opcode, llout, stall_empty, stall_pipe

update

[] (stall.pipe | stall.empty) -> npc' := npc; ncond' := ncond; src1' := src1;
   src2' := src2; dest' := dest; cin' := cin;
   cout' := cout; fosel' := fosel; ccwr' := ccwr;
   opcode' := opcode; liout' := liout

[] (stall.pipe | stall.empty) -> npc' := nondet; ncond' := nondet; src1' := nondet;
   src2' := nondet; dest' := nondet; cin' := nondet;
   cout' := nondet; fosel' := nondet; ccwr' := nondet;
   opcode' := nondet; liout' := nondet

endatom

endmodule