Evidence-Based Verification

A Dissertation Presented

by

Li Tan

to

Computer Science Department

in partial fulfillment of the requirements

for the degree of

doctor of philosophy

in

computer science

State University of New York

at Stony Brook

April 2002

State University of New York

at Stony Brook

The Graduate School

<u>Li Tan</u>

We, the dissertation committee for the above candidate for

the degree of Doctor of Philosophy,

hereby recommend acceptance of this dissertation.

Professor Rance W. Cleaveland, Advisor
Computer Science Department

Professor Scott A. Smolka, Chairman of Defense
Computer Science Department

Professor I. V. Ramakrishnan
Computer Science Department

Dr. Kedar Namjoshi
Bell Labs

This dissertation is accepted by the Graduate School.

Graduate School

Abstract of Dissertation

Evidence-Based Verification

by

Li Tan

State University of New York at Stony Brook

Advisor: Rance W. Cleaveland

2002

The ability of generating diagnostic information and performing other "post-verification" analyses is an important feature of verification tools. Traditionally these analyses rely on the proof computed during verification and hence are tightly geared to the infrastructures of these checkers. We present a framework for certifying verification results and efficiently generating diagnostic information in the domain of equivalence checking, preorder checking, and model checking of finite-state system. The central idea is to use a generic data structure called "abstract proof structure"(APS) to abstractly encode the proof structure by which a verification engine (checker) reaches its result. APS carries checker-independent evidence for justifying verification result and provides the basis for a variety of "post-verification" analyses.

In this framework, APS serves as interface data structure between checkers and analysis routines. Checkers are enhanced with bookkeeping code to produce APSs as their results instead of simple "yes/no" answer. A wide range of existing checkers can be modified to produce APS without compromising their time and space complexities.

We then show how APSs may be used to perform an array of "post-verification" analyses, including certification of verification result and generation of different forms of diagnostic information.

An immediate usage of APS is to certify verification result by checking the internal consistency of APS submitted by checker. We provide efficient

algorithms to preform such checking.

The primary goal of this framework is to improve the efficiency and flexibility of diagnostic generation routines. We show that many diagnostic analyses can be preformed using APSs , including some traditional (e.g., counterexamples [7] in model checking and HML formulas [22] for bisimulation) and novel(e.g., vacuity detection [2]) diagnostic schema. We also show how winning strategies for property-checking games [43] can be built from APS. The analysis routines based on APS enjoy independence from checkers; hence they can be easily modified and migrated from one checker to another. In essence APSs help standardize the output of checkers, and analysis routines therefore can be created and executed independently, which has other interesting applications such as a client-server architectures for verification.

谨以此文献给我的母亲, 方雪芬女士, 和我的外婆, 陈聪月女士
她们的关爱, 梦想, 和勇气激励着我, 永不停息.

To my mom, Ms. Xuefen Fang, and my grandma, Ms. Congyue Chen,
for their love, dreams, and courage inspire me, never stop.

Contents

List of Figures

Acknowledgements

First I want to thank my advisor, Prof. Rance Cleaveland, for his support during my doctoral research. I am one of lucky few who has an advisor both as a good mentor and as a great friend. As a mentor, he introduced me to the field of automatic verification. Under his advising, I enjoy the freedom of exploring those subjects of my special interests, while I can always rely on his expertise as my last resort. I enjoy discussions with him, whether on academic matter or on non-academic matter. Many ideas in this dissertation first appeared on the whiteboard of Rance's office (I should apologize for those pens laying dry on his pen-holder). I also enjoy and admire his rigorous while still subtle and elegant writing style.

I would also like to thank the other members of my dissertation committee, Prof. Scott Smolka, Prof. I.V. Ramakrishnan, and Dr. Kedar Namjoshi. They contributed lots of valuable advices on my research and dissertation writing. As the matter of fact, the second part of this dissertation was originated from one of my talks in LMC lab invited by Prof. Smolka. Prof. I. V. Ramakrishnan offered his expertise in formulating and publishing concepts and ideas. Prof. C. R. Ramakrishnan impressed me with his quick insight and broadness of his knowledge. They also generously extend their help with personal assistant on my career development.

I am glad to have so many wonderful souls as my friends in Stony Brook. To name few: Yi Xiao, Yi Yang, Sengupta Bikram, Leena Unnikrishna, Yang

Yang, Jun Zhang, Lan Huang, Basu Samik, Ping Yang, Michiel Van Osch, and so many other. I also want to thank Dawns Family, Robert, Catherine, and Jordan. They generously share with my family their jubilant happiness as Americans and appreciation of life. My life at Stony Brook would have been unbearable without all these friends.

This degree is not just a personal accomplishment. It is the result of great expectation and years' effort of my family. I want to thank my wife for her consistent love, support, and believe which comfort me when I am feeling down. Her lovely and spacious apartment at Lowell, Massachusetts provides enough space for me to grow ideas freely. I especially want to thank my mom Xuefen Fang, my grandpa Shaopu Fang, my grandma Congyue Chen, my grandpa Dr. Shi-Jie Tan, my aunt Wenying Lin, my aunt Rufen Fang, and many others who assisted my schooling. Without their love and craver for higher education which changed me forever, you would be holding a pound of blank pages right now.

Chapter 1

Introduction

"And ye shall know the truth, and the truth shall make you free." **-John 8:32**

Automatic verification refers to an array of techniques in which a verification algorithm (checker) checks whether or not a transition system satisfies some properties in a fully automatic fashion (cf. [18, 9, 36]). A verification routine can be viewed as a decision procedure which returns "yes/no" answer for verification problems in question. Automatic verification has been successfully used in a wide range of applications, from checking consistency of communication protocols to detecting security loopholes in mobile code. There are at least a dozen verification tools available for the public [24, 13].

Many verification tools won't stop at "yes/no" answer. They usually perform some addition analyses. For example, one such analysis is to provide users diagnostic information to understand verification result and locate potential errors. For users, such information is at least as important as simply knowing the result. Fundamentally diagnostic information is about understanding the correctness of result. To obtain such understanding, a diagnostic routine may undertake its own reasoning, or re-use the knowledge already collected by checkers during verification. For efficiency, almost all diagnostic routines chose the latter. Consequently they are tightly integrated with the internal data structures of checkers. Such connection with checkers may not be a good news for implementers: First, implementers are required to have good knowledge on the infrastructures of checkers. Second, creating a new diagnostic routine or modifying an existing one often requires adjustment of data structures of checkers. Finally, the diagnostic routine built on one checker will not in general work on a different checker. Considering that a verification tool may implement more than one checker, each of which is optimized for certain type of problems, this means substantial extra work to equip each checker with a diagnostic routine.

The field study also shows the importance of measuring qualify of verification process. An immediate concern is, of course, whether the verification result is correct. Experience with the development of verification tools show the design and implementation errors may prevent a verifier from functioning correctly, for example, [31] points out a design error in the algorithm proposed by [53]. Moreover, even the verification process is correct and the result is positive, potential errors may still exist in design systems because the part of specification (cf. [2]) or state space (cf. [28]) is rendered as "don't care" during verification.

The motivation for this work is to ease difficulties in designing and implementing analysis routines and to retain the information during verification for further measuring the quality of verification process. The idea is to encode in a uniform data structure the evidence collected during verification process. This uniformly encoded evidence, referred as *abstract proof structures (APS)* , can be used as a basis for many "post-verification" analyses, including generating many forms of diagnosis information and checking the consistency of checkers (Is APS internal consistent? Can checkers be trusted?). Once generated, these

APSs are independent of the checkers that produced them. therefore, analysis routines built on APSs are insensitive to the checkers being used.

In designing APSs, we wish to satisfy the following requirements,

1. APSs carry the enough information for constructing some most popular diagnostic approaches.

2. APSs should be extractable from a wide range of checkers. Previous works [39, 34] on this subject have strict limitations on which model checkers they can be applied to.

3. Modifying model checkers to produce APSs should not affect its original complexities. These evidences should be produced without compromising overall performance.

4. The internal consistency of APSs should be efficiently checkable. We want to be able to use APSs as a basis to double-checking verification results. For this to be practical, we need to be able to check the "internal consistency" of APSs in time that doesn't asymptotically exceed that needed by the checkers used to build the APSs.

The research is conducted in the domain of three major verification approaches. For equivalence checking, APS is an abstract version of partition refinement tree [47]; for refinement checking, it refers to an abstract version of Kernel-Auxiliary partition refinement tree [48]; for model checking, it turns out to be a *support set* [49], a characterized solution to Boolean equation systems. We will show how our choices of APSs in these three approaches meet above four requirements.

One of usages of APSs is to build diagnostic information. There are many different form of different diagnostic schema which have been studied [7, 2, 43]. Which of them are better is an open question. The answer to this depends on the applications, user preference, and the efficiency of generation routines. Among many diagnoses which can be performed using APSs, property-checking games have our special interest. Games [43]provide an interactive way to diagnose systems. Users can have freedom to choose the part of proof he want to further investigate. We show how various property games can be built on APSs.

Besides facilitating checker-independent analyzes, the framework may also trigger other interesting applications. For example, we already have formalisms to encode inputs of checkers, including dozens of specification languages for writing transition systems, and many temporal logics for encoding temporal properties. The introduction of APS make it possible to standardize the outputs of checkers. We would be able to eventually separate checkers from user interfaces and "post-verification" analyzes. This implies a possible client-server architecture for verification, in which checkers run on high-end computing server, while user interfaces/ "post-verification" analyzes can be done on low-cost client.

The rest of this dissertation is organized by verification approaches to which we want to apply our ideas. In the domain of each approach, we start with some background knowledge and notations, then introduce the definition of APS . We will show how very different checkers nevertheless produce these APSs , and utilize APSs to perform *post-verification* analyzes. These analyses typically include certifying results by checking validity of companying APSs , building games, and generating some interesting diagnostic information on APSs . We will finish our discuss with the conclusion and future directions.

Part I

Evidence-Based Model Checking

Chapter 2

Preliminaries

In model checking [8], we are concerned with whether or not a system possesses temporal property encoded in some temporal logic. We consider only explicit state model checking, in which model checker works explicitly on states and transitions, in comparison to symbolic model checking, in which checkers handle sets of states and transitions represented by Ordered Boolean Decision Diagram (OBDD) [35]. In what follows we fix a set $\mathcal{A}$ of *atomic propositions.*

2.1 Kripke Structures

We use Kripke structures to model systems.

Definition 2.1.1 *A Kripke structure is a tuple $\langle S, s_I, \rightarrow, V \rangle$, where S is the set of states, $s_I \in S$ is the start state, $\rightarrow \subseteq S \times S$ is the transition relation and $V : \mathcal{A} \rightarrow 2^S$ is the valuation.*

A Kripke structure encodes the operational behavior of a system, with S being the possible set of system states, s_I the initial state, and $\rightarrow$ the (atomic) state transitions. For a given atomic proposition $A \in \mathcal{A}$, $V(A)$ indicates in which states A is true. In what follows we write $s \rightarrow s'$ in lieu of $\langle s, s' \rangle \in \rightarrow$.

Given a Kripke structure $\langle S, s_I, \rightarrow, V\rangle$, a *path* from $s \in S$ is a maximal (hence potentially infinite) sequence $\sigma = s_0 s_1 \ldots$ where $s = s_0$ and $s_i \rightarrow s_{i+1}$ for all $i < |\sigma|$. (Here $|\sigma| = n$ if $\sigma = s_0 s_1 \ldots s_n$ and ∞ if σ is infinite.) We use $\sigma[i]$ for s_i and $\sigma^{(i)}$ for $s_i s_{i+1} \ldots$ if $i \leq |\sigma|$.

2.2 Computational Tree Logic (CTL*)

Formula from CTL* may be used to define properties of Kripke structures. These Formulas are given via the following grammar, where $A \in \mathcal{A}$.

$$\phi ::= A \mid \neg A \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{A}\ \psi \mid \mathsf{E}\ \psi$$
$$\psi ::= \phi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\ \psi \mid \psi\ \mathsf{U}\ \psi \mid \psi\ \mathsf{R}\ \psi$$

We refer to the formulas generated from ϕ as *state formulas* and those from ψ as *path formulas*. The CTL* formulas consist of the state formulas. We call A and E *path quantifiers* and X, U, R *path modalities*. E and R are the dual of A and U, respectively. The above grammar only generates formulas in positive normal form (PNF): negation only appears before atomic propositions. It is easy to see that a more general treatment of negation can be accommodated, since negations can be pushed inside. We do not consider this point further.

The sublogic CTL consists of those CTL* formulas in which every path modality is immediately preceded by a path quantifier.

CTL* formulas are interpreted with respect to a Kripke structure $T = \langle S, s_I, \rightarrow, V\rangle$ where $\rightarrow$ is *total*: for every $s \in S$ there exists $s' \in S$ with $s \rightarrow s'$. Given such a T, the semantics of CTL* formulas is given via a relation $\models_T$ associating states s in T to state formulas and paths σ in T to path formulas and which is defined below.

1. $s \models_T A (\neg A)$ iff $s \in V(A)$ ($s \notin V(A)$).

2. $s \models_T \phi_1 \wedge \phi_2$ ($\phi_1 \vee \phi_2$) iff $s \models_T \phi_1$ and (or) $s \models_T \phi_2$.

3. $s \models_T \mathsf{A}\psi$ ($\mathsf{E}\psi$) iff for every (some) path σ from s, $\sigma \models_T \psi$.

4. $\sigma \models_T \phi$, where ϕ is a state formula, iff $\sigma[0] \models \phi$.

5. $\sigma \models_T \psi_1 \wedge \psi_2$ ($\psi_1 \vee \psi_2$) iff $\sigma \models_T \psi_1$ and (or) $\sigma \models_T \psi_2$.

6. $\sigma \models_T \mathsf{X}\psi$ iff $\sigma^{(1)} \models \phi$.

7. $\sigma \models_T \psi_1 \mathsf{U} \psi_2$ iff for some $i \geq 0$, $\sigma^{(i)} \models \psi_2$ and $\sigma^{(j)} \models \psi_1$ for all $j < i$.

8. $\sigma \models_T \psi_1 \mathsf{R} \psi_2$ iff for all $i \geq 0$ $\sigma^{(i)} \models \psi_2$ or $\sigma^{(j)} \models \psi_1$ some $j < i$.

The release modality R is the dual of the until operator U. Intuitively, $\psi_1 \mathsf{R} \psi_2$ holds of a path if ψ_2 is kept true until "released" from this obligation by the truth of ψ_1.

2.3 The Modal Mu-Calculus and Boolean Equation Systems

The μ-calculus is an extremely expressive temporal logic [27]. It is as expressive as alternating automata on infinite trees, hence subsumes most known specification formalisms, including dynamic logics such as PDL [20] and temporal logics such as LTL and CTL*[16]. We will introduce the μ-calculus syntactically as a special instance of fixpoint equation systems. Translating between μ-calculus equation systems and its most traditional linear format [27] is straightforward.

Since we will use Boolean equation systems to encode model-checking problem, and Boolean equation systems can also been seen as instances of fixpoint equation system, we first give a general account of fixpoint equation systems over complete lattices.

2.3.1 Lattices and Environments

A *complete lattice* is a partially ordered set $\langle Q, \sqsubseteq \rangle$ with the following property: every subset $Q' \subseteq Q$ has a least upper bound $\bigsqcup Q'$ in Q. It can be shown that arbitrary upper bounds $\bigsqcap Q$ also exist and that any complete lattice has a unique least element $\bot$ and maximum element $\top$. In addition, the Tarski-Knaster theorem [52] guarantees the existence of unique least and greatest fixpoints for any monotonic function $f : Q \to Q$. Given monotonic f, we write $\mu f \in Q$ for the least "solution" to $f(x) = x$ and $\nu f \in Q$ for the greatest. These fixpoints are characterized as follows.

$$\mu f \;=\; \bigsqcap\{q \in Q \mid f(q) \sqsubseteq q\} \qquad \nu f \;=\; \bigsqcup\{q \in Q \mid q \sqsubseteq f(q)\}$$

Let $\langle Q, \sqsubseteq \rangle$ be a complete lattice and $\mathcal{X}$ be a finite set of *variables.* Then an *environment* over $\mathcal{X}$ is a function from $\mathcal{X}$ to Q.

We use $Q^{\mathcal{X}}$ to represent the set of all environments over $\mathcal{X}$. Environments constitute a complete lattice under the pointwise extension of $\sqsubseteq$ to $Q^{\mathcal{X}}$: $\theta \sqsubseteq \theta'$ if and only if for all $X \in \mathcal{X}, \theta(X) \sqsubseteq \theta'(X)$.

If $\theta \in Q^{\mathcal{X}}$ and $\mathcal{X}' \subseteq \mathcal{X}$, then $\theta|\mathcal{X}' \in Q^{\mathcal{X}'}$ is defined by $(\theta|\mathcal{X}')(X) = \theta(X)$ for all $X \in \mathcal{X}'$. If $\theta' \in Q^{\mathcal{X}'}$ then $\theta[\theta']$ denotes the environment obtained by *updating* θ by θ':

$$\theta[\theta'] = \begin{cases} \theta'(X) & \text{if } X \in \mathcal{X}' \\ \theta(X) & \text{otherwise} \end{cases}$$

2.3.2 Fixpoint Equation Systems

We now develop a general framework for systems of equations over a complete lattice. Throughout the remainder of this subsection we fix a complete lattice $\langle Q, \sqsubseteq \rangle$ and a finite set $\mathcal{X}$ of variables.

Syntax An *equation block* B is a set of equations $\{X_1 = f_1, \ldots, X_l = f_l\}$, where the f_i are monotonic functions in $Q^{\mathcal{X}} \to Q$, $\{X_1, \ldots X_l\} \subseteq \mathcal{X}$, and the X_i

are distinct. We use $lhs(B) = \{X_1, \ldots, X_l\}$ for the left-hand side variables of B, and $rhs(B, X_i) = f_i$ for the right-hand side for X_i in B, respectively. The f_i are often represented syntactically as expressions involving free occurrences of the variables from $\mathcal{X}$. In this case, we use $vars(f_i)$ to refer to the set of variables occurring freely in f_i, and we define $vars(B) = lhs(B) \cup \bigcup_{i=1}^{l} vars(f_i)$ as the *variables* in equation block B. We refer to the variables in $lhs(B)$ as *bound* and the variables in $vars(B) - lhs(B)$ as *free*.

A *parity block* E has form $\langle \sigma, B \rangle$, where $\sigma \in \{\mu, \nu\}$ is a *parity indicator* and B is an equation block. We lift the notions *lhs*, *rhs*, *vars*, free variable, and bound variable to parity blocks in the straightforward manner.

A *fixpoint equation system* is a nonempty sequence $\mathcal{E} = E_1 \ldots E_m$ of parity blocks whose left-hand sides are pairwise disjoint. If $\mathcal{E}'$ is an equation system and E is a parity block whose left-hand side variables are disjoint from those in $\mathcal{E}'$ then we write $E :: \mathcal{E}'$ for the equation system obtained by adding E to the front of $\mathcal{E}'$. We use $\mathcal{E}^{(k)} = E_k E_{k+1} \cdots$ to refer to the subsequence of $\mathcal{E}$ starting from k-th parity block. Alternatively, we may also write a fixpoint equation system as a sequence of equations decorated with their parities. Consequently a block refers to the maximum subsequence of equations with the same parity. Two equation systems are *syntactically identical* if two have the same sequence of blocks with possible exchange of equations inside each block.

The operations *lhs*, *rhs*, and *vars* are generalized in the straightforward manner. We call $\mathcal{E}$ *closed* if every $X \in vars(\mathcal{E})$ is bound, i.e. an element of $lhs(\mathcal{E})$. We also define $h_{\mathcal{E}}(X) = k$ when $X \in lhs(E_k)$ and refer to $h_{\mathcal{E}}(X)$ as the *depth* of X in $\mathcal{E}$. We write $h(X)$ when $\mathcal{E}$ is clear from context. We say X_i is *shallower*(or *higher*) than X_j if $h(X_i) < h(X_j)$, and *deeper* (or *lower*) if $h(X_i) > h(X_j)$. If X is a left-hand side variable in $\mathcal{E}$ we define $\sigma_{\mathcal{E}}(X)$ to be the parity of the unique parity block E in $\mathcal{E}$ such that $X \in lhs(E)$. We omit reference to $\mathcal{E}$ and write $\sigma(X)$ when $\mathcal{E}$ is clear from context.

In a fixpoint equation system $\mathcal{E}$, we say that X_i syntactically depends on X_j, written as $X_i \lhd X_j$, if $X_j \in \mathit{vars}(\mathit{rhs}(X_i))$. We write $\overset{*}{\lhd}$ for the transitive and reflexive closure of $\lhd$; so $X_i \overset{*}{\lhd} X_j$ if there is a path of syntactic dependencies from X_i to X_j.

Semantics Let $\theta \in Q^{\mathcal{X}}$ be an environment; then a single block $B = \{X_1 = f_1, \ldots, X_l = f_l\}$, where $\mathcal{X}' = \{X_1, \ldots, X_l\}$, may be seen as inducing a function $f_{B,\theta} : Q^{\mathcal{X}'} \to Q^{\mathcal{X}'}$ mapping environments over $\mathcal{X}'$ to environments over $\mathcal{X}'$ as follows.

$$f_{B,\theta}(\theta') = [X_1 \mapsto f_1(\theta[\theta']), \cdots, [X_l \mapsto f_l(\theta[\theta'])]]$$

where $[X_1 \mapsto f_1(\theta[\theta']), \cdots, X_l \mapsto f_l(\theta[\theta'])]$ is an environment which maps X_i to $f_i(\theta[\theta'])$. That is, $f_{B,\theta}(\theta')$ returns an environment over $\mathcal{X}'$ in which each X_i is mapped to the result of evaluating f_i on environment $\theta[\theta']$. Note that in $\theta[\theta']$ θ' "overwrites" the values for the X_i in θ. Consequently, θ may be seen as providing values only for variables that are not in $\mathcal{X}'$. It follows from the monotonicity of the f_j that for any θ, $f_{B,\theta}$ is a monotonic function over $\mathcal{Q}^{\mathcal{X}'}$, and consequently, least and greatest fixpoints, $\mu f_{B,\theta}$ and $\nu f_{B,\theta}$, which are environments over $\mathcal{X}'$, exist. Given $\theta \in Q^{\mathcal{X}}$, we then define the semantics of a parity block in terms of these fixed points:

$$[\![\langle \sigma, B \rangle]\!]\theta = \sigma f_{B,\theta}.$$

So $[\![\langle \sigma, B \rangle]\!]$ maps environments over $\mathcal{X}$ to environments over $\mathit{lhs}(B)$.

Fixpoint equation systems may be interpreted using the account just given for blocks. For an environment $\theta \in Q^{\mathcal{X}}$ and equation system $\mathcal{E}$ with $\mathit{lhs}(\mathcal{E}) = \mathcal{X}_{\mathcal{E}} \subseteq \mathcal{X}$ we define a function $f_{\mathcal{E},\theta} : Q^{\mathcal{X}_{\mathcal{E}}} \to Q^{\mathcal{X}_{\mathcal{E}}}$. We then use an appropriate fixpoint of this function to arrive at the meaning of $\mathcal{E}$. We define $f_{\mathcal{E},\theta}$ by induction on the structure of $\mathcal{E}$. When $\mathcal{E} = \langle \sigma, B \rangle$ (i.e. $\mathcal{E}$ contains a single parity block), then we take $f_{\mathcal{E},\theta} = f_{B,\theta}$. In this case $f_{\mathcal{E},\theta}$ is clearly monotonic, and we define $[\![\mathcal{E}]\!]\theta = [\![\langle \sigma, B \rangle]\!]\theta$. When $\mathcal{E}$ contains two or more blocks, it may be written as $\mathcal{E} = \langle \sigma, B \rangle :: \mathcal{E}'$, where $\mathcal{E}'$ is a fixpoint equation system. In this case $f_{\mathcal{E},\theta}$ is defined as

$$f_{\mathcal{E},\theta}(\theta') = (\llbracket \mathcal{E}' \rrbracket(\theta[\theta']))[f_{B,\theta[\llbracket \mathcal{E}' \rrbracket(\theta[\theta'])]}(\theta' | \mathcal{X}_B)],$$

where $\mathcal{X}_B = lhs(B)$. This expression may be understood via its subexpressions.

$\llbracket \mathcal{E}' \rrbracket(\theta[\theta'])$ is the environment defined by equation system $\mathcal{E}'$ in environment θ updated with bindings contained in θ'. This environment assigns a "fixpoint value" to every left-hand variable in $\mathcal{E}'$.

$f_{B,\theta[\llbracket \mathcal{E}' \rrbracket(\theta[\theta'])]}$ is the function on environments defined by block B and the environment obtained by updating θ with the bindings in $\mathcal{E}'$.

$\theta' | \mathcal{X}_B$ is the subenvironment of θ' obtained by restricting variables to those that appear as left-hand sides in B.

It is easy to show that $f_{\mathcal{E},\theta}(\theta')$ is monotonic over the lattice $Q^{\mathcal{X}_\mathcal{E}}$ and hence has unique least and greatest fixpoints. We then define $\llbracket \mathcal{E} \rrbracket \theta$ as follows.

$$\llbracket \langle \sigma, B \rangle :: \mathcal{E}' \rrbracket \theta = \sigma f_{\langle \sigma, B \rangle :: \mathcal{E}', \theta}$$

If $\mathcal{E}$ is closed then for any θ, θ' we have that $\llbracket \mathcal{E} \rrbracket \theta = \llbracket \mathcal{E} \rrbracket \theta'$. In this case we often omit reference to θ and write $\llbracket \theta \rrbracket$ for this (unique) environment.

We conclude this general treatment of fixpoint equation systems with the definition of *alternation depth*. Here we adopt the convention that $max\emptyset = 0$.

Definition 2.3.1 *Let $\mathcal{E} = E_1 E_2 \cdots E_m$ be an equation system and X a left-hand side variable in $\mathcal{E}$. Then the* alternation depth, *$ad(X)$, of X is given as:*

$$ad(X) = 1 + max\{ad(X') \mid \sigma(X) \neq \sigma(X'), h(X') < h(X), X \overset{*}{\lhd} X' \overset{*}{\lhd} X\}$$

2.3.3 The Modal Mu-Calculus

In this paper we define modal mu-calculus formulas using fixpoint equation systems whose right-hand sides are formulas built as follows, where $A \in \mathcal{A}$ and $\mathcal{X}' \subseteq \mathcal{X}$.

$$f = A \mid \neg A \mid \bigwedge \mathcal{X}' \mid \bigvee \mathcal{X}' \mid \langle\rangle X \mid []X$$

The lattice $\langle Q, \sqsubseteq\rangle$ used to interpret variables is given by fixing a Kripke structure $T = \langle S, s_I, \rightarrow, V\rangle$ and taking $Q = 2^S$ and $\sqsubseteq = \subseteq$. We also adopt the usual semantics of modal formulas given below. Here $\theta \in (2^S)^{\mathcal{X}}$ maps propositional variables to sets of states in T.

$$\begin{array}{rclrcl}
[\![A]\!]_T\theta & = & V(A) & [\![\neg A]\!]_T\theta & = & S - V(A) \\
[\![\bigvee \mathcal{X}']\!]_T\theta & = & \bigcup\{\theta(X) \mid X \in \mathcal{X}'\} & [\![\bigwedge \mathcal{X}']\!]_T\theta & = & \bigcap\{\theta(X) \mid X \in \mathcal{X}'\} \\
[\![\langle\rangle X]\!]_T\theta & = & \{s \mid \exists s \rightarrow s'.\, s' \in \theta(X)\} & [\![[]X]\!]_T\theta & = & \{s \mid \forall s \rightarrow s'.\, s' \in \theta(X)\}
\end{array}$$

For any modal formula f, $[\![f]\!]_T$ is a monotonic function from $(2^S)^{\mathcal{X}}$ to 2^S.

Mu-calculus fixpoint equation systems in essence define a collection of formulas, one for each $X \in \mathit{lhs}(\mathcal{E})$. This notation is clumsy for users, but more user-friendly logics such as CTL, LTL and CTL* may be translated efficiently into it [16, 4].

2.3.4 Boolean Equation Systems

Boolean equation systems are fixpoint equation systems defined over the Boolean lattice $\langle\{\mathbf{0}, \mathbf{1}\}, \sqsubseteq\rangle$, where $\mathbf{0}$ and $\mathbf{1}$ are the boolean values "false" and "true", respectively, and $\mathbf{0} \sqsubset \mathbf{1}$. In this setting environments may be viewed as characteristic functions of subsets of $\mathcal{X}$, so we allow the use of the standard set operators $\cup$, $\cap$, and $-$ on such environments. The right-hand sides of equations are the formulas given by the following, where $\mathcal{X}' \subseteq \mathcal{X}$.

$$f := \bigvee \mathcal{X}' \mid \bigwedge \mathcal{X}'$$

We often write tt for $\bigwedge \emptyset$ and ff for $\bigvee \emptyset$. The definition of $[\![f]\!]\theta$ is standard: $[\![\bigvee \mathcal{X}']\!]\theta = \mathbf{1}$ iff $\mathcal{X}' \cap \theta \neq \emptyset$, and $[\![\bigwedge \mathcal{X}']\!]\theta = 1$ iff $\mathcal{X}' \subseteq \theta$.

Solving Boolean equation systems has particular interest because such a system can encode a model-checking problem by combining the temporal property (in form of μ-calculus equation system) and Kripke structure. Intuitively, this is done by assigning a boolean variable to each state / mu-calculus variable pair; the boolean variable is intended to indicate whether or not the state is in the set of states associated with the mu-calculus variable. Formally, for a μ-calculus equation system $\mathcal{E}$ and a Kripke structure $T = \langle S, s_I, \rightarrow, V\rangle$, a boolean equation system $\mathcal{E}_T$ can be built as below. We introduce a unique variable $\langle s, X_i\rangle \in \mathit{vars}(\mathcal{E}_T)$ for each pair of $s \in S$ and $X_i \in \mathit{vars}(\mathcal{E})$. If $(\sigma_i X_i = f_i)$ is the equation for X_i in $\mathcal{E}$, the equation for $\langle s, X_i\rangle$ is $\sigma_i\langle s, X_i\rangle = F(s, f_i)$, where $F(s, f_i)$ instantiates f_i with respect to s as follows.

$$
\begin{array}{rclrcl}
F(s, A) & = & \begin{cases} \mathbf{tt} & \text{if } s \in V(A) \\ \mathbf{ff} & \text{Otherwise} \end{cases} & \begin{array}{r} F(s, \bigwedge \mathcal{X}_j) \\ F(s, \langle\rangle X_j) \end{array} & \begin{array}{c} = \\ = \end{array} & \begin{array}{l} \bigwedge\{\langle s, X_p\rangle \mid X_p \in \mathcal{X}_j\} \\ \bigvee\{\langle s', X_j\rangle \mid s \rightarrow s'\} \end{array} \\
F(s, \bigvee \mathcal{X}_j) & = & \bigvee\{\langle s, X_p\rangle \mid X_p \in \mathcal{X}_j\} & F(s, []X_j) & = & \bigwedge\{\langle s', X_j\rangle \mid s \rightarrow s'\}
\end{array}
$$

The order of equations in $\mathcal{E}_T$ should conform to the order of their original equations in $\mathcal{E}$. i.e., $h(\langle s, X_i\rangle) < h(\langle s, X_j\rangle)$ if $h(X_i) < h(X_j)$. We refer to the above transforming procedure as Π: $\mathcal{E}_T = \Pi(\mathcal{E}, T)$. Clearly the resulting boolean equation system has alternation depth no greater than the mu-calculus equation system from which it is derived.

When a model-checking problem is encoded as a boolean equation system, the goal is typically to determine the value of a single distinguished variable ("Does the start state satisfy the formula?"). Practically users may only care about the value of a distinguished variable, which value answers model-checking problem in question. In other words, we only need *partial solution* related to this variable. In the rest of this section, we fix the boolean equation system $\mathcal{E} = (\sigma_1 X_1 = f_1) \cdots (\sigma_n X_n = f_n)$. We define $\subseteq_r$ as a decorated operator: it acts as $\subseteq$ if $r = \mathbf{1}$, and as $\supseteq$ if $r = \mathbf{0}$.

An environment θ' is a *partial model* with the support value r iff $\theta' \subseteq_r [\![\mathcal{E}]\!]\theta$. Therefore, variables having the same value as a support value in a partial model have exactly the same value in the model.

Chapter 3

Abstract Proof Structures for Model Checking

3.1 Support Sets

In the context of model checking, we are often interested in the value of a distinguished boolean variable. A support set is intended to record the evidence to support the value of such a variable.

Definition 3.1.1 (Support Set) *Let $\mathcal{E} = E_1 \ldots E_m$ be a closed boolean equation system with $\mathcal{X} = lhs(\mathcal{E})$, let $X \in \mathcal{X}$, and $r \in \{\mathbf{0}, \mathbf{1}\}$. Then a* support set *for r and X is a triple $\Gamma = \langle r, X, \Xi \rangle$, where $\Xi : \mathcal{X} \to 2^{\mathcal{X}}$ is a partial function such that $\Xi(X)$ is defined and the following properties hold for each X_i where $\Xi(X_i)$ is defined ($X_i \underset{\Gamma}{\rightarrow} X_j$ if $(\Xi(X_i))(X_j) = r$).*

I.(Direct Inference) $[\![f_i]\!](\Xi(X_i)) = r$

II.(Inclusion) If $X_i \underset{\Gamma}{\rightarrow} X_j$, then $\Xi(X_j)$ is defined.

III.(Circularity Restriction) If there exists a loop $\rho \equiv X_{p_1} \underset{\Gamma}{\rightarrow} \ldots \underset{\Gamma}{\rightarrow} X_{p_1}$ and X_i is the shallowest variable on ρ, then $(r = \mathbf{1} \Rightarrow \sigma_i = \nu) \wedge (r = \mathbf{0} \Rightarrow$

$\sigma_i = \mu$).

We call r the *support value* and X the *support variable* of support set $\Gamma = \langle r, X, \Xi \rangle$. We refer to a support set $\langle r, X, \Xi \rangle$ as *positive* if $r = \mathbf{1}$, and as *negative* otherwise. A variable X_i is *covered* by Ξ if $\Xi(X_i)$ is defined. Support sets may be understood as follows. Recall that environments over the boolean lattice are isomorphic to sets of variables (i.e. an environment may be seen as a characteristic function). Thus Ξ may be seen as associating a set of variables (i.e., an environment) to each variable X_i on which it is defined. Formally we define the *dependency set* T_i for such X_i as follows.

$$T_i = \begin{cases} \Xi(X_i) & \text{If } r = \mathbf{1} \\ \mathcal{X} - \Xi(X_i) & \text{Otherwise} \end{cases}$$

Intuitively the value of X_i is supported by variables in its dependency set, each of which also has the same value as X_i. The existence of an edge $X_i \underset{\Gamma}{\rightarrow} X_j$ indicates the existence of a dependency of the value of X_i on X_j.

Condition I asserts that the environment associated with X_i should immediately support its value in terms of the equation for X_i. Condition II requires all variables on which X_i depends to be in the domain of Ξ, therefore the values of these variables are also supported by their own dependency sets. The last condition imposes restrictions on cyclic dependencies: the parity of the "shallowest" variable on the cycle must be consistent with r.

Lemma 3.1.2 directly follows from Definition.

Lemma 3.1.2 *If $\Gamma = \langle r, X, \Xi \rangle$ is a support set for $\mathcal{E}$ and X' is covered by Γ, then $\Gamma' = \langle r, X', \Xi \rangle$ is also a support set for $\mathcal{E}$.*

We can reduce a support set in the following sense. A support set $\Gamma = \langle r, X, \Xi \rangle$ is reduced if the following condition hold,

1. For every X' covered by Γ, $X \underset{\Gamma}{\rightarrow}^* X'$ (i.e. X' affects X).

2. If $r = \mathbf{1}$ then for every variable X' whose right-hand side uses $\bigvee$ such that $\Xi(X')$ is defined, $|\Xi(X')| = 1$, and dually for $r = \mathbf{0}$.

Intuitively, a support set is reduced if it contains no extraneous information. A support set $\Gamma = \langle r, X, \Xi \rangle$ can be reduced by the following steps, assuming $r = 1$.

1. For each covered variable X', update $\Xi(X')$ by a least subset ξ of $\Xi(X')$ such that $[\![f_i]\!](\xi) = 1$.

2. Keep in Ξ only those X' such that $X \underset{\Gamma}{\rightarrow}^* X'$.

The case for $r = \mathbf{0}$ is dual. In a reduced support set, a dependency set of a covered variable the right-hand side of which has $\bigvee$ (or, $\bigwedge$ if $r = 0$) contains only one variable.

Lemma 3.1.3 *If* Γ *is a support set for* $\mathcal{E}$ *and* Γ' *is reduced from* Γ*, then* Γ' *is also a support set for* $\mathcal{E}$*.*

On the other hand, we may *canonicalize* a support set by expanding dependency sets. A support set $\Gamma' = \langle r, X, \Xi_{can} \rangle$ is *canonical* if it satisfies the following properties, assuming $r = 1$.

For every X' covered by Ξ, then $\Xi(X'') \subseteq \Xi(X')$ for any X'' such that $X' \underset{\Gamma}{\rightarrow} X''$ and $h(X') < h(X'')$

The case for $r = 0$ is dual. Clearly a support set can be canonicalized by adding to the dependency set of each covered variable the dependency set of variable its value depends on. In a canonical support set, the dependency set of a variable contains all the variables its value depends on.

Lemma 3.1.4 *If* Γ *is a support set for* $\mathcal{E}$ *and* Γ' *is canonical support set for* Γ*, then* Γ' *is also a support set for* $\mathcal{E}$*.*

Proof: It immediately follows from Definition 3.1.1 and the *canonicalizing* procedure described above. ■

We may define an environment $g(\Gamma)$ for $\Gamma = \langle r, X, \Xi \rangle$ as follows.

$$(g(\Gamma))(X_i) = \begin{cases} r & \text{if } \Xi(X_i) \text{ is defined} \\ \bar{r} & \text{otherwise} \end{cases}$$

Lemma 3.1.5 *Given a closed boolean equation system* $\mathcal{E}$ *with* $X \in lhs(\mathcal{E})$ *and a support set* $\Gamma = \langle r, X, \Xi \rangle$*, let* $\theta = g(\Gamma)$ *be the environment induced by* $\langle r, X, \Xi \rangle$*. Then* θ *is a partial model with the support value* r *for* $\mathcal{E}$*.*

Proof: See appendix. ■

Theorem 3.1.6 *Given a closed boolean equation system* $\mathcal{E}$ *with* $X \in lhs(\mathcal{E})$ *and a support set* $\Gamma = \langle r, X, \Xi \rangle$ *for* $\mathcal{E}$*, then* $\Xi(X) = [\![\mathcal{E}]\!](X)$*.*

Proof: By lemma 3.1.5 $\theta = g(\Gamma)$ is a partial model with the support value r for $\mathcal{E}$. We consider the case $r = \mathbf{1}$, while the case where $r = \mathbf{0}$ can be handled similarly. Since X is covered by Ξ, $X \in \theta$ by the definition of g. Since $\theta \subseteq [\![\mathcal{E}]\!]$, $X \in \mathcal{E}(X)$, i.e., $[\![\mathcal{E}]\!](X) = \mathbf{1}$. ■

The next theorem guarantees the existence of support sets.

Theorem 3.1.7 *Given a closed boolean equation system* $\mathcal{E}$ *and a distinguished variable* $X \in lhs(\mathcal{E})$*, let* $r = [\![\mathcal{E}]\!](X)$*. Then there exists a support set for* $\mathcal{E}$ *with support value* r *and support variable* X*.*

Proof: See appendix. ■

In the next chapter, readers will have another chance to see the correctness of Theorem 3.1.7 by observing how a local algorithm LAFP can be modified to construct a desired support set for input closed boolean equation system.

Since both LAFP and the modification works on any closed boolean equation system, a support set desired by theorem 3.1.7 will exist for any closed boolean equation system.

3.2 Support Sets for Temporal Logics

The previous definition introduces support sets in the context of boolean equation systems. At their lowest level model checkers may be seen to manipulate such equation systems. However, conveying support-set-based information to users of model checkers requires the translation of boolean variables into user-level notations. The remainder of this section sketches how this can be done.

Users of model checkers typically input a Kripke structure and a formula in a temporal logic; the boolean variables that the model checker manipulates constitute assertions about whether or not a given state in the Kripke structure satisfies a given temporal formula that is derived in some sense from the formula input by the user. A *decorated support set* includes functions for extracting this information from boolean variables.

Definition 3.2.1 *Let $T = \langle S, s_I, \rightarrow, V\rangle$ be a Kripke structure, ϕ be a formula in a temporal logic Φ (i.e. Φ is the set of formulas) with satisfaction relation $\models_T \subseteq S \times \Phi$. Also let $\mathcal{E}$ be a boolean equation system with $\mathcal{X} = lhs(\mathcal{E})$. Then $\langle \Gamma, \pi_T, \pi_\Phi\rangle$ is a* decorated support set *if $\Gamma = \langle r, X, \Xi\rangle$ is a support set over $\mathcal{X}$ and $\pi_T : \mathcal{X} \rightarrow S$, $\pi_\Phi : \mathcal{X} \rightarrow \Phi$ satisfy the following for all X_i such that $\Xi(X_i)$ is defined.*

1. *$\pi_S(X) = s_I$ and $\pi_\Phi(X) = \phi$.*

2. *If $\Xi(X_i)$ is defined then $\pi_S(X_i) \models_T \pi_\Phi(X_i)$ iff $r = \mathbf{1}$.*

3. *If $X_j \in \Xi(X_i)$ then either $\pi_T(X_i) = \pi_T(X_j)$ or $\pi_T(X_i) \rightarrow \pi_T(X_j)$.*

In a decorated support set π_T and π_Φ extract state and temporal-formula information from the boolean variables in Γ. Condition 1 requires that the support variable of Γ be mapped to the start state of T and the initial formula ϕ, while Condition 2 stipulates that the value returned in Γ respect the semantics of the temporal logic. Condition 3 requires dependencies among boolean variables to "respect" T's transition relation.

Chapter 4

Extracting Support Sets

As a generic vehicle for conveying model-checker reasoning, support sets are only useful to the extent that existing model checkers can be modified to compute them. In this chapter we show how this may be done by presenting two extended examples.

We begin by noting that for explicit-state mu-calculus model checkers, whether global [1, 15, 17] or local [1, 3, 31], the extraction of support sets is straightforward, since such procedures typically work by implicity or explicitly converting a mu-calculus model-checking problem into a boolean equation system as described in Chapter 2. As an example, we will show how support sets can be built by an on-the-fly algorithm [31] without major changes on it data structures. We then show how an automaton-based algorithm for CTL* may be modified to construct support sets [30]. This algorithm is not obviously mu-calculus-related; nevertheless, support-set information may be extracted without compromising the time or space complexity of the procedure.

4.1 Case I: A Local Algorithm for the Full μ-calculus

We choose as our first example a local algorithm, *LAFP* [31], for solving boolean equation system. Since chapter 2 has shown a μ-calculus model-checking problem can be translated to a boolean equation system, *LAFP* may be used to solve model-checking problem in the domain of the full μ-calculus. One reason we choose it is because the algorithm has been implemented on at least two verification tools [41, 13], and in both cases shows superior performance in practice, which is partially due to its on-the-fly style evaluation. It is also the base for an algorithm in [50] which, in addition to good practical performance, matches the best complexities of linear-space global algorithm for the full μ-calculus [32]. The technique introduced in this example can be naturally extended to the algorithm in [50], and its idea of deploying dependency relations among variables to generate support sets may apply to many other global as well as local algorithms.

[31] introduces *LAFP* on a framework called *partition dependency graph* (PDG), which can been seen as a graphic presentation of boolean equation system. Figure 1 rewrite the psudo-code of *LAFP* in the language of boolean equation system. Starting with a distinguished variable X_1, *LAFP* evaluates boolean equation system $\mathcal{E} = \sigma_1 X_1 = f_1 \cdots \sigma_n X_n = f_n$ in a demand-driven fashion. It maintains three disjoint sets for variables being visited: Q, the variables whose values are uncertain; I, the variables currently evaluated as $\mathbf{1}$; and O, the variables currently evaluated as $\mathbf{0}$. In addition, each variable in $I \cup O$ is associated with a set of variables, called a *dependency set*, as the evidence for its current value. A variable in $I \cup O$ can be put back to Q if any variable in its dependency set changes its current values, i.e., its "evidence" has been invalidated.

Each variable is also assigned a default value based on its parity: a ν-variable has the default value $\mathbf{1}$, and a μ-variable has the default value $\mathbf{0}$. A variable in Q has the default value as its value. A variable in $I \cup O$ is *alternating* if its value is different from its default value, and it is *straight* otherwise. We denote $Q^+ = \{X \mid X \in Q \wedge \sigma(X) = \nu\}$ and $Q^- = Q - Q^+$.

Q can be viewed as the set of subgoals $LAFP$ needs to solve. Variables are evaluated in decreasing order of their depths. The main iteration of $LAFP$ picks a shallowest variable in Q, say X_i. A *configuration* of $LAFP$ is captured by the snapshot of Q, I, O, and dependency sets. If the current configuration asserts X_i as $\mathbf{1}$, as indicated by the condition of *case 1*, then X_i will be moved to I and $closure^I(X_i, S)$ is called to compute the dependency set for X_i. Here, S is a disjunct of $rhs(X_i)$ which contains only the positive variables. We refer to S as X_i's *direct dependency set.* If X_i is an alternating variable, then *restoring strategy*, represented by line $9-12$ and $17-20$, will check whether the change of X_i invalidates the "evidence" (dependency sets) for some other variables. If so, it will put those affected variables back to Q for further evaluation. *Case 2* is basically the dual of *case 1*. The *default case* states that more variables need to examined (visited) if current configuration cannot make X_i either true or false.

Lemma 4.1.1 *[31] The main loop of LAFP maintains the following invariant.*

a. If $X \in I$, then $X \in [\![rhs(X_i)]\!](X.T)$ and $X.T \subseteq I \cup Q^+$

b. If $X \in O$, then $X \in [\![rhs(X_i)]\!](\mathcal{X} - X.T)$ and $X.T \subseteq O \cup O^-$

At the end of execution we can construct a support set $\Gamma = \langle r, X_1, \Xi \rangle$ as follows. Note that X_1 will end up in either I or O. r is the value of X_1. If $r = \mathbf{1}$, then $\Xi(X_i)$ is defined for every $X_i \in I$ such that $\Xi(X_i)$ is just X_i's

```
procedure eval( ℰ ≡ (σ₁X₁ = f₁ ⋯ σₙXₙ = fₙ) )
    initialize I := ∅, O := ∅, Q := {X₁}
    while Q ≠ ∅ do
      choose Xᵢ as one of the depest variables in Q.
      case
        1. ∃S ∈ disj(fᵢ). S ∈ Q⁺ ∪ I:
             I := I ∪ {Xᵢ}
             Q := Q − {Xᵢ}
             Xᵢ.T = Closure¹(Xᵢ, S)
             if σ(Xᵢ) = μ then
               M := {Xⱼ | Xⱼ ∈ O, Xᵢ ∈ Xⱼ.T}
               Q := Q ∪ M
               O := O − M
        2. ∃S ∈ conj(fᵢ). S ∈ Q⁻ ∪ O:
             O := O ∪ {Xᵢ}
             Q := Q − {Xᵢ}
             Closure⁰(Xᵢ, S)
             if σ(Xᵢ) = ν then
               M := {Xⱼ | Xⱼ ∈ I, Xᵢ ∈ Xⱼ.T}
               Q := Q ∪ M
               I := O − M
        3. otherwise:
             choose Xⱼ ∈ Var(fᵢ) such that Xⱼ ∉ Q ∪ I ∪ O
             Q := Q ∪ {Xⱼ}
```

Figure 1: Pseudo-code for LAFP [31]

dependency set $X_i.T$. If $r = \mathbf{0}$, then $\Xi(X_i)$ is defined for those $X_i \in O$ and $\Xi(X_i) = \mathcal{X} - X_i.T$.

Theorem 4.1.2 $\Gamma = \langle r, X_1, \Xi \rangle$ *constructed above is a support set.*

Proof: We only consider the case $r = \mathbf{1}$. The case where $r = \mathbf{0}$ can be proven similarly. Note that $\Xi(X_1)$ is always defined, we only need to show Γ satisfies Properties I, II, and III of definition 3.1.1. Let X be a variable defined on Ξ, hence $X \in I$. Property I holds because $X \in [\![rhs(X_i)]\!](X.T) =$

procedure Closure$^1(X_i, S)$

$X_i.T := S$

do the following until $X_i.T$ stops increasing

  if $X_j \in X_i.T$ and $(\sigma(X_j) = \mu$ or $h(X_j) > h(X_i))$ **then**

    $X_i.T := X_i.T \cup X_j.T$

procedure Closure$^0(X_i, S)$

$X_i.T := S$

do the following until $X_i.T$ stops increasing

  if $X_j \in X_i.T$ and $(\sigma(X_j) = \nu$ or $h(X_j) > h(X_i))$ **then**

    $X_i.T := X_i.T \cup X_j.T$

Figure 2: Construct the dependence set in LAFP

$[\![rhs(X_i)]\!](\Xi(X))$ by Lemma 4.1.1. Note that Q is empty when *LAFP* terminates. Using induction on $\underset{\Gamma}{\rightarrow}$ and Lemma 4.1.1, one may show that Property II holds.

Property III can be shown by contradiction. Suppose that X_{p_1} is among the shallowest variables on the loop $\rho = X_{p_1} \underset{\Gamma}{\rightarrow} \cdots \underset{\Gamma}{\rightarrow} X_{p_1}$ and $\sigma(X_{p_1}) = \mu$. Note that by the definition of $\underset{\Gamma}{\rightarrow}$, $X_{p_{i+1}} \in X_{p_i}.T$ for each pair of neighboring variables on ρ. Using induction on ρ and the definition of closure operator $closure^I$, one may show that $X_{p_1}.T$ contains every variable on ρ: base case $X_{p_2} \in X_{p_1}.T$ is clear. Assume we want to show $X_{p_{i+1}} \in X_{p_1}.T$. There are two cases,

1. X_{p_i} is deeper than X_{p_1}. By the induction hypothesis $X_{p_i} \in X_{p_1}$, hence $X_{p_i}.T \subseteq X_{p_1}.T$. Note that $X_{p_{i+1}} \in X_{p_i}.T$, therefore $X_{p_{i+1}} \subseteq X_{p_1}.T$.

2. X_{p_i} is in the same block as X_{p_1}. Clearly by Lemma 4.1.1 every node on ρ will be in I. The fact $\sigma(X_{p_1}) = \mu$ implies that both X_{p_1} and X_{p_i} are in a μ-block so they are alternating variables. Therefore, $X_{p_i}.T \subseteq X_{p_1}.T$ by $closure^I$. It follows that $X_{p_{i+1}} \in X_{p_1}.T$.

Now we have $X_{p_1} \in X_{p_1}.T$. Assume that X_{p_1} has been moved from Q to I at k-th iteration, then X_{p_1} can be included into its own dependency set only in one of the following cases.

1. X_{p_1} is an element of direct dependency set S and X_{p_1} has the value of $\mathbf{0}$ before k-th iteration, which is impossible because $X_{p_1} \in Q$ and $\sigma(X_{p_1}) = \mu$ before k-th iteration.

2. $X_{p_1} \in X_j.T$, $X_j \in X_i.T$ and, either X_j is alternating or X_j is higher than X_i. In either case, X_j is in I to ensure variant a of Lemma 4.1.1 for X_i. To ensure variant a of Lemma 4.1.1 for X_j, X_{p_1} has to be evaluated as $\mathbf{1}$ before the k-th iteration, which contradicts the assumption $X_{p_1} \in Q$ and $\sigma(X_{p_1}) = \mu$.

Therefore, an alternating variable X_{p_1} cannot appear in its own dependency set, hence loop ρ cannot exist.

$\blacksquare$

In this example, support sets are constructed from dependency sets. Since *LAFP* otherwise still computes and maintains dependency sets, the extra effort of constructing support sets will not increase the space and time complexities of *LAFP* . Another alternative way to do this is to use direct dependency sets instead of dependency sets. Following similar reasoning, we can prove that this still yields a support set. One may show that support sets constructed by the latter are *reduced* and those constructed by the former are *canonical*. The general technique of constructing support sets from μ-calculus model checkers is to retain the *local dependency relation* during the computation that records, for each variable, a set of variables causing the change of its value.

4.2 Case 2: An Automaton-based Model Checker for CTL*

In automaton-based model checking for CTL*, formulas are converted into tree automata accepting the trees that make the formula true. Checking whether a Kripke structure satisfies a formula involves determining whether or not the (infinite) tree obtained by unwinding the Kripke structure is accepted by the formula's tree automaton. This acceptance checking is typically performed by viewing the Kripke structure itself as a tree automaton accepting the (single) tree obtained by the unwinding process just mentioned, computing a product with it and the automaton for the formula in question, and then checking whether or not the resulting product automaton is nonempty.

The automaton-based model checker considered below comes from [30], although for technical convenience the definitions of the automata used borrow ideas from [21] as well. Recall that $\mathcal{A}$ is the (fixed) set of atomic propositions.

Definition 4.2.1

1. *An* alternating tableau transition system *is a tuple* $\langle Q, \rightarrow, q_I, \ell \rangle$*, where* Q *is a finite set of states,* $\rightarrow \subseteq Q \times Q$ *is the transition relation,* $q_I \in Q$ *is the start state, and* $\ell \in Q \rightarrow \mathcal{A} \cup \{\neg A \mid A \in \mathcal{A}\} \cup \{\wedge, \vee, [], \langle\rangle\}$ *is the labeling, and the following holds for all* $q \in Q$.

$$|\{q' \mid q \rightarrow q'\}| \begin{cases} = 0 & \textit{if } \ell(q) \in \mathcal{A} \cup \{\neg A \mid A \in \mathcal{A}\} \\ \geq 1 & \textit{if } \ell(q) \in \{\wedge, \vee\} \\ = 1 & \textit{if } \ell(q) \in \{[], \langle\rangle\} \end{cases}$$

2. *Alternating tableau transition system* $\langle Q, \rightarrow, q_I, \ell \rangle$ *is* hesitant *if for every non-trivial*[1] *strongly connected component* $Q_i \subseteq Q$ *of the graph* $\langle Q, \rightarrow \rangle$

[1] *A strongly connected component* Q_i *is nontrivial if there exist* $q, q' \in Q_i$ *such that* $q \rightarrow q'$.

and every $q \in Q_i$, either $\ell(q) \in \{\wedge, []\}$ or $\ell(q) \in \{\vee, \langle\rangle\}$. In the former case Q_i is called existential*, while in the latter it is called* universal.

3. *A* hesitant alternating tableau automaton (HATA) *is a tuple $\langle Q, \rightarrow, q_I, \ell, \langle G, B\rangle\rangle$ where $\langle Q, \rightarrow, q_I, \ell\rangle$ is a hesitant alternating tableau transition system and $G, B \subseteq Q$ constitute the acceptance condition.*

HATAs are very similar to the hesitant alternating automata in [30]; the only significant difference is the use of labels on states rather than transitions to record "alternation information".

HATAs generate "runs" as they process Kripke structures.

Definition 4.2.2 *Given HATA $M = \langle Q, \rightarrow, q_I, F, \ell, \langle G, B\rangle\rangle$ and a Kripke structure $T = \langle S, s_I, \rightarrow, V\rangle$, a* run *of M on T is a maximal tree in which the nodes are are labeled by elements of $S \times Q$ as follows. (1) The root of the tree is labeled by $\langle s_I, q_I\rangle$. (2) For each node σ labeled by $\langle s, q\rangle$:*

1. *If $\ell(q) \in A$ then σ is a leaf.*

2. *If $\ell(q) = \wedge$ and $\{q' \mid q \rightarrow q'\} = \{q_1, \cdots, q_m\}$, then σ has children $\sigma_1, \cdots, \sigma_m$, with σ_i labeled by $\langle s, q_i\rangle$.*

3. *If $\ell(q) = \vee$ then σ has one child, σ', which is lableled by $\langle s, q'\rangle$ for some $q' \in \{q' \mid q \rightarrow q'\}$.*

4. *If $\ell(q) = []$, $q \rightarrow q'$, and $\{s' \mid s \rightarrow s'\} = \{s_1, .., s_m\}$ then σ has children $\sigma_1, .., \sigma_m$, with σ_i is labeled by $\langle s_i, q'\rangle$.*

5. *If $\ell(q) = \langle\rangle$ and $q \rightarrow q'$ then σ has one child σ', and σ' is labeled by $\langle s', q'\rangle$ for some s' such that $s \rightarrow s'$.*

Note that any infinite path in a run eventually consists only of states from the same nontrivial strongly connected component. We call such an infinite path *existential* if this component is existential and *universal* otherwise. Because the transition relation of T is total, the leaves in a run must be labeled either by $\langle s, A\rangle$ or $\langle s, \neg A\rangle$ for some $A \in \mathcal{A}$. We call leaves *successful* if they are labeled $\langle s, A\rangle$ and $s \in V(A)$ or $\langle s, \neg A\rangle$ and $s \notin V(A)$. A run is *successful* iff: every leaf is successful; every existential infinite path contains infinitely many occurrences of states in G; and every universal infinite path contains finitely many occurrences of states in B.

In order to determine whether or not a HATA M has a successful run on Kripke structure T, [30] advocates checking the nonemptiness of a product automaton built from M and T. In the slightly revised setting considered here the nonemptiness check may be defined on an and-or graph $G_{T,M}$ defined as follows.

- The vertex set of $G_{T,M}$ is $S \times Q$.
- The edge relation E is defined by: $\langle\langle s, q\rangle, \langle s', q'\rangle\rangle \in E$ iff $\langle s', q'\rangle$ satisfies the conditions of being a child of $\langle s, q\rangle$ in some run of M on T.
- Labeling function $f \in S \times Q \to \{\wedge, \vee\}$ is defined as follows.

$$f(\langle s, q\rangle) = \begin{cases} \wedge & \text{if } \ell(q) \in \{\wedge, []\} \text{ or } \langle s, q\rangle \text{ would be a successful leaf} \\ & \text{in some run} \\ \vee & \text{otherwise} \end{cases}$$

The model-checking routine works by assigning truth values **0** and **1** to vertices in $G_{T,M}$. Details may be found in [30], but the rough idea is to process the strongly connected components in $G_{T,M}$ in reverse topological order, starting with components containing no edges to other components. Strongly connected components in $G_{T,M}$ may be seen to have the property that all nodes share the same label ($\wedge$ or $\vee$). Sink nodes in the graph correspond to successful or unsuccessful leaves in some run and are assigned **1** in the first case and

$\mathbf{0}$ in the latter. The following process is then repeated for each component. First, values in lower components that the current component has edges into are "propagated upwards" into the current component, and new boolean values assigned to nodes in the current component in the obvious manner (i.e. a $\vee$-labeled node is assigned $\mathbf{1}$ if it has an edge to a node assigned boolean value $\mathbf{1}$, etc.). This propagation process is continued until no more is possible. If the component still has unlabeled nodes then all nodes are assigned $\mathbf{1}$ if the component is labeled $\vee$ and there exists a vertex in the component of form $\langle s, q\rangle$ for some $q \in G$, or if the component is labeled $\wedge$ and there is no vertex of form $\langle s, q\rangle$ for some $q \in B$; and $\mathbf{0}$ otherwise. It may be shown that $\langle s_I, q_I\rangle$ is assigned $\mathbf{1}$ if HATA M accepts Kripke structure T and $\mathbf{0}$ otherwise.

In order to extract support sets from the information computed by this model-checking algorithm, we first must define a closed boolean equation system from T and M. This may be done from $G_{T,M}$ as follows. *Variables* correspond to vertices in $G_{T,M}$, while *right-hand sides* are constructed from labels and edges in $G_{T,M}$: if the label of a vertex v is $\wedge$, then the right-hand side for the variable v is $\bigwedge\{v' \mid \langle v, v'\rangle \in E\}$, and similarly for $\vee$. *Blocks* are constructed from the strongly connected components of $G_{T,M}$. Let $G_1, \ldots, G_m$ be these components listed in topographical order: if $i < j$ then there is no edge from any node in G_j to G_i. For each G_i we construct two parity blocks E_i, E'_i as follows.

If G_i's label is $\vee$: Let E_i contain the equations whose left-hand sides $\langle s, q\rangle$ are nodes in G_i and with the property that $q \in G$. Let E'_i consist of the other equations whose left-hand sides are in G_i. Assign ν as the parity of E_i and μ as the parity of E'_i.

If G_i's label is $\wedge$: Let E_i contain the equations whose left-hand sides $\langle s, q\rangle$ are nodes in G_i and with the property that $q \in B$. Let E'_i consist of the other equations whose left-hand sides are in G_i. Assign μ as the parity of E_i and ν as the parity of E_i'.

The Boolean equation system $\mathcal{E}_{T,M}$ is the sequence $E_1 E_1' \ldots E_m E_m'$.

In order to construct a support set from a run of $G_{T,M}$, we need to record for each (labeled) node the set of variables its value depends on. Note that the labelling procedure is conducted in two steps for each strongly-connected component. The strategy for marking dependency sets for variables in component G_i are different in two steps.

1. In the first step, label those variables whose values can be determined from the values of leaves or labeled nodes in lower components. The dependency set of such a variable is the set of its children whose values determine its new value.

2. In the second step, the unlabeled variables are labeled with the same value r. We consider the case that G_i is existential. The case where G_i is universal can be handled dually. If there is a variable $\langle s', q' \rangle \in G_i$ such that $q' \in G$, then all the unlabeled variables will be labeled with $\mathbf{1}$. To mark dependency sets, we conduct a depth-first search from $\langle s', q' \rangle$ in C_i. The dependency set for a variable will contain only its parent, except for $\langle s', q' \rangle$. The dependency set of $\langle s', q' \rangle$ will contain any one of its children. If there is no variable $\langle s', q' \rangle \in G_i$ such that $q' \in G$, then all the unlabeled variable will be labeled as 0, and the dependency set of such a variable will contain its children.

Clearly the time and space complexities of marking dependency sets are linear in the size of graph, and hence the extraction will not increase the complexities of model-checking algorithm.

With the dependency relation being recorded, now we can construct $\Gamma = \langle r, \langle s_I, q_I \rangle, \Xi \rangle$ as below. We consider the case $\langle s_I, q_I \rangle$ is labeled with $\mathbf{1}$. The other case can be handled similarly. r is the value of $\langle s_I, q_I \rangle$, i.e., $r = 1$. Ξ

is defined for every labeled variable, and $\Xi(\langle s_i, q_i \rangle)$ is the dependency set of $\langle s_i, q_i \rangle$ marked as above.

Lemma 4.2.3 *Let $\mathcal{E}$ be the boolean equation system for Kripke structure T and HATA M, $\Gamma = \langle r, \langle s_I, q_I \rangle, \Xi \rangle$ constructed above is a support set for $\mathcal{E}$.*

Proof: We consider the case $r = \mathbf{1}$. The case $r = \mathbf{0}$ can be handled similarly. Clearly Γ satisfies Property I of Definition 3.1.1. To see Property II, note that for any variable $\langle s_i, q_i \rangle$ covered by Γ, every $\langle s_j, q_j \rangle$ which $\langle s_i, q_i \rangle$'s value depends on is labeled either before $\langle s_i, q_i \rangle$ (in the first step) or at the same $\langle s_i, q_i \rangle$ is labeled (in the second step). Hence $\langle s_j, q_j \rangle$ should also be covered by Γ.

Property III can be shown by induction on the topological ordering of components. Suppose that we have already proved all the variables in lower components satisfy III, and now we want to show that each variable in C_i also satisfies III. We consider the case C_i is existential. The case where C_i is universal can be handled dually. Let $\langle s, q \rangle$ be a variable in C_i. If $\langle s, q \rangle$ is labeled in the first step of labelling C_i, then, the dependency relation among $\langle s, q \rangle$ and the variables it directly or indirectly depends on is a path, which leads $\langle s, q \rangle$ to a leaf or a variable in a lower component which is labeled $\mathbf{1}$. Hence, $\langle s, q \rangle$ won't involve any loop in C_i. If, otherwise, $\langle s, q \rangle$ is labeled during the second step, then there are two cases.

1. If there is a vertex $\langle s', q' \rangle$ is in G, then all unlabeled variables will be labeled with $\mathbf{1}$. The dependency relation among these variables constitute a DFS tree with the root $\langle s', q' \rangle$. The only loop is caused by the dependency set of $\langle s', q' \rangle$, which links the root to a variable on the tree. The only loop $\langle s, q \rangle$ may involve must go through a ν-variable $\langle s', q' \rangle$ in $\mathcal{E}$. By the construction of $\mathcal{E}$, ν-variable $\langle s', q' \rangle$ is also in the shallower block, hence any loop involving $\langle s', q' \rangle$ respects III.

2. If no vertex is in G, then all unlabeled variables will be labeled with $\mathbf{0}$. Therefore, no loop in C_i involves a variable in G. By the construction of $\mathcal{E}$, this means that the variables in any loop are constrained to a μ-block. Property III also holds for this case.

■

Lastly, the successful extraction of a support set from a run of model-checking algorithm can be used to prove the correctness of algorithm. We will not directly make an example on this; we instead will show how the existence of support sets means that the translation from a HATA mode-checking problem to a Boolean equation system preserves the semantics of the original problem.

Theorem 4.2.4 *Let T be a Kripke structure and M a HATA, and let s be a state in T and q a state in M. Then $[\![\mathcal{E}_{T,M}]\!](\langle s,q\rangle) = \mathbf{1}$ iff the value assigned to node $\langle s,q\rangle$ in $G_{T,M}$ by the model-checking algorithm of [30] is $\mathbf{1}$.*

Proof: If $\langle s,q\rangle$ is labeled with $\mathbf{1}$, then by Lemma 4.2.3 a support set with support value $\mathbf{1}$ can be constructed from a run of algorithm [30] which labels $\langle s,q\rangle$ with $\mathbf{1}$. By Theorem 3.1.6, this means $[\![\mathcal{E}_{T,M}]\!](\langle s,q\rangle) = \mathbf{1}$. Similarly, a run which labels $\langle s,q\rangle$ with $\mathbf{0}$ means the existence of a support set with support value $\mathbf{0}$. Therefore, $[\![\mathcal{E}_{T,M}]\!](\langle s,q\rangle) = \mathbf{0}$ by Theorem 3.1.6. Since a run of algorithm [30] marks $\langle s,q\rangle$ with either $\mathbf{0}$ or $\mathbf{1}$, therefore, we have proven both directions of theorem. ■

We conclude this section with some comments on decorated support sets. In CTL* automaton-based model checking the HATA is constructed from a CTL* formula provided by the user. As the model checker should return a decorated support set, one may wonder how to define the functions π_T and π_Φ. In the procedure just outlined the mapping π_T is straightforward, since every boolean variable corresponds to a pair $\langle s,q\rangle$, where s is a system state.

As for π_ϕ, the HATA constructions in [21, 30] work by associating HATA states with (sets of) CTL* propositions. These CTL* propositions can then be returned by π_Φ.

Chapter 5

Certifying Model-Checking Results

Support sets constitute the evidence to justify model-checking results. In the next two chapters we will show how this evidence can be used for a variety of "post-model-checking" analyses. We start with an efficient algorithm which checks the validity of a support set submitted by a model checker. Such a routine has several practical motivations [39]:

- It can be used to check for bugs in model checkers: if a support set returned by a model checker is in fact not a support set, then the reasoning undertaken by the model checker is faulty.

- Structures like support sets can be used as "certificates" attesting to system correctness. A support-set validity checker can then be used to check the "internal consistency" of such a certificate, i.e. that it represents a valid piece of reasoning.

As stated in Chapter 1, the existence of such an efficient certifying routine is also one of the general requirements that support sets should meet.

In what follows we fix $\mathcal{E} = E_1 \dots E_n$ as the boolean equation system in question with $E_1 \cdots E_n$ as its parity blocks. Let $\Gamma = \langle r, X, \Xi \rangle$ be a support set for $\mathcal{E}$ submitted by a checker. Without loss of generality, assume $r = \mathbf{1}$. Validating Γ amounts to checking that Properties I, II and III in Definition 3.1.1 hold. Properties I and II on Γ can be ascertained with routines that each execute in $O(|\Gamma|)$. The former checks whether the dependency set $\Xi(X_i)$ includes a disjunct from the right-hand side of X_i. The latter involves a reachability test on the paths induced by $\xrightarrow[\Gamma]{}$.

Checking Property III on Γ can be reduced to an *even-cycle* problem on labeled directed graphs. A labeled directed graph is a tuple $G = \langle D, V, E, \ell \rangle$, where $\ell : V \to D$ labels each vertex with a element from D. The *even-cycle* problem is given as follows.

> Given a labeled directed graph $G = \langle \{1, 2, \cdots, k\}, V, E, \ell \rangle$, determine whether there is a cycle ρ in it such that $min_{v \in \rho}\{\ell(v)\}$ is even.

Γ induces a labeled directed graph $G = \langle \{1, 2, \cdots, k\}, V, E, \ell \rangle$ as follows. V is the set of all variables defined on Ξ, E is the relation $\xrightarrow[\Gamma]{}$; ℓ satisfies the following criteria.

- If $X', X'' \in lhs(E_i)$ then $\ell(X') = \ell(X'')$.
- If $i < j$ then $\ell(E_i) \leq \ell(E_j)$. (Here $\ell(E_i)$ is the common value shared by all left-hand sides in E_i.)
- If the parity of E_i is μ then $\ell(E_i)$ is even; otherwise, $\ell(E_i)$ is odd.

A labeling satisfying these properties can easily be constructed in $|\mathcal{E}|$ time with $k \leq n$, where n is the number of blocks.

Checking III on Γ is equivalent to checking whether there is an even cycle in G. [26] shows that the even-cycle problem can be solved in

$O((|V|+|E|)log(\lceil\frac{k}{2}\rceil))$. Their approach is a variant of an algorithm for hierarchical clustering [51]. Interested readers may refer to [26] for the description of algorithm. As our construction of ℓ above restricts $k \leq n$, checking property III can be done in $O(|\Xi|log(\lceil\frac{n}{2}\rceil))$, where n is the number of parity blocks of $\mathcal{E}$.

The complexity can be improved by noticing that an even-cycle can only exist in a strongly-connected component of G_Γ. Therefore, we can check each strongly-connected component independently. By definition 2.3.1, the maximal labeling number k won't exceed the alternation depth of $\mathcal{E}$. Thus, the overall time complexity is $O(|\mathcal{E}| \cdot |\mathcal{T}| \cdot log(\lceil\frac{ad(\mathcal{E})}{2}\rceil))$, which is lower than the lower bound of μ-calculus model checking. This suggests that the certifier will not in general increase the cost of overall complexities of a verification tool.

Chapter 6

Generating Diagnostic Information

Building checker-independent diagnostic routine is one of our initiatives to create such framework. In this chapter we will show how different diagnostic information can be generated from support sets. We start with one of most popular diagnostic schema, linear counterexample, and follow by a novel diagnostic schema, interesting witness. We conclude our discussion with model-checking game, an interactive way to diagnose system in which users enjoy the freedom to explore only the part of proof they are interested in.

6.1 Counterexample and Witness

Counterexamples are widely used to indicate why a Kripke structure fails to satisfy a temporal property. Intuitively, a counterexample is a part of system "responsible" for the property being violated. Dually, when system satisfies a temporal property, a *witness* is provided as a portion of the system responsible for the property being satisfied. Although counterexample generators have existed for a number of years, to the best of our knowledge [7] represents the first systematic explanation of how they work. Their definitions in the setting of CTL require that counterexamples and witnesses to be *linear*, i.e., execution paths of the system. Their definition may be written as below,

Definition 6.1.1 *[7]*

If $\mathsf{A}f$ *is a failing formula in model* $T = \langle S, s_I, \rightarrow, V \rangle$*, then* C *is a counterexample for* $\mathsf{A}f$ *in* T *iff* C *is a computation path of* T *such that* $C \not\models_T f$*.*

If $\mathsf{E}f$ *is a successful formula in model* $T = \langle S, s_I, \rightarrow, V \rangle$*, then* C *is a witness for* $\mathsf{E}f$ *in* T *iff* C *is a computation path of* T *such that* $C \models_T f$*.*

In general the existence of linear counterexamples / witnesses depends on the structure of formulas as well as the Kripke structure being checked. In the case of CTL, for example, linear counterexamples (witnesses) exist if the primary path quantifier used is A (E). [2] also gives the definition for witness and counterexample in $ACTL$. Their definition is semantics-based:

A counter-example of a failing formula φ in model $T = \langle S, s_I, \rightarrow, V \rangle$ is a computation path $C = s_0 \cdots$ in T such that $s_0 \not\models_C \varphi$.

[29] extends the definitions to CTL* and show that judging whether a CTL* formulae admits such counterexamples / witnesses under the definitions of [2] is PSPACE-complete.

It's worth to point out that the definitions in [2, 29] are not exactly same as in [7]. Let's consider the formula $\varphi = AG(AFa \vee AFb)$ and the transition system T_2 in Figure 3. According the definition in [7], path $C = s_0 s_2$ is a counterexample for φ because s_0 don't satisfy $\phi = \mathsf{A}\mathbf{F}a \vee \mathsf{A}\mathbf{F}b$ in model T_2. Nevertheless, under the model C, both s_0 and s_2 satisfies ϕ. Hence C is not a counterexample by the definitions of [2].

An alternative way to understand the definitions in [7] is below. We create a new atomic proposition for every top-level state subformula in φ. Let $\mathcal{B}$

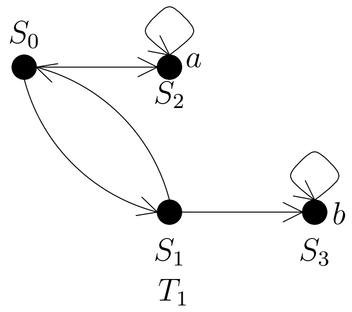


Figure 3: Constructing counter-examples

be the set of these new atomic propositions such that $\mathcal{B} \cap \mathcal{A} = \emptyset$ and F be the function which maps each top-level state subformula to its corresponding atomic proposition in $\mathcal{B}$. A decorated path $C = \langle S, s_I, \rightarrow', V' \rangle$ in model T is a computation path of T such that $V'(s) = V(s) \cup \{b \mid F(\phi) = b \wedge s \models_T \phi\}$. By definition a decorated path is also a transition system. Counterexample in [7] can be defined in term of decorated path and formula $\varphi[F]$,

Definition 6.1.2 *Let $\mathcal{B}$ be a set of atomic propositions distinct from $\mathcal{A}$. A counter-example of a failing formula φ in model $T = \langle S, s_I, \rightarrow, V \rangle$ is a decorated path $C = \langle S, s_I, \rightarrow', V' \rangle$ of T under F such that $C \models \varphi[F]$, where F replaces each top-level subformulae of φ with an unique proposition in $\mathcal{B}$.*

We extend the notion of counterexamples to Boolean equation system without reference to the actual temporal logics it is formulated from. In the definition of [7], we focus on the structure of formula and treat each top-level state subformula as *atomic*. Similarly, when defining a counterexample for μ-calculus equation system, we focus on the equations in the top-level strongly-connected component and treat the semantics of variables in deeper components as *atomic*. We start by defining branching degrees.

Definition 6.1.3 *Let $\langle \Gamma, \pi_T, \pi_\Phi \rangle$ be a decorated support set with $\Gamma = \langle r, X, \Xi \rangle$. For each variable X such that $\Xi(X)$ is defined, its branching degree is given* *as.*

$$deg(X) = |\{X' \mid X \underset{\Gamma}{\rightarrow} X' \wedge \rho(X, X')\}|$$

where $\rho(X, X')$ *is true iff* X *and* X' *are in the same strongly-connected component induced by* $\underset{\Gamma}{\rightarrow}$. *The branching degree of* $\langle \Gamma, \pi_T, \phi_\Phi \rangle$ *is given as,*

$$deg(\langle \Gamma, \pi_T, \pi_\Phi \rangle) = max\ \{deg(X') \mid \rho(X, X')\}$$

A decorated support set is linear if the branching degree of the strongly-connected component containing X *is no greater than 1. It is recursively linear if the branching degree of every strongly-connected component is no greater than 1.*

If a decorated support set with support value $r = \mathbf{0}$ is linear then one may extract a *linear counterexample* to the result contained in the support set as follows. Let $\Gamma = \langle \Gamma' = \langle \mathbf{0}, X, \Xi \rangle, \pi_T, \pi_\Phi \rangle$ be a linear decorated support set for Kripke structure T. Then the *state projection* $\pi_S(\Gamma')$ is defined as follows: let $C(X)$ be the strongly-connected component of X induced by Ξ and $X_1 X_2 \ldots X_n$ be a depth-first search within $C(X)$ beginning at $X = X_1$. Then $\pi_S(\Gamma) = \pi_T(X_1) \ldots \pi_T(X_n)$. In general, $\pi_S(\Gamma)$ is not an execution sequence of T, since the definition of decorated support set allows the states associated with adjacent variables in Γ to be the same, and hence not connected by a transition. However, a subsequence of $\pi(\Gamma)$ is guaranteed to be a computation path of T: delete all but one occurrence of a state in contiguous subsequences containing only this state. Let $\pi(\Gamma)$ be this sequence; it is easy to show that it is a computation path in T that is a linear model of the result reported by the model checker.

The construction of counterexamples just described focuses on the primary strongly-connected component while treating the variables in deeper blocks as atomic propositions. If users would like to understand the evaluation of these variables, the above construction procedure may be called to recursively construct a counterexample on deeper strongly-connect components. For example, assume that X_i depends X' and X' has been omitted during the construction of counterexample because X' is in a different strongly-connected component. Note that $\langle \mathbf{0}, X', \Xi \rangle$ is still a support set. The construction procedure can be called on $\langle \Gamma'' = \langle \mathbf{0}, X', \Xi \rangle, \pi_T, \pi_\Phi \rangle$ if users want a counterexample to investigate X'. The notion of *recursively linear* ensures that such construction is feasible on any strongly-connected component, which gives users the power to obtain a counterexample on any level. Moreover, the witness-extraction procedure can be applied to support sets that, while not linear themselves, reduce to linear support sets.

Finally, we point out that the linear counterexample defined and constructed in context of support set is consistent with the original definition in [7]: a CTL* formula φ can be translated to a *hesitant* μ-calculus equation system $\mathcal{E}_\varphi$ in the sense that, if we label each variable with the operator in its right-hand side, then the labels of variables in the same strongly-connected component of $\mathcal{E}_\varphi$ are either all from $\{\wedge, []\}$, in which case the component is referred as *universal*, or all from $\{\vee, \langle\rangle\}$, in which case the component is referred as *existential*. This translation can be done by first translating the CTL* formula to a HATA [30], then to a μ-calculus equation system per the approach given in Section 4.2. In addition, if CTL* formula has the form $\mathsf{A}f$, then the top strongly-connected component of $\mathcal{E}_\Phi$ is universal. Let $\mathcal{E} = \Pi(T, \mathcal{E}_\varphi)$ be the boolean equation system obtained by combining transition system T with $\mathcal{E}_\varphi$. The operators in the right-hand side of variables in top strongly-connected component of $\mathcal{E}$ can only be $\wedge$. Therefore, if $s_I \not\models_T \varphi$, then $\langle s_I, X_0 \rangle \notin [\![\mathcal{E}]\!]$ and there exists a decorated support set with the support value $\mathbf{0}$. Clearly we can always minimize this decorated support set to a linear one. Thus, a linear counterexample always exists if $s_I \not\models_T \varphi$.

6.2 Vacuity Detection

Traditional diagnostic information sets to help user diagnose why a system fails to satisfy a property. In the case of success, model-checking tools typically provide little information on how “well” a system passes a specification. Recently study shows that how a system satisfies a property does matter to engineers, and additional diagnostic information in successful cases may help detect potential errors in design systems. For example, [6] discusses the coverage metrics, which distinguishes “don’t care” states that don’t contribute to the model-checking result. Such states stand for the part of transition system whose behavior does not affect the formula, and hence requires further investigation by other formulas. Similarly, Beer *et al* [2] discussed the existence of “don’t-care” subformulae that do not affect the outcome of model-checking problem. Such vacuity often implies a serious problems in system designs. To quote from [2]:“Our experience has shown that typically 20% of specifications pass vacuously during the first formal-verification runs of a new hardware design, and that vacuous passes always point to a real problem in either the design or its specification or environment.” [2] discusses the vacuity problem in the setting of w-ACTL, a sublogic of universal fragment of CTL. [29] extends the discussion on CTL* and show that vacuity can be determined in $O(C_M(\varphi) \cdot |\varphi|)$, where $C_M(\varphi)$ is the complexity of model checking problem $\langle M, \varphi \rangle$.

Intuitively, a system M satisfies φ vacuously if M satisfies it in a uninteresting way. A good example for this is *antecedent failure*, which means the formula is trivially true because the antecedent of an implication is false. For instance, a system M may vacuously pass $\mathsf{AG}(\mathit{request} \rightarrow \mathsf{AX}(\mathit{response}))$ because no *request* has ever been made in any execution of M. Formally vacuity is defined as below.

Definition 6.2.1 *[2] Subformula ϕ of φ does not affect the truth value of φ in*

M (ϕ doesn't not affect φ in M, for short) if for every formula ξ, the system M satisfies $\varphi[\phi \leftarrow \xi]$ iff M satisfies φ.

Definition 6.2.2 *[2] A system M satisfies a formula φ vacuously iff $M \models \varphi$ and there is some subformula ϕ of φ such that ϕ does not affect φ in M.*

In the setting of w-ACTL, with a slightly modified notion of vacuously passing (vacuity occurs only on *important* subformula), vacuity detection can be done by checking M on a witness formula φ'. Witness formulas can be generated from a w-ACTL formula in linear time. More generally, [29] proves Lemma 6.2.3, given below. Therefore, to check whether a subformula doesn't affect φ in M in the case M passes φ, one only need to replace this formula by $\mathbf{ff}$. The replacement can be done in a user-driven fashion, or automatically by tools. Moreover, by Lemma 6.2.4 one may only need to check *literals*. The semantics of $\varphi[\phi \leftarrow \bot]$ is interpreted with respect to the *polarity* of ϕ in φ: the polarity of ϕ is *positive* if ϕ is proceeded by even number of negation in φ, otherwise the polarity of ϕ is *negative*. $\varphi[\phi \leftarrow \bot]$ is obtained by replacing ϕ by $\mathbf{ff}$, if the polarity of ϕ is positive, and replacing ϕ by $\mathbf{tt}$ otherwise. Dually, $\varphi[\phi \leftarrow \top]$ replaces a positive ϕ by $\mathbf{tt}$, and a negative ϕ by $\mathbf{ff}$.

Lemma 6.2.3 *[29] for a subformula ϕ of φ and for every system M, if $M \models \varphi[\phi \leftarrow \bot]$, then for any formula ξ, we have $M \models \varphi[\phi \leftarrow \xi]$. Also, if $M \not\models \varphi[\phi \leftarrow \top]$, then for any formula ξ, we have $M \not\models \varphi[\phi \leftarrow \xi]$.*

Lemma 6.2.4 *M satisfies φ vacuously iff $M \models \varphi$ and there is a subformula ϕ of φ such that ϕ is atomic and $M \models \varphi[\phi \leftarrow \bot]$.*

Proof: We consider the case the polarity of ϕ in φ is positive, while the negative case can be handled dually.

($\Rightarrow$) If M satisfies φ vacuously, then $M \models \varphi$ and there exists a subformula ψ such that for any formula ξ, $M \models \varphi[\psi \leftarrow \xi]$. Let ϕ be an atomic proposition in ψ. Hence $M \models \varphi[\psi \leftarrow \psi[\phi \rightarrow \mathbf{ff}]] = \varphi[\xi \rightarrow \mathbf{ff}]$.

($\Leftarrow$) By Definition 6.2.2 and Lemma 6.2.3 ϕ doesn't affect the true value of φ in M, hence M satisfies φ vacuously.

■

Support sets can be used to help detect vacuity. Intuitively, if a subformula of φ isn't *covered* by a decorated support set for the model-checking problem for φ and M, then it won't affect the truth value of φ in M. Formally we can define the notion of *vacuously support* for support sets.

Definition 6.2.5 $\Gamma' = \langle \Gamma = \langle \mathbf{1}, X, \Xi \rangle, \pi_M, \pi_\varphi \rangle$ *vacuously supports* φ *in system* $M = \langle S, s_I, \rightarrow, V \rangle$ *iff,*

1. Γ' *is a decorated support set with* $\pi_M(X) = s_I$ *and* $\pi_\varphi(X) = \varphi$
2. *there exists a subformula* l *of* φ *such that* l *is an atomic proposition and there is no* X' *such that* $\Xi(X')$ *is defined and* $\pi_\varphi(X') = l$.

Definition 6.2.5 doesn't refer to any specific temporal logics. In the setting of CTL* the existence of a Γ' which *vacuously supports* φ in M implies that M satisfies φ vacuously. To see this, let $\mathcal{E}$ be the boolean equation system for model-checking problem $\langle M, \varphi \rangle$, and $\mathcal{E}'$ be the one for model-checking problem $\langle M, \varphi[l \leftarrow \perp] \rangle$ where l is an atomic proposition in φ. We will consider the case that the polarity of l in φ is positive. The negative case can be handled similarly. $\mathcal{E}$ and $\mathcal{E}'$ differs only on the variables related to l: a variable $\langle s, l \rangle$ in $\mathcal{E}$ is replaced by $\langle s, \mathbf{ff} \rangle$ in $\mathcal{E}$. By Definition 6.2.5 Γ doesn't involve any $\langle s, l \rangle$, hence it is also a support set for $\mathcal{E}'$ and the result for model-checking problem $\langle M, \varphi[l \leftarrow \mathbf{ff}] \rangle$ should also be positive. By Lemma 6.2.3 l doesn't affect the truth value of φ in M, therefore φ satisfies M vacuously.

On the other hand, if φ satisfies M vacuously, there is no guarantee that the support set returned by checker *vacuously supports* φ in M. The reason can be illustrated by the following example: let model-checking problem be whether M is a model of $\varphi = \mathsf{EG}(p \rightarrow \mathsf{AX}(q))$. Assume there are several pathes satisfying $\mathsf{G}(p \rightarrow \mathsf{AX}(q))$, one of them, denoted as ρ, has some states satisfying p, while another path ρ' doesn't have a state satisfying p. Clearly φ passes M vacuously because of the existence of ρ'. Nevertheless, a model checker may happen to choose ρ, hence the resulting support set must cover every part of φ. But even in this case, the use of support set still brings us some benefit.

1. The previous method for vacuity detection is to try every state subformula by substituting $\perp$ for it. In the worst case, it requires $O(|\varphi|)$ calls to model checker. This method neglects the possibility that the proof used by the checker may not cover all the subformula, therefore, a single run of model checker is already sufficient to reveal *vacuity* in the model-checking problem.

2. Even if support sets cover all the state subformula, we may still minimize support sets submitted by checkers. Clearly minimization will not increase the set of subformulae covered by support sets. One may hope that an atomic proposition may go uncovered after the minimization.

6.3 Model-Checking Games

Traditional diagnostic generation, like that in Section 6.1 and 6.2, requires little or no user intervention. The downside of this is that users have been deprived of the freedom to explore only the part of proof he is interested in. Model-checking game we are about to introduce grants such freedom to users.

Games have been used to characterize the decidability problem for long time (cf. [38]). We use the notation of model-checking games due to Stirling [43]. In what follows, we fix $T = \{S, s_I, \rightarrow, V\}$ as the transition system in question and $\mathcal{E}_\mu = (\sigma_1 X_1 = f_1) \cdots (\sigma_n X_m = f_m)$ as the property encoded as a closed μ-calculus equation system. A model-checking game attempts to determine whether or not $s_I \in [\![\mathcal{E}_\mu]\!]_T(X_1)$. A model checking game involves two players: I, who believes that the answer should be "no"; and II, who assumes the opposite.

Definition 6.3.1 *A play of model-checking game $\Omega(T, \mathcal{E}_\mu)$is a maximal sequence $\alpha = \langle s_I, X_1\rangle \cdots$ such that, for every $\langle s, X\rangle$ on α, let $\sigma X = f_i$ is the equation for X, the next configuration is decided by the following.*

1. *If $f \equiv \langle\rangle X'$, then the next configuration is $\langle s', X'\rangle$, where s' is a successor of s chosen by player II.*

2. *If $f \equiv []X'$, then the next configuration is $\langle s', X'\rangle$, where s' is a successor of s chosen by player I.*

3. *If $f \equiv \vee\mathcal{X}'$, then the next configuration is $\langle s, X'\rangle$, where $X' \in \mathcal{X}'$ is chosen by player II.*

4. *If $f_i \equiv \wedge\mathcal{X}'$, then the next configuration is $\langle s, X'\rangle$, where $X' \in \mathcal{X}'$ is chosen by player I.*

Let $inf(\alpha)$ be the set of variables visited infinitely often by play α. I wins α iff α is maximal *and, either α is a finite sequence ending with a configuration which is II's turn or α is an infinite sequence such that the shallowest variable in $inf(\alpha)$ has parity of μ . II wins in α iff α is* maximal *and, either α is a finite sequence ending with a configuration which is I's turn or α is an infinite sequence such that the shallowest variable in $inf(\alpha)$ has parity of ν.*

A strategy for a player is a collection of decisions by which he/she chooses the next move. A player has a *winning strategy* if there is a strategy for him by which, no matter how his opponent acts, he always can win the game. A *history-free strategy* is a strategy in which the player makes his/her decision based only on the current configuration.

When model-checking games are used as a diagnostic routine, two *physical* players are involved: computer and user. I and II will be referred as logical players, which can be seen as the roles the computer and the user choose to play. We require that the schema for model-checking game should be *fair*, i.e., if the answer to the model-checking problem is positive, then, II has a strategy to win each and every play no matter which moves I chooses, otherwise I has a winning strategy. Theorem 6.3.2 shows the fairness of the model-checking game defined above.

Theorem 6.3.2 *[43, 33] Given a Kripke structure $T = \langle S, s_0, \rightarrow, \mathcal{V} \rangle$ and a μ-calculus equation system $\mathcal{E}_\mu$, if $s_I \in [\![\mathcal{E}_\mu]\!]_T(X_1)$, then I has a history-free winning strategy in model-checking game $\Omega(T, \mathcal{E}_\mu)$, otherwise II has a history-free winning strategy.*

Since the computer wants to convince the user of the correctness of its model-checking result, it will play the role in favor of model-checking result. That is, if the result is positive, then the computer will act as II, otherwise it plays as I. The user takes the other role. If the model-checking result is correct, then by Theorem 6.3.2 the computer has a winning strategy. Now the question is how to synthesize this winning strategy. We will show that this can be constructed from support sets.

Let $\mathcal{E} = \Pi(\mathcal{E}_\mu, T)$ be the boolean equation system encoding the μ-calculus model-checking problem $\langle T, \mathcal{E}_\mu \rangle$ and $\Gamma = \langle r, \langle s_I, x_1 \rangle, \Xi \rangle$ be a support set for $\mathcal{E}$. Γ may be computed by, for example, the enhanced *LAFP* described in Chapter 4. A strategy W_Γ for computer can be synthesized as below.

We first investigate the case that Γ is positive, in which the computer acts as II. Assume that the current configuration $\langle s, X \rangle$ is II's turn and $\sigma X = f$ is the equation for X in $\mathcal{E}_\mu$. Note that a pair of a state in T and a variable in $\mathcal{E}$ can be interpreted either as a configuration in the game $\Omega(T, \mathcal{E}_\mu)$ or as a variable of $\mathcal{E}$, we have,

1. If $f \equiv \langle\rangle X'$. II chooses a successor s' of s such that such that $\langle s', X'\rangle \in \Xi(\langle s, X\rangle)$.

2. If $f \equiv \bigvee \mathcal{X}'$. II chooses a $X' \in \mathcal{X}'$ such that $\langle s, X'\rangle \in \Xi(\langle s, X\rangle)$.

In other words, II always chooses the next configuration from the dependency set of current configuration. Lemma 6.3.3 shows that this strategy is feasible in the sense that it provides at least one valid move for II, regardless of the history of play.

Lemma 6.3.3 *Let ρ be a play under the strategy W_Γ, then all the configurations of ρ is covered by Ξ.*

Proof: This can be shown by induction on ρ. The lemma holds trivially on the initial configuration. Suppose that $\rho' = \langle s_I, x_1\rangle \cdots \langle s, X\rangle\langle s', X'\rangle$ is a prefix of ρ and the property holds up to and including $\langle s, X\rangle$. We can show that the lemma still holds at $\langle s', X'\rangle$ regardless of who is the current player. If II is the current player, then by the above strategy the $\langle s', X'\rangle$ II chooses must be in the dependency set of $\langle s, X\rangle$. Now assume that I is the current player, i.e., the equation for X in $\mathcal{E}_\mu$ has the form either $\sigma X = []X'$ or $\sigma X = \bigwedge \mathcal{X}'$. Therefore the right-hand side of $\langle s, X\rangle$ in $\mathcal{E}$ has the form $\wedge Y$. Note that $Y \subseteq \Xi(\langle s, X\rangle)$ by Property I of Definition 3.1.1 and $\langle s', X'\rangle \in Y$ by Definition 6.3.1. We have $\langle s', X'\rangle \in \Xi(\langle s, X\rangle)$, hence $\langle s', X'\rangle$ is also covered by Ξ due to Property II of Definition 3.1.1. ■

Constructing a winning strategy for the computer in the case of negative support set is similar. the computer (as I) will choose the next configuration as an element of the dependency set of current configuration. Theorem 6.3.4 states that the above approach will yield a winning strategy.

Theorem 6.3.4 *W_Γ is a history-free winning strategy for computer.*

Proof: We consider only the case where $r = \mathbf{1}$. The case $r = \mathbf{0}$ can be handled similarly. Let ρ be an *maximal* play under W_Γ. We need to show II wins ρ. There are two cases,

1. ρ is a finite sequence ending with $\langle s, X\rangle$. We use contradiction to show that $\langle s, X\rangle$ can only be I's turn. Suppose that it is II's turn, By Lemma 6.3.3, ρ is also a path along $\underset{\Gamma}{\rightarrow}$, hence $\langle s, X\rangle$ is covered by Ξ. There are two cases, depending on the right-hand side of X in $\mathcal{E}_\mu$.

   (a) $rhs(X) \equiv \vee\mathcal{X}'$. However, $\mathcal{X}'$ cannot be empty because, if so, no choice of $\Xi(\langle s, X\rangle)$ could make the right-hand side of $\langle s, X\rangle$ be true, which contradicts Property I of definition 3.1.1. Therefore, there is at least one $\langle s, X'\rangle$ such that $X' \in \mathcal{X}'$ and $\langle s, X'\rangle \in \Xi(\langle s, X\rangle)$. W_Γ will allow II to choose $\langle s, X'\rangle$ as the next configure, hence ρ can not be *maximal*.

   (b) $rhs(X) \equiv \langle\rangle X'$. s should have at least one successor because, if not, the right-hand side of $\langle s, X\rangle$ in $\mathcal{E}$ will be $\mathbf{ff}$ and no choice of $\Xi(\langle s, X\rangle)$ could make it be true, which contradicts Property I of definition 3.1.1. Therefore, there is at least one $\langle s', X\rangle$ such that $s \rightarrow s'$ and $\langle s', X\rangle \in \Xi\langle s, X\rangle$. W_Γ will allow II to choose $\langle s', X\rangle$ as the next configure, hence ρ cannot be *maximal*.

   Thus, $\langle s, X\rangle$ is I's turn and II shall win ρ because ρ is a maximal but finite play.

2. ρ is infinite. By Lemma 6.3.3, ρ is also a path along $\underset{\Gamma}{\longrightarrow}$. By III of definition 3.1.1, the shallowest node(s) in $inf(\rho)$ can only have the parity of ν, hence II wins ρ.

■

Part II

Evidence-Based Equivalence Checking

Chapter 7

Preliminaries

7.1 Transition System and Equivalence Relations

In equivalence checking, we are concerned with whether the behavior of two processes are related by some semantics equivalence. Traditionally, this setting focuses on the behavior of two processes in terms of their transition relations instead of the properties holding at each state. Specifically, processes are modeled as *labeled transition system* (LTS), , in which each transitions is decorated by an actions. Formally a labeled transition system is a triple $\langle S, A, \rightarrow \rangle$, where S is the set of states, A is the set of actions which contains visible actions $A - \{\tau\}$ and an internal action τ, and $\rightarrow \subseteq S \times A \times S$ is the transition relation. We write $s \xrightarrow{a} s'$ in lieu of $\langle s, a, s' \rangle \in \rightarrow$, and denote $s \xrightarrow{\bullet} = \{a \mid s \xrightarrow{a}\}$ for the set of initial actions of s. We write $s \xrightarrow{a}$ if $a \in s \xrightarrow{\bullet}$, and $s \not\xrightarrow{a}$ otherwise. We denote $s \xrightarrow{a} \bullet$ for the set $\{s' \mid s \xrightarrow{a} s'\}$. We lift the notation of $\rightarrow$ to sets of states straightforwardly: $C \xrightarrow{a} C'$ iff $C' = \{s' \mid \exists s \in C. s \xrightarrow{a} s'\} \neq \emptyset$. A state s is stable if $s \not\xrightarrow{\tau}$. A LTS P is *deterministic* if $|s \xrightarrow{a} \bullet| \leq 1$ for any state s and action a. Otherwise, it is *non-deterministic*.

The weak (observational) transition relation $\Rightarrow$ is defined as below. Let $\beta \in (A - \{\tau\})^*$ be a sequence of visible actions, then:

1. $\stackrel{\epsilon}{\Rightarrow} = \stackrel{\tau^*}{\longrightarrow}$, where $\stackrel{\tau^*}{\longrightarrow}$ is the transitive and reflexive closure of $\stackrel{\tau}{\longrightarrow}$.

2. $\stackrel{a\beta'}{\Rightarrow} = \stackrel{\epsilon}{\Rightarrow} \circ \stackrel{a}{\longrightarrow} \circ \stackrel{\beta'}{\Rightarrow}$. where $\circ$ denotes relational composition.

We extend the notion of $\Rightarrow$ to sets of states: $C \stackrel{\beta}{\Rightarrow} C'$ iff $C' = \{s' \mid s \stackrel{\beta}{\Rightarrow} s'\} \neq \emptyset$. We refer to $C^\epsilon = \{s' \mid s \stackrel{\epsilon}{\Rightarrow} s'\}$ as C's ϵ-closure. β is a trace of s_0 if there exists s' such that $s_0 \stackrel{\beta}{\Rightarrow} s'$. The language of $s_0 \in S$, written as $L(s_0)$, is defined as all the traces of s_0.

We now define semantics equivalence which relate processes (states in a transition system) on the basis of their behavior. In what follows, we fix a labeled transition transition $T = \langle S, A, \rightarrow \rangle$.

Definition 7.1.1 (Π-bisimulation) *[10] Given a relation $\Pi \subseteq S \times S$, $\sim_\Pi$ is a Π-bisimulation iff $\sim_\Pi \subseteq \Pi$ and $s_0 \sim_\Pi s_1$ implies the following,*

1. $s_0 \stackrel{a}{\longrightarrow} s_0' \Leftrightarrow \exists s_1'.s_1 \stackrel{a}{\longrightarrow} s_1' \wedge s_0' \sim_\Pi s_1'$.

2. $s_1 \stackrel{a}{\longrightarrow} s_1' \Leftrightarrow \exists s_0'.s_0 \stackrel{a}{\longrightarrow} s_0' \wedge s_1' \sim_\Pi s_0'$.

Note that if $\Pi = S \times S$ then a Π-bisimulation $\sim_\Pi$ becomes the bisimulation $\sim$ in the usual sense [36].

Definition 7.1.2 (Language Equivalence) *Let s_0 and s_1 be two states of T. $s_0 =_{may} s_1$ iff $L(s_0) = L(s_1)$.*

$=_{may}$ is also called *language equivalence.*

The notion of failure equivalence [23] relies on the following definition,

1. A state s *refuses* $B \subseteq A - \{\tau\}$ if $|B| < \infty$ and for all $b \in B$, $s \stackrel{b}{\not\rightarrow}$. A set of states C *refuses* B if there exists $s \in C$ which refuses B.

2. A state s is *divergent*, written as $s \uparrow$, if and only if s is on a τ loop. s is divergent on a sequence of visible actions, written $s \uparrow \beta$, if and only if there exists a prefix β' of β and a state s' such that $s \overset{\beta'}{\Rightarrow} s'$ and $s \uparrow$. We write $s \downarrow \beta$ if $s \uparrow \beta$ is not true and say that s converges on β in this case. A set of states C diverges on β, written $C \uparrow \beta$, if and only if there exists an $s \in C$ such that $s \uparrow \beta$. $C \downarrow \beta$ iff such s doesn't exist.

3. Let β be a sequence of visible actions and $B \subseteq A$ be finite. Then $\langle \beta, B \rangle$ is a failure for s if either $s \uparrow \beta$ or there is a q' such that $q \overset{\beta}{\Rightarrow} q'$ and q' refuses B. The length of failure $\langle \beta, B \rangle$ is defined as the length of β. $\langle \beta, B \rangle$ is a *shortest* failure for s if it has the minimal length among all the failures for s.

We write $\mathcal{F}(s)$ for the set of all the failures for s. Failure equivalence $=_{must}$ is defined as follows.

Definition 7.1.3 (Failure Equivalence) *Let s_0 and s_1 be two states of T. Then $s_0 =_{must} s_1$ iff $\mathcal{F}(s_0) = \mathcal{F}(s_1)$.*

7.2 Computing Other Equivalences as Bisimulation

[11] showed that many other equivalences, including language and failure equivalences, may be reduced to Π-bisimulation. The idea is to transform original transition system to a new one that they call a *reference transition system.* Computing an equivalence relation on the original system is then equivalent to compute a Π-bisimulation on the corresponding reference transition system. [11] also suggests a family of reference transition systems, called *acceptance graphs*, for language equivalence and failure equivalence. The definition of acceptance graphs relies on the notion of acceptance set. The acceptance set of

s after β is defined as below.

$$Acc(s, \beta) = \{s' \xrightarrow{\bullet} \mid s \stackrel{\beta}{\Rightarrow} s' \wedge s' \stackrel{\tau}{\not\rightarrow}\}$$

The acceptance set for a set of states is defined as the union of the acceptance sets of its members.

$$Acc(C, \beta) = \bigcup_{s \in C} Acc(s, \beta)$$

Acceptance sets can be minimized in the following manner,

$$min\ \mathcal{A} = \{B \in \mathcal{A} \mid \neg\exists B' \in \mathcal{A}.B' \subset B\}$$

So the minimized version of $\mathcal{A}$ contains only the elements of an acceptance set $\mathcal{A}$ having no proper subsets that are also elements of $\mathcal{A}$. We say two acceptance sets are *equal* iff their minimized versions are same.

Finally, an *acceptance graph* is a deterministic transition system in which each state has the form $t = \langle Q, b\rangle$ such that Q is a set of states and b is a boolean value. We may also refer to b as t's *closed* field, written as $t.closed = b$. In addition, each state t is labeled with a set of set of actions, denoted as $t.acc$. Particularly, an *acceptance graph* for failure equivalence for transition system T is given in Definition 7.2.1.

Definition 7.2.1 *Given a LTS* $T = \langle S, A, \rightarrow\rangle$, *its reference acceptance graph* $SL(T) = \langle S_r, A - \{\tau\}, \rightarrow_r\rangle$ *is given as below,*

1. $S_r = \{\langle Q, b\rangle \in 2^S \times \{\mathbf{0}, \mathbf{1}\} \mid Q = Q^\epsilon \wedge (b = \mathbf{1} \rightarrow Q \downarrow)\}$

2. *For* $t = \langle Q, b\rangle \in S_r$,

$$t.acc = \begin{cases} \emptyset & \textit{if } b = \mathbf{0} \\ min\ Acc(Q, \epsilon) & \textit{Otherwise} \end{cases}$$

3. *For* $t_1 = \langle Q_1, b_1\rangle$ *and* $t_2 = \langle Q_2, b_2\rangle$, $t_1 \xrightarrow{a}_r t_2$ *exactly when the following properties hold.*

(a) $a \neq \tau$

(b) $(Q_1 \overset{a}{\Rightarrow} \bullet)^\epsilon = Q_2$

(c) $(b_1 = \mathbf{1}) \wedge (b_2 = \mathbf{0} \Rightarrow Q_2 \uparrow)$

Clearly $SL(T)$ is deterministic. We may define as below a mapping function g which relates states in original system with states in $SL(T)$: $g(q) = \langle \{q\}^\epsilon, \{q\} \downarrow \rangle$.

Lemma 7.2.2 *[11] $SL(T) = \langle S_r, T_r, \rightarrow_r \rangle$ enjoys the following properties.*

(i). Let $s_r = \langle Q, b \rangle \in S_r$ be a reference state, then s_r has no outgoing transition from s_r if $b = \mathbf{0}$, i.e., there is no outgoing transitions from a reference state which is related to a divergent state in original system.

(ii). Let $s \in S$ be a state in original system such that $s \downarrow \beta$, then $\beta \in L(s)$ iff $\beta \in L(g(s))$, where $g(s)$ is the state related to s in $SL(T)$.

(iii). Let $s \in S$ be a state in original system such that $s \downarrow \beta$, $\beta \in L(s)$, and $g(s) \overset{\beta}{\rightarrow} s_r$, then $s_r.acc = min\ Acc(s, \beta)$.

[11] has shown that checking failure equivalence on a LTS can be reduced to check Π-bisimulation on its reference acceptance graph.

Theorem 7.2.3 *[11] $s_0 =_{must} p_1$ if and only if $g(s_0) =_\Pi g(s_1)$, where $g(s)$ is the state related to s in $SL(T)$, and $\Pi = \{\langle t_1, t_2 \rangle \mid t_1.acc = t_2.acc\}$.*

The acceptance graph for checking language equivalence is obtained by substituting

$$S_r = \{\langle Q, \mathbf{0} \rangle \mid Q = Q^\epsilon\}$$

for (1) of Definition 7.2.1. We refer to the resulting acceptance graph as $D(T)$. Since all the *closed* fields are $\mathbf{0}$, By (2) of Definition 7.2.1 all the *acc* fields are empty. Basically $D(T)$ is just the deterministic version of T on the account of observation behaviors. If we define $g(s) = \langle\langle\{s\}^\epsilon\rangle, \mathbf{0}\rangle$ as the mapping function, then $L(s) = L(g(s))$ for any $s \in S$. [10] shows that checking language equivalence on T can be reduced to computing Π-bisimulation on $D(T)$.

Theorem 7.2.4 *[10] $s_0 =_{may} s_1$ if and only if $g(s_0) \sim_\Pi g(s_1)$, where $g(s)$ is the state related to s in $D(T)$, and $\Pi = \{\langle t_1, t_2\rangle \mid t_1.acc = t_2.acc\}$.*

Practically, if we are interested in only the relation of two states s_0 and s_1, we only need to construct the part of $SL(T)$ or $D(T)$ reachable from $g(s_0)$ or $g(s_1)$.

Chapter 8

Partition Refinement Trees

This chapter introduces *partition refinement trees* as *abstract proof structures* for equivalence checking. Particularly we will introduce an extension of partition refinement tree, Kernel-Auxiliary partition refinement tree, which can serve as the evidence for both equivalence checking and preordering checking.

8.1 Definition

A partition refinement tree is a *decorated* binary tree. A binary tree D is prefix-closed subset of $\{0,1\}^*$, where λ, the empty sequence, is the root of the tree. An edge is a tuple $\langle d, d.i\rangle \in E$ such that $d, d.i \in D$. d is an internal node if d is the non-trivial prefix of some $d' \in D$, and it is a leaf otherwise. A binary tree D is complete if both $d.0$ and $d.1$ are in D for every internal node d. A forest of binary trees is the set of binary trees. We shall write $k\beta$ for the node β in the k-th subtree of a forest.

A partition refinement tree is defined with respect to a transition system. In what follows, we fix the transition system $T = \langle S, A, \rightarrow\rangle$. We refer to $P \subseteq S$ as *stable* with respect to $P' \subseteq S$ on action $a \in A$ iff either $Q = P$ or $Q = \emptyset$, where $Q = \{s \in P \mid \exists s' \in P'. s \xrightarrow{a} s'\}$. $P \subseteq S$ is *stable* with respect to P' iff P

is stable with respect to P' on every action $a \in A$.

Definition 8.1.1 *A partition refinement tree (*PRT*) $\gamma = \langle D, \ell, \ell_s, \ell_a, T\rangle$ is a labelled complete binary tree D defined with respect to a* LTS *$T = \langle S, A, \rightarrow\rangle$, where $\ell : D \rightarrow (2^S - \emptyset)$ labels each node with a non-empty subset of S. $\ell_a : D \rightarrow A$ and $\ell_s : D \rightarrow D$ associates each* internal node *with an action and a node, respectively; and, ℓ_a and ℓ_s satisfies the following properties,*

1. *$\ell(\lambda) = S$.*
2. *For each internal node d, $\ell(d.0) = \{s \in \ell(d) \mid \exists s' \in \ell_s(d).s \stackrel{\ell_a(d)}{\rightarrow} s'\}$, and $\ell(d.1) = \ell(d) - \ell(d.0)$.*
3. *There is a well-founded order $<_\omega$ on D such that, for any $d \in D$, $d <_\omega d.i$ and $d \geq_\omega \ell_s(d)$.*

A node d is stable *with respect to node d' on action a iff $\ell(d)$ is stable with respect to $\ell(d')$ on a. A partition refinement tree is* stable *if, for any leaves q_1 and q_2, $\ell(q_1)$ is stable with respect to $\ell(q_2)$ on every action.*

We refer to $\ell_s(q)$ and $\ell_a(q)$ as the *splitter* and *splitting action* of q. When clear from context we avoid the formal distinguishing between a node q and the set $\ell(q)$ it represents. Intuitively, when q splits with respect to $\ell_s(q)$ and $\ell_a(q)$, its left child $\ell(q.0)$ contains those states of q which can make a $\ell_a(q)$ transition to some states of $\ell_s(q)$, and the right child $q.1$ contains the rest of states in q which are not able to do so.

Most of properties of PRTs are clear to those readers who are familiar with partition refinement-based algorithms (cf. [25]), except 3. The *well-founded order* ensures that the final partition represented by leaves is not "finer" than the partition induced by bisimulation. We need some notions before giving a formal proof for this.

A partition Γ of S is stable if every $P \in \Gamma$ is stable with respect to every block $P' \in \Gamma$. A partition Γ is coarser than Γ', written $\Gamma' \sqsubseteq \Gamma$, iff for each $P \in \Gamma$, there exists a $P' \in \Gamma'$ such that $P' \subseteq P$.

Lemma 8.1.2 *[40]*

1. *There is only one coarsest stable partition for any transition system, and*

2. *The coarsest stable partition is just the partition induced by the bisimulation relation $\sim$.*

Theorem 8.1.3 *Given a stable partition refinement tree $\gamma = \langle D, \ell, \ell_s, \ell_a, T\rangle$, $\Gamma_\gamma = \{\ell(q) \mid q$ is a leaf$\}$ is the partition induced by the bisimulation $\sim$.*

Proof: Let Γ be the coarsest stable partition. We need to show $\Gamma_\gamma = \Gamma$. Clearly by Definition 8.1.1 $\Gamma_\gamma \sqsubseteq \Gamma$. We only need to show that $\Gamma \sqsubseteq \Gamma_\gamma$.

We choose to prove a stronger condition: γ respects partition Γ, i.e., each node of γ is an union of some blocks in Γ. We use contradiction to show this. Let w be the well-founded order for splitters in D. Clearly the first node on $<_\omega$, the root, satisfies this. Assume that $d.k$ is the first node on $<_\omega$ which violates the condition. We consider the case $k = 0$, while the case that $k = 1$ can be proven similarly.

The assumption that $d.0$ is not the union of some classes in Γ implies the existence of s_i and s_j such that $s_i \in \ell(d.0)$, $s_j \notin \ell(d.0)$, and $\{s_i, s_j\} \subseteq P$ for some $P \in \Gamma$. Since d shall appear before $d.i$ on $<_\omega$ and $s_i \in \ell(d.0) \subseteq \ell(d)$, s_i and s_j are both in $\ell(q)$ and $s_j \in \ell(d.1)$. Moreover, by Lemma 8.1.2, $\{s_i, s_j\} \subseteq P$ for some $P \in \Gamma$ implies $s_i \sim s_j$.

By Property 2 of Definition 8.1.1, there is a state $s'_i \in \ell(\ell_s(d))$ such that $s_i \overset{\ell_a(d)}{\rightarrow} s'_i$. Nevertheless, $s_i \sim s_j$ implies the existence of a s'_j such that $s_j \overset{\ell_a(d)}{\rightarrow} s'_j$ and $s'_i \sim s'_j$. Since d's splitter $\ell_s(d)$ shall appear before $d.k$ on $<_\omega$, $\ell_s(d)$ respects Γ by our assumption. It follows that s'_i and s'_j should be both in

$\ell(\ell_s(d))$ because $s'_i \in \ell(\ell_s(d))$ and $s'_i \sim s'_j$. Therefore, s_j should also be in $\ell(d.0)$, but this contradicts the fact that $s_j \notin \ell(d.0)$. Thus, γ shall respect Γ and $\Gamma_\gamma = \Gamma$. ∎

Theorem 8.1.4 *Given a labeled transition system $T = \langle S, A, \rightarrow \rangle$ and two states $s_1, s_2 \in S$, then,*

1. *$s_1 \sim s_2$ iff there is a stable PRT γ for T such that s_1 and s_2 are in the same leaf.*

2. *$s_1 \nsim s_2$ iff there is a PRT γ for T such that s_1 and s_2 are in different node.*

Proof: We postpone the proof to the next chapter, in which we actually show how such a partition refinement tree may be constructed by an adapted bisimulation checker. ∎

Theorem 8.1.4 shows the soundness and completeness of partition refinement tree as the evidence for justifying bisimulation-checking results.

Remarks In the setting of Π-bisimulation, the initial partition may contain more than one equivalence classes; hence *partition refinement tree* becomes a *forest*. Each tree in this *partition refinement forest* is a partition refinement tree, with one modification: the root of each tree now represents a unique block in the partition induced by Π instead of the set of all the states. One may prove in a similar manner that the leaves of a stable partition refinement forest represent the partition induced by $\sim_\Pi$.

8.2 Paige-Tarjan Partition Refinement Tree

Definition 8.2.1 *γ is a Paige-Tarjan partition refinement tree (PT-PRT) iff it is a PRT which satisfies the following additional properties.*

4. For every node d with $q.k = \ell_s(d)$ and $a = \ell_s(d)$, let $q.(1-k)$ be the sibling of $q.k$ and d' be the parent of d. Then one of the following holds,

a. $\ell_s(d') = q.(1-k)$ and $\ell_a(d') = a$, or d' is stable with respect to $q.(1-k)$ on a.

b. For the left child $d.0$ of d, $\ell_s(d.0) = q.(1-k)$ and $\ell_a(d.0) = a$, or $d.0$ is stable with respect to $q.(1-k)$ on a. For the right child $d.1$, $d.1$ is stable with respect to $q.(1-k)$.

5. For every node d with $q = \ell_s(d)$ and every action $a \in A$, Let q' be an ancestor of q. Then, there exists an ancestor d' of d such that either $\ell_a(d') = a$ and $\ell_s(d') = q'$, or d' is stable to q' on a.

Paige-Tarjan partition refinement trees are introduced to describe the special category of PRTs generated by Paige-Tarjan algorithm [40, 19]. As we will see shortly in Chapter 10, identifying these features helps expedite verifying trees produced by implementations of Paige-Tarjan algorithm, so that the time spent on verifying results won't exceed the time cost by checkers, per general requirement 4 for the choice of abstract proof structure in Chapter 1.

Before we give some intuition on (4) and (5), we need briefly to review the Paige-Tarjan algorithm. In the Paige-Tarjan algorithm, siblings always work consecutively as splitters. To better formulate the description of this mechanism, [19] elaborates the idea of *compound splitter* in [40] by introducing a tree structure for compound splitters, which can be seen as a subtree of refinement tree and often represented by just its root. We refer to traditional splitters as *simple* splitters. The result of splitting can be seen as applying the children of a compound splitter consecutively.

The algorithm maintains the following vital invariant:

(A). If d is split by a compound splitter rooted at p on action a, then the node d itself is stable with respect to the node p on a.

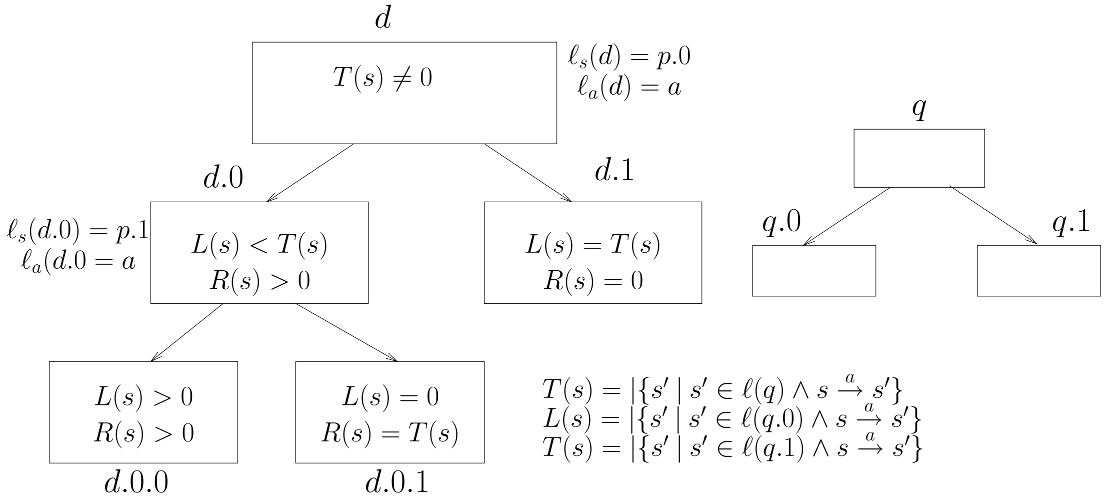


Figure 4: Three-way splitting

This invariant is initially established by splitting the root by itself as a simple splitter. Figure 4 shows what happens when d split by compound splitter q and action a. The resulting leaves are irrelevant to the choice which child of q acts first as a splitter. W.l.o.g., we assume that $q.0$ performs splitting before $q.1$ does so. Figure 4 also lists the conditions of states must hold in each stage of splitting. $T(s)$, $L(s)$, and $R(s)$ recodes the number of transitions from a state s to the root, left child, and the right child of the compound splitter q, respectively. Because d has already been stable to q on a by (A), $d.1$ won't be split further by $q.1$ on action a. The splits yields three new leaves as below instead of four, which gains its name *three-way splitting*,

1. $d.0.0$ holds those states having transitions to both $q.0$ and $q.1$.

2. $d.0.1$ holds those states having transitions to $q.0$ but not to $q.1$.

3. $d.1$ holds those states having transitions to $q.1$ but not to $p.0$.

Splitting can be preformed efficiently by associating with each compound splitter a counter table, which records for every state the number of transitions for it to the root of the compound splitter. Let $C(q)$ denote q's counter table such that $C(q)[s]$ is the counter for s. To determine to which leaf a state $s \in \ell(d)$ should go, we only need to compute $C(p.i)$ for one child of q. The trick is that we can always pick up the smaller child of q. This helps reduce the time complexity to $O(|T| \cdot log(|S|))$ because every time we only care about incoming transitions to less than half states of the compound splitter. [19] explains some practical issues related to Paige-Tarjan algorithm [40].

(4) of Definition 8.3.3 is intended to *simulate* the idea of *compound splitters* by requesting both siblings shall act as splitters consecutively. The sibling of the splitter for node d can be used to split either d's parent or its children during three-way splitting, so (4) tries to cover both scenarios . (5) requests that all the ancestors of q have been tried before q can act as a splitter. In other words, (5) preserves invariant (A) for the evolution of tree.

8.3 Kernel-Auxiliary Partition Refinement Tree*

A partition refinement tree can be extended to encode the evidence for preordering checking. In preordering checking, we are interested in whether one process is less than the other with respect to some preordering relation. The simulation relation we are about to introduce is a preordering relation.

Definition 8.3.1 (simulation) *Let $T = \langle S, A, \rightarrow \rangle$ be a* LTS*, a preordering relation $R \subseteq S \times S$ is a simulation over S if, for every $\langle s_1, s_2 \rangle \in R$ and $a \in A$, whenever $s_1 \xrightarrow{a} s_2$ then there is a s_2' such that $s_2 \xrightarrow{a} s_2'$ and $\langle s_1', s_2' \rangle \in R$.*

It is easy to establish that there is a maximum simulation, $\preceq$, which is the union of all the simulations on S. We say that s_2 can simulate s_1 iff $s_1 \preceq s_2$.

Theorem 8.3.2 *[48] Let $\langle S, A, \rightarrow \rangle$ be a* LTS*, with $s_1, s_2, s_3 \in S$. Then:*

1. *If* $s_1 \sim s_2$ *and* $s_2 \preceq s_3$*, then* $s_1 \preceq s_3$*.*

2. *If* $s_1 \preceq s_2$ *and* $s_2 \sim s_3$*, then* $s_1 \preceq s_3$*.*

Theorem 8.3.2 shows that bisimulation respects the maximum simulation in the sense that the states in the same bisimulation classes can be simulated by the same set of states. This observation inspires the idea of adapting PRT for preordering checking.

Definition 8.3.3 *Given a LTS* $T = \langle S, A, \rightarrow \rangle$*, a kernel-auxiliary partition refinement tree (KA-PRT)* $\gamma = \langle D, \ell, \ell_s, \ell_a, T \rangle$ *is a labeled complete binary tree* D *defined with respect to a LTS* $T = \langle S, A, \rightarrow \rangle$*.* $\ell : D \rightarrow (2^S - \emptyset)$ *labels each node with a tuple* $\langle K, D \rangle$*, where* $K, D \subseteq S$ *and* $K \neq \emptyset$*.* $\ell_a : D \rightarrow A$ *and* $\ell_s : D \rightarrow D$ *associate each* internal node *with an action and a node, respectively. In addition,* ℓ_a *and* ℓ_s *satisfy the following properties,*

1. $\ell(\lambda) = \langle S, S \rangle$*.*

2. *For each internal node* d*, let* $\langle K, D \rangle = \ell(d)$ *and* $\langle K_s, D_s \rangle = \ell(\ell_s(d))$*. Then the children of* d *are defined as below,*

   (a). $\ell(d.0) = \langle K_0, D_0 \rangle$ *is the left child of* d*, such that* $K_0 = \{s \in K \,|\, \exists s' \in K_s . s \xrightarrow{\ell_a} s'\} \neq \emptyset$*,* $D_0 = \{s \in D \mid \exists s' \in D_s, s \xrightarrow{\ell_a} s'\}$*, and* $K_0 \neq K \Rightarrow D_0 \neq D$*.*

   (b). $\ell(d.0) = \langle D - K_0, D \rangle$ *is the right child of* d*.*

3. *There is a well-founded order* $<_\omega$ *on* D *such that, for any* $d \in D$*,* $d <_\omega d.i$ *and* $d \geq_\omega \ell_s(d)$*.*

d is stable with respect to d' on a iff there doesn't exist a KA-PRT on which d is a node with d' and a as its splitter and splitting action respectively. A kernel-auxiliary partition refinement tree is stable *if, for any leaves q_1 and q_2, $\ell(q_1)$ is stable with respect to $\ell(q_2)$ on any action.*

We will show that a KA-PRT gives the evidence for both simulation checking and bisimulation checking. Before we go any further, we need to introduce some notations.

Definition 8.3.4 *Let $T = \langle S, A, \rightarrow \rangle$ be a labeled transition system. Then,*

1. *A* kernel-auxiliary pair *(KA-pair) has form $\langle B, X \rangle$, where $B \subseteq S$ is nonempty and $X \subseteq S$. We write $\langle B, X \rangle \subseteq \langle B', X' \rangle$ if $B \subseteq B'$ and $X \subseteq X$. We often refer to B as the* kernel set *of $\langle B, X \rangle$ and X as the* auxiliary set*.*

2. *A set Γ of KA-pairs is a* kernel-auxiliary partition *(KA-partition) on S if $\Gamma' = \{B \mid \langle B, X \rangle \in \Gamma\}$ is a partition on S.*

3. *A KA-partition P* refines *KA-partition P' ($P \trianglelefteq P'$) if for every $\langle B, X \rangle \in P$ there is a $\langle B', X' \rangle \in P'$ such that $\langle B, X \rangle \subseteq \langle B', X' \rangle$.*

4. *KA-pair $\langle B, X \rangle$ is* stable *with respect to KA-pair $\langle B', X' \rangle$ on action $a \in A$ if either $\forall s \in B.(\nexists s' \in B'.(s \xrightarrow{a} s'))$, or $\forall s \in B.(\exists s' \in B'.(s \xrightarrow{a} s'))$ and $\forall s \in X.(\exists s' \in X'.(s \xrightarrow{a} s'))$. A KA-partition P is stable to KA-pair $\langle B', X' \rangle$ on action a if every $\langle B, X \rangle \in \Gamma$ is stable with respect to $\langle B', X' \rangle$. KA-partition Γ is stable with respect to KA-partition Γ' if Γ is stable to every KA-pair in P' on any action $a \in A$. KA-partition P is self-stable if it is stable with respect to itself.*

Note that if Γ is a self-stable KA-partition on S and $\langle B, X \rangle, \langle B', X' \rangle \in P$, then either no state in B has a transition into B', or every state in B has a transition into B' and every state in X has a transition into X'. When Γ is a self-stable KA-partition, the set $\{B \mid \langle B, X \rangle \in P\}$ is a self-stable partition.

Every KA-partition P defines a relation $R(P) \subseteq S_1 \times S_2$ given by: $\langle s_1, s_2 \rangle \in R(P)$ if and only if there exists $\langle B, X \rangle \in P$ such that $s_1 \in B$ and $s_2 \in X$.

Theorem 8.3.5 *[48] Let $T = \langle S, A, \rightarrow \rangle$ be a* LTS *with $s_1, s_2 \in S$ and $s_1 \in S$. Then there is a unique coarsest self-stable KA-partition $P_{\max}$ on S, and $s_1 \preceq s_2$ if and only if $\langle s_1, s_2 \rangle \in R(P_{\max})$.*

Theorem 8.3.5 shows that the relational coarsest KA-partition implies maximum simulation relation. Therefore, computing simulation relation can be reduced to solving the relational coarsest KA-partition problem defined as below,

Given: Transition systems $T = \langle S, A, \rightarrow \rangle$ with $|S| < \infty$.

Compute: The coarsest self-stable partition P on S.

The statement of this problem makes no reference to partitions of the state set of the "upper" transition system. The following corollary indicates that auxiliary sets can be efficiently represented as unions of bisimulation-equivalence classes.

Corollary 8.3.6 *Let $T = \langle S, A, \rightarrow \rangle$ be a labeled transition system, let $P_{\max}$ be the relational coarsest KA-partition on T. and let Q be the coarsest self-stable partition over S. Then for any $\langle B, X \rangle \in P_{\max}$ and $C \in Q$, either $X \subseteq C$ or $X \cap C = \emptyset$.*

Theorem 8.3.7 shows that a stable KA-PRT implies the coarsest self-stable partition.

Theorem 8.3.7 *Given a stable partition refinement tree $\gamma = \langle D, \ell, \ell_s, \ell_a, T \rangle$, $\Gamma_\gamma = \{\ell(q) \mid q$ is a leaf$\}$ is the relational coarsest KA-partition.*

Proof: Let Γ be the relational coarsest KA-partition. We need to show $\Gamma_\gamma = \Gamma$. By Definition 8.3.3, the fact that γ is stable implies that $\Gamma_\gamma \trianglelefteq \Gamma$. We only need to show that $\Gamma \trianglelefteq \Gamma_\gamma$.

We choose to prove a stronger condition: γ respects partition Γ during its evolution, i.e., γ holds an invariant that each node of γ is the union of some sets in Γ. We use contradiction to show this. Let $<_\omega$ be the well-founded order for splitters. Clearly the least node on $<_\omega$, the root, satisfies this. Assume that $d.k$ is the least node on $<_\omega$ which violates the invariant. We consider the case $k = 0$, while the case $k = 1$ can be proven similarly.

Let $\langle K_0, D_0 \rangle = \ell(d.0)$. There are two ways in which $d.0$ may violate the invariant. In the first case, there are two s_i and s_j such that $s_i \in K_0$ and $s_j \notin K_0$, but $\{s_i, s_j\} \subseteq B$ for some $\langle B, X \rangle \in \Gamma$. The dispute for this case is similar to that in the proof of Theorem 8.1.3. In the second case, in which there are two s_i and s_j such that $s_i \in K_0$ and $s_j \notin D_0$, but $s_i \in B$ and $s_j \in X$ for some $\langle B, X \rangle \in \Gamma$. Let $\langle K, D \rangle = \ell(d)$ and $\langle K_1 = K - K_0, D_1 = D \rangle = \ell(d.1)$. Since $d.i$'s parent d shall appear before $d.i$ on $<_\omega$, $s_i \in K_1 \subseteq K$ and $s_j \in D = D_2$. Moreover, by Theorem 8.3.5, $s_i \in B$ and $s_j \in X$ for some $\langle B, X \rangle \in \Gamma$ implies $s_i \preceq s_j$.

Let $q = \ell_s(d)$ with $\langle B, X \rangle = \ell(q)$ and $a = \ell_a(d)$. By Property (2) of Definition 8.3.3, there is a state $s'_i \in B$ such that $s_i \xrightarrow{a} s'_i$. Nevertheless, $s_i \preceq s_j$ implies the existence of a s'_j such that $s_j \xrightarrow{a} s'_j$ and $s'_i \preceq s'_j$. Since $q <_\omega d.0$ by (3) of Definition 8.3.3. Hence p should respect Γ by the assumption. It follows that s'_i and s'_j must be in p's kernel and auxiliary sets simultaneously. Since $s_j \xrightarrow{a} s'_j$ and $s'_j \in X$, s_j must be in D_0. Therefore, the second case also cannot exist, and we proved that γ should respect Γ. ■

The idea of using KA-PRT to encode evidence for simulation checking can be understood as below. Every node of a KA-PRT represents a KA-pair $\langle K, A \rangle$, where $K, A \subseteq S$. We use K to keep an approximation of a bisimulation class, which is similar to the function of a node in a PRT, while use A to encode the set of states which may simulate the states in K.

Theorem 8.3.8 proves the soundness and completeness of $\mathsf{KA\text{-}PRT}$ as the evidence for justifying results of both simulation and bisimulation checking.

Theorem 8.3.8 *Given a labelled transition system $T = \langle S, A, \rightarrow \rangle$ and two states $s_0, s_1 \in S$, then,*

1. *$s_0 \preceq s_1$, if there exists a stable $\mathsf{KA\text{-}PRT}$ γ and a leaf d of γ with $\ell(d) = \langle K, D \rangle$, then $s_0 \in K$ and $s_1 \in D$.*

2. *$s_0 \sim s_1$, if there exists a stable $\mathsf{KA\text{-}PRT}$ γ and a leaf d of γ with $\ell(d) = \langle K, D \rangle$, then $\{s_0, s_1\} \in K$.*

3. *$s_0 \not\preceq s_1$, if there exists a $\mathsf{KA\text{-}PRT}$ γ and a node d on γ with $\ell(d) = \langle K, D \rangle$, then $s_0 \in K$ and $s_1 \notin D$.*

4. *$s_0 \not\sim s_1$, if there exists a $\mathsf{KA\text{-}PRT}$ γ and a node d on γ with $\ell(d) = \langle K, D \rangle$, then $s_i \in K$ and $s_{1-i} \notin K$ for some $i \in \{0, 1\}$.*

Proof:

The proof will be postponed to the next chapter, in which we will actually how to construct such $\mathsf{KA\text{-}PRT}$s by modifying a naive algorithm for the relational coarsest KA-partition problem. ∎

Chapter 9

Extracting Partition Refinement Trees

9.1 Extracting PRT

A partition refinement algorithm usually explicitly or implicitly constructs a partition refinement tree during its execution. All we need to do is to retain the necessary information during the execution and recover a PRT from internal data structures at the end of computation. To avoid reference to a particular checker, we will discuss the extraction procedure on a naive algorithm shown in Figure 9.1. More sophistic refinement-based algorithms such as three-way splitting algorithm [40], and the Kernel-Auxiliary partition algorithm [48], may produce refinement tree (forest) with similar modifications.

We will construct a PRT $\gamma = \langle D, \ell, \ell_s, \ell_a \rangle$ during the execution of $bisim(s_0, s_1)$. The construction maintains the following invariant for Γ.

A. *For each block P in Γ, there is a leaf d in the current configuration of γ such that $\ell(d) = P$*

Initially, γ has only one node ϵ with the label $\ell(\epsilon) = S$. Assume that P has

Procedure $bisim(s_0, s_1, T = \langle S, A, \rightarrow \rangle)$
$\Gamma = \{S\}$
while Γ is not stable **do**
Choose $P, P' \in \Gamma$ and a such that
P is not stable to P' and a.
Replace P in Γ by $P_1 = \{s \in P \mid \exists s'.s \xrightarrow{a} s'\}$ and $P_2 = P - P_1$.
if s_0 and s_1 in different blocks **then**
return **false**
return true.

Figure 5: A naive partition-refinement algorithm

been split by P' on a. Let d and d' be the current leaves related to P and P' by the above invariant. We create two children for d such that $\ell(d.0) = P_1$ and $\ell(d.1) = P_2$. Meanwhile, we also mark the splitter and splitting action for d as $\ell_s(d) = d'$ and $\ell_a(d) = a$.

Clearly the resulting PRT γ satisfies (1) and (2) of Definition 8.1.1. To see (3), we may order the nodes by its splitting time. Clearly this will yield a desired well-founded order.

To establish the complexities, we need some assumptions on the data structures used to encode PRT. We assume that the ℓ field is stored only for leaf nodes. $\ell(q)$ for an internal node q can be recovered from ℓ field of all the leaves in the subtree rooted at q. The total cost of storing ℓ is bounded by $O(|S|)$. Since the cost of storing ℓ_s and ℓ_q is constant for each node, and the number of nodes in γ is bounded by $O(|S|)$, the total cost of storing γ is bounded by $O(|S|)$. Moreover, when a leaf node q split, we just move the states in $\ell(q)$ to the ℓ field of its children, which can be done in $O(1)$ space. Therefore, the space complexity of constructing and maintaining γ is bounded by $O(|S|)$. By invariant A, the naive algorithm can use the leaves of PRT to store the current partition.

The only extra work for maintaining the PRT is to update ℓ_s and ℓ_a for the node being split, which can be done in $O(1)$. Therefore, adapting the naive algorithm to construct a PRT does not compromise its time and space complexities. In general, a partition refinement-based algorithm can be adapted to use the leaves of PRT to store the current partition. The only extra cost in this case is to keep track of the parent/children relation, which usually can be done in $O(1)$. In our study, we found that constructing PRT also won't compromise more sophisticated partition refinement-based algorithms like [40] and [48].

If two states s_0 and s_1 are not related, then the PRT generated above may not necessarily be stable. However, if $s_0 \sim s_1$, then s_0 and s_1 should be in the same leaf, and the resulting PRT must be stable. These refinement trees, for example, may be constructed by above procedure. Now we give the proof of Theorem 8.1.4 by showing how such a PRT can be built.

Proof of Theorem 8.1.4

Proof:

1. ($\Rightarrow$) Let γ be the partition refinement tree produced by the naive algorithm adapted to build PRTs. By the correctness of the naive algorithm, the final partition presented by leaves should be the partition induced by $\sim$. Since $s_1 \sim s_2$, s_1 and s_2 shall be in the same leaf. Moreover, the tree must be stable since every leaf is stable; otherwise, an unstable leaf cannot stand for an equivalence class on $\sim$ by Definition 7.1.1.

   ($\Leftarrow$) Follows from Lemma 8.1.3.

2. ($\Rightarrow$) Let γ be the partition refinement tree produced by the modified naive algorithm. By the correctness of naive algorithm, $s_0 \nsim s_1$ implies that s_0 and s_1 must be in different leaves when the algorithm terminates.

   ($\Leftarrow$) γ can be extended to a stable PRT by doing the following till every leaf is stable, where initially $<_\omega$ is the well-found order for splitters in γ.

Let d be a leaf node unstable with respect to a node d' on an action a. Then, d will have two children $d.0$ and $d.1$ such that $\ell(d.0) = \{s \in \ell(d) \mid \exists s' \in \ell(d').s \xrightarrow{a} s'\}$ and $\ell(d.1) = \ell(d) - \ell(d.1)$. Update $\ell_s(d) = d'$ and $\ell_a(d) = a$, and append d' to $<_\omega$.

Clearly the resulting tree γ' is a stable PRT with $<_\omega$ as the desired well-found orders for splitters. Since s_1 and s_2 were already in different nodes in γ, they will end up in different leaves of γ'. By Theorem 8.1.3, this implies that $s_1 \not\precsim s_2$.

■

9.2 Extracting KA-PRT*

[48] shows that computing the simulation preordering can be reduced to building the relational coarsest KA-partition, and introduces two refinement-based algorithms for simulation checking. The algorithms work by refining Kernel-Auxiliary partition. Both algorithms start from the initial partition $\{\langle S, S\rangle\}$, and continue the refinement process until the pair of states in question are proven not to be related, or the partition becomes stable.

Figure 9.2 shows the naive algorithm used in [48] to refine KA-partition. When $\langle B, X\rangle$ is split by a splitter $\langle B', X'\rangle$ on an action $a \in A$, it will be replaced by two new KA-pairs,

$$\langle B_1, X_1\rangle := \langle\{s \mid s \in B \wedge s \xrightarrow{a} B'\}, \{s \mid s \in X \wedge s \xrightarrow{a} X'\}\rangle$$

and,

$$\langle B_2, X_2\rangle := \langle B - B_1, X\rangle$$

Using the technique similar to the one used to construct a PRT in the previous section, we can construct a KA-PRT $\gamma = \langle D, \ell, \ell_s, \ell_a\rangle$ during the

Procedure $sim(s_0, s_1, T = \langle S, A, \rightarrow \rangle)$
    $\Gamma = \{\langle S, S\rangle\}$
    while Γ is unstable **do**
        Choose $\langle B, X\rangle, \langle B', X'\rangle \in \Gamma$ and an action a such that
        $\langle B, X\rangle$ is unstable to $\langle B', X'\rangle$ and a
        Replace $P = \langle B, X\rangle$ in Γ by $\langle B_1, X_1\rangle =$
        $\langle \{s \mid s \in B \wedge s \xrightarrow{a} B'\}, \{s \mid s \in X \wedge s \xrightarrow{a} X'\}\rangle$
        and $\langle B_2, X_2\rangle = \langle B - B_1, X_1\rangle$
        If $s_0 \in B_2$ and $s_1 \in B_1$ **then**
            return **false**
    return **true**

Figure 6: A naive refinement-based algorithm for computing simulation

execution of $sim(s_0, s_1, T)$. The construction maintains the following invariant during the execution of procedure sim.

B. *For each block P in Γ, there is a leaf d in γ such that $\ell(d) = P$*

Initially, γ has only one node λ with label $\ell(\epsilon) = \langle S, S\rangle$. Assume that in the current iteration $P = \langle B, X\rangle \in \Gamma$ is split by $P' = \langle B', X'\rangle \in \Gamma$ on a, and is replaced by $\langle B_1, X_1\rangle$ and $\langle B_2 = B - B_1, X\rangle$. Let d and d' be the leaves related to P and P' by invariant B. We create two children for d such that $\ell(d.0) = \langle B_1, X_1\rangle$ and $\ell(d.1) = \langle B_2, X_2\rangle$. The splitter and splitting action for d are marked as $\ell_s(d) = d'$ and $\ell_a(d) = a$.

Clearly the resulting KA-PRT γ satisfies (1) and (2) of Definition 8.3.3. To see (3), we may order the nodes by its splitting time. Clearly this will yield a desired well-founded order.

To analyze the complexities, we need some assumptions on the data structures used to encode KA-PRT. First, only leaves need to record their kernel sets. The kernel set of an internal node d can be recovered by the union of kernel sets of all the leaves in the subtree rooted at d. Clearly the space needed for storing kernel sets is bounded by $O(|S|)$. For auxiliary sets, instead of storing them directly, we store the *differential set* for each internal node d which is defined as below,

Definition 9.2.1 *Let d be an internal node of a* KA-PRT $\gamma = \langle D, \ell, \ell_s, \ell_a \rangle$ *with* $\langle B, X \rangle = \ell(d)$ *and* $\langle B_1, X_1 \rangle = \ell(d.0)$. *The differential set for d is defined as* $\ell^-(d) = \ell(d) - \ell(d.0)$.

The auxiliary set of a node d with $\langle B, X \rangle = \ell(d)$ can be computed by subtracting the differential sets of d's ancestors from S,

$$X = S - \bigcup \{\ell^-(d') \mid d' \text{ is an ancestor of d}\}$$

Lemma 9.2.2 *Given a transition system* $T = \langle S, A, \rightarrow \rangle$ *and a* KA-PRT $\gamma = \langle D, \ell, \ell_s, \ell_a \rangle$ *for T, for any pair of S' and s such that $S' \subseteq S$ is a bisimulation class of T and $s \in S$ is a state, then there exists at most one node $d \in D$ with* $\langle K^0, A^0 \rangle = \ell(d.0)$ *such that $S' \in K^0$ and $s \in \ell^-(d)$.*

Proof: We use contradiction to show this. Suppose not. Let d_0 and d_1 be two nodes such that $S' \in K^0$ and $s \in \ell^-(d_i)$, where $\ell(d_i.0) = \langle K^0, A^0 \rangle$. Since d_0 and d_1 share the same set of states in their kernel sets, they must be on the same branch of γ. W.l.o.g., Let d_0 be the one closer to the root. Since $s \in \ell^-(d)$, the nodes in the left subtree of d_0 do not contain s in their auxiliary sets, and hence d_1 must be a node in the right subtree of d. However, S' could not be in the kernel sets of any nodes in the right subtree of d because $S' \subseteq K$. Nevertheless the assumption on d_1 implies that S' and s should be in d_1's kernel set and differential set respectively, Therefore d_1 cannot exist. ■

By Lemma 9.2.2, the maximum number of occurrences of s in differential sets are limited to $|\mathbb{S}|$, where $|\mathbb{S}|$ is the number of bisimulation classes of T. Therefore, the total space of maintaining a KA-PRT is bounded by $O(|\mathbb{S}| \cdot |S|)$, which does not exceed the quadratic space required by standard simulation algorithms.

The naive algorithm can be adapted to use KA-PRTs as its internal data structures for storing partitions. The only extra work for maintaining the tree is to update ℓ_s and ℓ_a for the node being split, which can be done in $O(1)$. Therefore, adapting the naive algorithm to construct a KA-PRT won't compromise its time and space complexities. We found that the construction of KA-PRT does not compromise the time and space of more sophisticate refinement-based simulation algorithm like [48].

If $s_1 \not\preceq s_2$, then the KA-PRT generated above may not necessarily be stable. However, if $s_1 \sim s_2$, then s_1 and s_2 should be in the same leaf, and the resulting KA-PRT must be stable. These refinement trees, for example, can be constructed by above procedure. Now we can give the proof of Theorem 8.3.8 by showing how to construct such KA-PRT.

Proof of Theorem 8.3.8

Proof: The proof of (2) and (4) are similar to the proof of Theorem 8.1.4. We will prove (1) and (3),

1 ($\Rightarrow$) Let γ be the KA-PRT produced by the adapted naive algorithm in Figure 9.2. By the correctness of this algorithm (see, [48]) , the final partition represented by the leaves should be the relational coarsest KA-partition. The tree must be stable because every leaf is stable. Since $s_0 \preceq s_1$, there must be a leaf d with $\langle D, X\rangle = \ell(d)$ such that $s_0 \in D$ and $s_1 \in X$.

($\Leftarrow$) Follows from Theorem 8.3.7.

3 ($\Rightarrow$) Let γ be KA-PRT produced by the adapted naive algorithm in Figure 9.2. Since $s_0 \not\preceq s_1$, by the correctness of naive algorithm, there must be a leaf d such that $s_0 \in D$ and $s_1 \notin X$.

($\Leftarrow$) KA-PRT γ may be extended to a stable one by doing following, where initially $<_\omega$ is the well-founded order on splitters in γ.

If d is a leaf unstable with respect to node d' on action a, then, $d.0$ and $d.1$ are added as children of d so that

$$\begin{aligned}\ell(d.0) &= \langle B_1, X_1 \rangle \\ &= \langle \{s \mid s \in B \wedge s \xrightarrow{a} B'\}, \{s \mid s \in X \wedge s \xrightarrow{a} X'\} \rangle\end{aligned}$$

and,

$$\begin{aligned}\ell(d.1) &= \langle B - B_1, X \rangle \\ &= \ell(d) - \ell(d.1)\end{aligned}$$

Update $\ell_s(d) = d'$ and $\ell_a(d) = a$, and append d' to $<_\omega$.

The resulting tree γ' is stable, and $<_\omega$ as a desired well-founded order on splitters. Since s_0 and s_1 were already in different nodes in γ, they will end up in different leaves of γ'. By Theorem 8.3.7, it follows that $s_0 \not\precsim s_1$.

■

Chapter 10

Certifying Equivalence-Checking Results

10.1 Verifying PRT

By Theorem 8.1.4, the correctness of an equivalence-checking result can be verified by checking the validity of PRT submitted by checkers. Lemma 10.1.1 shows that the verification can be done in $O(|S| \cdot |\rightarrow|)$.

Lemma 10.1.1 *The validity of a partition refinement tree γ can be verified in $O(|S| \cdot |\rightarrow|)$.*

Proof: First, we assume that a data structure used to encode *sets* supports *insertion*, *deletion*, and *membership checking* in $O(1)$. A data structure to support this could be, for example, a two-level hash table.

To verify the validity of γ, a verifier needs to check all three properties given in Definition 8.1.1. Checking (1) is straightforward and can be done in $O(|S|)$. To check (2), we need to examine whether every internal node splits properly. That is, let s be a state in an internal node d and $d' = \ell_s(d)$ be the splitter of d, we need to check whether s is in d's left child $d.0$ if

$\exists s'.(s' \in \ell(d') \wedge s \rightarrow s')$ and s is in its right child $d.1$ otherwise. To decide to which child s should go, in the worst case we need check all s's successors to see whether one of those successors falls in $\ell(d')$. Clearly the cost for this is $O(|s \rightarrow \bullet|)$. Moreover, since there are at most $|S|$ internal nodes in a partition refinement tree, the accumulative cost of checking (3) for a given state s is bounded by $O(|S| \cdot |s \rightarrow \bullet|)$. Therefore, the total cost of checking (3) is bounded by $\sum_{s \in S} O(|S| \cdot |s \rightarrow \bullet|) = O(|S| \cdot | \rightarrow |)$.

To check (3), we construct a graph $\beta = \langle Q, E \rangle$ from γ in which $\langle q, q' \rangle \in E$ if $\langle q, q' \rangle$ is an edge in γ, or, $q \neq q'$ and q is q''s splitter. Clearly (3) holds if and only if there does not exist a non-trivial loop [1] in β. Moreover, if such loop does not exist, then any topological order on β will yield a desired well-founded order. Therefore, verifying (3) on γ can be done in $O(|\gamma|)$. Since we have shown that $|\gamma|$ is bounded by $O(|S|)$, the overall cost of verifying (1), (2), and (3) is bounded by $O(|S| \cdot | \rightarrow |)$. ■

10.2 Verifying PT-PRT

The verification procedure proposed by the proof of Lemma 10.1.1 simply checks the three properties on γ in a straightforward manner. There is one problem though. Since the fastest partition refinement tree [40] can run in $O(|T| \cdot log(|S|))$, the verification should also run within this time limit to preserve the complexity of overall system, per requirement (4) in Chapter 1. We will not pursuit an efficient algorithm for any arbitrary PRT. Instead, we develop an efficient checker for the type of trees which can be generated by various implementations of [40]. Since the trees generated by Paige-Tarjan are PT-PRTs, we are seeking for the efficient algorithm to check the validity of PT-PRTs.

[1] A loop is non-trivial iff it contains more than one nodes

A PT-PRT must satisfy two additional properties in Definition 8.2.1. These additional constraints enable us to use *counters* similar to those in [40] to expedite the verification process. Since siblings always act consecutively as splitters, we refer to the parent of siblings as a *compound splitter*. By saying that a node split with respect to a *compound splitter*, we mean that the node is split by its two children consecutively. During the computation, we associate every compound splitter p with a counter table $C(p)$ such that $C(p)[s]$ records the number of transitions from s to p. The algorithm has two phrases.

1. In the initialization phase, Properties (1) and (2) in Definition 8.1.1, (4) and (5) in Definition 8.2.1 are checked on the root node straightforwardly, and the counter table for the compound splitter λ is initialized.

2. In the second phase, the properties are checked for compound splitters. The verification starts from the root. Assume that d is the node we want to check, and p is the compound splitter for d, i.e., $\ell_s(d) = p.i$ and $\ell_s(d.0) = p.(1-i)$ for some $i \in \{0, 1\}$. q and $q.0$ will share the same splitting action, per Property (4) of Definition 8.2.1. Let $a = \ell_a(q) = \ell_a(q.0)$, and w.l.o.g., we assume $p.i$ is the smaller of p's children. To determine which child of p a state $s \in \ell(q)$ should go, we compute $C(p.i)[s] = |\{s' \mid s \xrightarrow{a} s' \wedge s' \in \ell(p.i)\}|$. If $C(p.i)[s] = C(p)[s]$, then, s shall be in $d.0.1$ (i.e., s has transition to $p.i$ but not $p.(1-i)$). If $0 < C(p.i)[s] < C(p)[s]$, then, s should be in $q.0.0$ (i.e., s has the transition to both $p.i$ and $p.(1-i)$). Otherwise, $C(p.i)[s] = 0$ and s shall be put into $q.1$ (i.e., s has transition to $p.(1-i)$ but not $p.i$).

The first phase can be finished in $O(|T|)$. Note that in the second phase we only count those transitions to the smaller child of a compound splitter. Since the smaller child contains fewer than half states in its parent, clearly the number of such "smaller" children is bounded by $O(log|S|)$. Next, we calculate the accumulative cost of counting transitions to a given "smaller" child. Fix the compound splitter p, and assume that $p.i$ is p's smaller child. Note that each transition $s \xrightarrow{a} s'$ such that s' is in $p.i$ will be counted only once, because, once the block containing s splits by the compound splitter p, the resulting nodes will be stable with respect to both children $p.0$ and $p.1$. Hence, the accumulative cost of counting the transitions to $p.i$ is bounded by $|T|$. Therefore, the cost of second phase is bounded by $O(|T|log(|S|)$.

Remarks we have provided an algorithm specially tailored for verifying the trees generated by implementations of Paige-Tarjan algorithm. Generally, to ensure efficiency users may want to use those verifiers specially tailored for certain category of PRTs. Using terms in object-orient techniques, we can model PRTs as objects which provide a method *verify* to verify their own consistency. The term *class* will refer to a certain category of PRTs generated by those checkers of interest. For example, we may define a class PT_PRT for PT-PRT. Method *verify* will be provided per class. Hence designing and implementing method *verify* can be done independently of checkers which produce objects. We only need to certify method *verify* in classes, because the consistency of an object can be then verified using method *verify* of the class this object is instantiated from. One may wonder why we just simply certify checkers themselves. Here are several reasons: we only need to certify method *verify* once for each class, while there are many implementations and variants of checkers which can fit to one class. Second, certifying method *verify* is often based on the abstract definition for certain categories of PRTs, so we don't have to know the details of individual checker. Therefore, it reduces possible human errors which might occur if we otherwise choose to certify checkers directly.

Chapter 11

Generating Diagnostic Information

In this chapter, we will show how to build diagnostic information using PRTs. Many traditional diagnostic schema may be built on the basis of PRTs, including Hennessy-Miler formulae in bisimulation checking [22], traces in language equivalence checking, and failure in failure equivalence checking [23]. For example, Cleaveland *et al* [37] have shown in the CWB-NC how to implement the above schema on a decorated tree structure which retains information of both splitters and splitting actions. Since it is straightforward to adapt the technique in [37] to using PRTs, we will not discuss the generation of static diagnostic information here. Interested readers can refer to [37].

Instead, we choose to show how an interactive and dynamic diagnostic schema, equivalence games, can be built on PRTs. We are motivated by two considerations: first, equivalence games are very powerful in terms of helping users understand why or why not two systems are equivalent, since they provide flexibility to users to choose which part of the checker's reasoning he/she wants to explore. This presents a challenge for PRT which, as a static form of evidence, must bear enough information to encode the checker's entire proof, since it cannot be know in advance which part of the proof the users may choose to explore. Second, there are many different equivalence semantics, each of which demands an unique game semantics. By show how PRTs support different equivalence-checking games, we show the versatility of PRTs: The evidence it encode may be used for a variety of equivalence checkings. Particularly we define equivalence-checking games on a three-level semantics hierarchy, which exploits the similarity among games. The implementation following this hierarchy enjoys greater code sharing and hence reduce the workload of implementing different games.

11.1 A Game Semantics Hierarchy

To define and implement an equivalence game efficiently and consistently, we introduce a three-level hierarchy by abstract levels of game semantics. At the top is an abstract game, which subsumes all property-checking games. In the middle is a generalized equivalence game, which captures the common features of equivalence games. On the bottom are individual games as instances of the generalized equivalence game.

11.1.1 The Abstract Game

This section serves two purposes: first, it introduces general concept of property-checking game in a quite abstract level. Second, by recognizing the common functions shared by all the games, we define the top of three-level hierarchy.

In the remaind of this Chapter, term "game" refers to property-checking game. The purpose of a game is to determine whether or not a transition system satisfies a certain property. There are two players in a game: **I**, who believes that the property doesn't hold for the system, and **II**, who assumes the opposite. We use $A, B \cdots$ to range over $\{\mathbf{I}, \mathbf{II}\}$.

The definition of a game can be given as a tuple $\langle C, c_0, \delta, \lambda, w\rangle$, which shall specify the following elements.

The set of all possible configurations C and initial configuration $c_0 \in C$.

Rules $\delta \subseteq C \to 2^C$ defines which move is legal. A *play* $p = c_0 \cdots \in C^*$ should be a sequence of configurations starting with c_0 which respects δ. We write $p^{(i)}$ for $c_0 \cdots c^i$. $\lambda : C \to \{\mathbf{I}, \mathbf{II}\}$ is a partial function which defines who move next from a given configuration. If $\lambda(c)$ is not defined, then the next player is "don't care". This happens, for example, in a model-checking game when a formula needs to be unrolled.

Winning criteria winning criteria determines who wins the game. It can be modeled as a partial function $w : 2^{(C^*)} \to \{\mathbf{I}, \mathbf{II}\}$. We require that the *winning criteria* is *complete*, that is, any infinite play shall eventually have a winner. In what follows, we shall consider only *non-trivial* plays, that is, the play is valid and terminates whenever someone wins the game.

The behavior of players are described by their *strategies*. Intuitively a strategy of player is a decision function by which a player chooses his next move. Formally a strategy of player A can be modeled as a partial function $\sigma_A : C^\omega \to 2^C$ such that $\sigma_A(pc_i) \subseteq \delta(c_i)$ whenever $\sigma_A(c_i)$ is defined and $\lambda(c_i) = A$. A strategy is *deterministic* if σ_A always returns a singleton. A play $p = c_0c_1 \cdots$ is under a strategy σ_A if $c_{i+1} \in \sigma_A(p^{(i)})$ whenever $\lambda(c_i) = A$ and $\sigma_A(p^{(i)})$ is defined. A strategy is *complete* if, for any play p under σ_A, $\sigma_A(p^{(i)})$ is defined whenever $\lambda(c_i) = A$. It is *history-free* if $\sigma_A(pc_{i+1}) = \sigma_A(qc_{i+1})$ for any prefix p and q. Strategy σ_A is a *winning* strategy iff any non-trivial play under σ_A will be won by A. It follows that at most one player has a winning strategy for any game.

When a game is used as a diagnostic routine, it has two physical players: computer and user. **I** and **II** will be referred as logical players, which can be seen as the roles the computer and the user choose to play. The computer want to convince the user of the correctness of the verification result. Therefore, it always chooses to act as the player in favor of the result: if the verification result is positive, the computer will choose to act as **II**, otherwise it will act as **I**. The user will take the opposite role. We require that the definition of game is *fair*, i.e., if the property does hold for the system, then, **II** has a winning strategy, otherwise, **I** has a winning strategy. Therefore, the computer always has a winning strategy in a fair game, if the verification result is correct.

There may exist more than one "fair game" definition for a verification semantics. Besides the logic design to ensure the fairness, a good definition of game should also be easily understandable to users and helpful for diagnosing design system.

11.1.2 A Generalized Equivalence-Checking Game

At the second level of the semantics hierarchy we consider a generalized equivalence game. It is in the middle because, to the lower level it defines the common aspects of individual games, and to the higher level it can be seen as an instance of abstract game.

In an equivalence-checking game, the property being checked is whether two processes t_0 and t_1 are related by a given equivalence relation $=$. In what follows, we will fix the transition system $T = \langle S, A, \rightarrow \rangle$, and consider t_0 and t_1 as the executions of T starting from two different states s_0 and s_1. Hence an equivalence-checking game is to determine whether these two states are equivalent.

The generalized equivalence-checking game defines the common aspects shared by all the equivalence-checking games. In this generalized version of equivalence-checking game, configurations have the form either $F(q_0, q_1)$ or $S(q_0, q_1, i, b, q')$, where $q_0, q_1, q' \subseteq S$ are the set of states, $i \in \langle 0, 1\rangle$ stands for the process t_i, and $b \in B$ is a symbolic action. We refer to B as the set of symbolic actions, the process t_i as the *side* i, and q_i as the configuration of the side i which represents the status of process t_i. Generally q_i is the set of states because it needs to record all the possible non-deterministic steps made by the side i. For example, considering the case of weak bisimulation, we need to know all the possible states resulting from executing a trace in order to judge whether two processes are equivalent.

A valid play P is an alternating sequence of F-configuration and S-configuration. We call a pair of alternating F and S-configuration a *round* of game. The initial configuration is a F-configuration $F(q_0, q_1)$ such that $s_0 \in q_0$ and $s_1 \in q_1$.

The next player $\lambda(c_i)$ is defined as below: if c_i is a F-configuration, then $\lambda(c_i) = \mathbf{I}$, otherwise $\lambda(c_i) = \mathbf{II}$. The definition δ relies on the notion of two transition relations, which is unique for each individual game: the natural transition $T_f : 2^S \rightarrow B \times 2^S$; and the matching transition $T_s : 2^S \times B \rightarrow 2^{(2^S)}$, where $Q = 2^S$. Given T_f and T_s, the next step of the current configuration pc_i is defined as below.

1. if $c_i \equiv F(q_0^k, q_1^k)$ is a F-configuration,then $\delta(c_i) = \{S(q_0^k, q_1^k, i, b, q')\}$, where $i \in \{0, 1\}$, and $\langle b, q'\rangle \in T_f(q_i^k)$ for some symbolic action $b \in B$. This can be understood as below. Player **I** picks side i and chooses a natural transition as the next state of side i.

2. If c_i is a S-configuration $S(q_0^k, q_1^k, i, b, q')$, then $\delta(c_i) = \{F(q_0^{k+1}, q_1^{k+1})\}$ such that $q_i^{k+1} = q'$ and $q_{1-i}^{k+1} \in T_s(q_{1-i}^k, b)$, This can be understood as below. After **I** chooses a side and a natural transition for that side, **II** matches **I**'s choice by choosing a matching transition for the other side.

The reason why we need two separate transition functions is that for some equivalence-checking games the definition of transition in the first step of a round is different from that in the second step. For example, in the weak bisimulation game the former refers to original transitions, while the latter refers to weak transitions.

Winning criteria is defined on two levels of this semantics hierarchy. The part of winning criteria common to all the equivalence games is defined in the generalized equivalence-checking game, and the other part is defined in each individual game. The latter is often defined as an equivalence relation Π. $w(c_i)$ is defined only if c_i is a F-configuration. i.e., we check the winning criteria only at the beginning of each round. Let $P = F(q_0^0, q_1^0)S(\cdots)\cdots F(q_0^k, q_1^k)$ be a play, **I** wins the game if,

1. There exists an action b and a side i such that $\exists q'.(\langle b, q'\rangle \in tr_f(q_i^k))$ and $tr_s(q_{1-i}^k, b) = \emptyset$, i.e., side i may fire an action b for which the other side cannot match; Or,

2. $\langle q_0^k, q_1^k\rangle \notin \Pi$.

II wins if,

1. **I** cannot make any move because $tr_f(q_0^k) = tr_f(q_1^k) = \emptyset$, i.e., none of sides may fire an action; or,

2. The play is infinite. Since we are considering only the finite transition systems, this is equivalent to check whether a F-configuration $F(q_0^n, q_1^n)$ has been repeated.

11.1.3 Semantics Plug-ins

The lowest level of the semantics hierarchy defines the part unique to each individual game. It usually includes the definition of initial configuration, the set of symbolic actions, an equivalence related Π used to assemble winning criteria, the natural transition relation T_f, and the matching transition relation T_s. We will consider bisimulation game, language equivalence game, and failure equivalence game.

11.1.3.1 The Bisimulation Game

The equivalence relation in question is the strong bisimulation relation $\sim$. In the bisimulation game $G(s_0, s_1)$, each side contains only a single state. The initial configuration is $F(\{s_0\}, \{s_1\})$. The set of symbolic actions is just A, the set of actions of T. Both nature and matching transitions refer to the transition relation of T. Formally, we have $T_f(\{s\}) = \{\langle b, \{s'\}\rangle \mid s \xrightarrow{b} s'\}$ and $T_s(b, \{s\}) = \{\{s'\} \mid s \xrightarrow{b} s'\}$. There is no special semantics-oriented winning criteria for this game. Therefore, Π is just the universal relation. **I** wins a play $P = F(\{s_0\}, \{s_1\}) \cdots F(\{s_0^k\}, \{s_1^k\})$ if $\exists a.\ (s_i^k \xrightarrow{a}) \wedge (s_{1-i}^k \not\xrightarrow{a})$. **II** wins the game if $F(\{s_0^k\}, \{s_1^k\})$ has occurred before.

The complete definition of strong-bisimulation game can be assembled from its three-level definition. Nevertheless, it can be interpreted in plain English easily as below. The game involves two players and two processes (i.e., sides). A play is a sequence of rounds. In each round **I** always goes first. He picks up a process and makes an action on this process. **II** will choose the other process to show that she too can fire the same action. If no initial action is shared by two processes, then **I** wins. If a configuration has been repeated, then, **II** wins. To access the fairness of this game, one may refer to [44].

11.1.3.2 The Language Equivalence Game

In language equivalence game $GL(s_0, s_1)$ the property in question is whether s_0 and s_1 have the same set of traces. The initial configuration $F(\{s_0\}^\epsilon, \{s_1\}^\epsilon)$ starts with the ϵ- closure of s_0 and s_1. The natural and matching transitions refer to an extended definition of weak transitions in term of sets of states.

That is, $t_f(q) = \{\langle a, q' \rangle \mid q \stackrel{a}{\Rightarrow} q' \wedge q' \neq \emptyset\}$ and $t_s(a, q) = \{q' \mid q \stackrel{a}{\Rightarrow} q' \wedge q' \neq \emptyset\}$. There is no need for semantics-oriented winning criteria in this game. i.e., Π is just the universal transition.

By their definitions both t_f and t_s are deterministic. The proceeding of a play is controlled solely by **I**. **II** just passively matches the action chosen by **I** because the deterministic matching transition leaves no choice to **II**. Theorem 11.1.1 shows the fairness of language equivalence game defined above.

Theorem 11.1.1 *Given a LTS $T = \langle S, A, \rightarrow \rangle$ with s_0 and s_1 being two distinguished states of T,*

1. *If $L(s_0) = L(s_1)$, then* **II** *has a winning strategy in language equivalence game $GL(s_0, s_1)$.*

2. *If $L(s_0) \neq L(s_1)$, then* **I** *has a winning strategy in language equivalence game $GL(s_0, s_1)$.*

Proof:

1. We will use contradiction to show that **II** will win each and every play in game $GL(s_0, s_1)$. Suppose not, since any play must be won by someone (**II** wins infinite plays), we can find a play $P = F(\{s_0\}^\epsilon, \{s_1\}^\epsilon)S(\cdots)\cdots F(q_0^k, q_1^k)$ which is won by **I**. By winning criteria this means q_i^k can make an action a while q_{1-i}^k cannot. Let β be the trace followed by P so far, then the trace βa is accepted only by side i but not by side $1 - i$, which contradicts the fact $L(s_0) = L(s_1)$. Therefore, **II** wins each and every play in $GL(s_0, s_1)$ if $L(s_0) = L(s_1)$.

2. Let β be a trace such that $\beta \in L(s_0)$ but $\beta \notin L(s_1)$. then, **I** just chooses to follow β whenever it need make choice. Eventually there is a prefix β' of β after which **II** cannot match **I**'s choice. Clearly this will yield a winning strategy for **I**.

■

From the proof of Theorem 11.1.1, we can see that the winning strategy for **II** in the case $L(s_0) = L(s_1)$ is quite simple: it just passively follows whatever action is chosen by **I**. In the case that $L(s_0) \neq L(s_1)$ with $\beta \in L(s_i) - L(s_{1-i}$, the computer as **I** simply chooses side i and follows β.

11.1.3.3 Failure Equivalence Game

The failure equivalence game $GF(s_0, s_1)$ intends to show whether two processes represented by two states s_0 and s_1 share the same set of failures. The initial configuration is $F(\{s_0\}^\epsilon, \{s_1\}^\epsilon)$ starts with the ϵ- closure of s_0 and s_1.

The natural transition T_f and matching transition T_s are defined as below,

$$T_f(q) = \begin{cases} \{\langle a, q' \rangle \mid q \stackrel{a}{\Rightarrow} q' \wedge q' \neq \emptyset\} & \text{If } q \downarrow \\ \emptyset & \text{Otherwise} \end{cases}$$

and,

$$T_s(a, q') = \begin{cases} \{q' \mid q \stackrel{a}{\Rightarrow} q' \wedge q' \neq \emptyset\} & \text{If } q \downarrow \\ \emptyset & \text{Otherwise} \end{cases}$$

The definition of T_s and T_r are very similar to those for language equivalence game in the case of convergence. If one side diverges, then it has no outgoing transition. Again, since both T_s and T_r are deterministic, only **I** can control the proceeding of a play by choosing the next visible action. **II** can only passively follow **I**'s choice. A play ending with an F-configuration can be seen as the result of executing a visible sequence from the initial configuration.

The semantics-oriented winning criteria Π is defined as below. $\langle q, q' \rangle \in \Pi$ iff one of the following stands,

1. Both q and q' diverge.

2. Both converge and $min\ Acc(q) = min\ Acc(q')$.

In other word, if the convergence status are different for two sides, or both converges but with different minimal acceptance set, then, two sides are considered as not equivalent. As the result, **I** will win the play.

Lemma 11.1.2 *Let* $P = F(q_0^0 = \{s_0\}^\epsilon, q_1^0 = \{s_1\}^\epsilon)S(\cdots)\cdots F(q_0^k, q_1^k)$ *be a play of the game* $GF(s_0, s_1)$ *resulting from executing the visible sequence* β. *Then,*

(i). β *is a trace of both* s_0 *and* s_1.

(ii). If $q_i^k \downarrow$, *then* $s_i \downarrow \beta$.

Proof: (i) follows from the definition of t_f and t_s, which define the visible behavior of both sides. We may use contradiction to prove (ii). Suppose s_i diverges on β. Let $\beta_0 = a_0 \cdots a_j$ be the shortest prefix of β such that $s_i \uparrow \beta_0$, i.e., q_{1-i}^{j+1} diverges. It follows that there is no outgoing transition on t_f from q_{1-i}^{j+1}, which contradicts the fact that β is a trace on t_f proceeding beyond q_{1-i}^{j+1}. ■

Lemma 11.1.3 *If* **I** *wins a play in the game* $GF(s_0, s_1)$, *then* $s_0 \not=_{must} s_1$.

Proof: Let the play won by **I** be $P = F(q_0^0 = \{s_0\}^\epsilon, q_1^0 = \{s_1\}^\epsilon)S(\cdots)\cdots F(q_0^k, q_1^k)$ with β as the trace being followed by both sides so far. We will show that there is a failure not shared by both sides. By the winning criteria, **I** wins the play iff one of following holds for $F(q_0^k, q_1^k)$.

(i). q_i^k converges but q_{1-i}^k diverges. Clearly in this case $\langle \beta, \emptyset \rangle$ is a failure of s_{1-i} but not s_i.

(ii). Both sides converges but have different minimal acceptance sets. Then there is a i and an element $B \in$ *min* $Acc(q_i^k)$ such that $C \in$

$min\ Acc(q^k_{1-i}) \Rightarrow C \nsubseteq B$. We may define the *differential set* for a set of states B and a set of sets of states D as below,

$$diff(B, D) = \bigcup_{C \in D} (C - B)$$

We denote $\mathcal{K} = diff(B, minAcc(q^k_{1-i}))$. We will show that $\langle \beta, \mathcal{K} \rangle$ is a failure for s_i but not s_{1-i}. By the definition of acceptance set, we can find a state $s \in q^k_i$ whose acceptance set is B, hence s refuses $\mathcal{K}$ and $\langle \beta, \mathcal{K} \rangle$ is a failure of s_i. Now we only need to show B^- is not a failure of s_{1-i}, which follows from the fact that any state in q^k_{1-i} cannot refuse $\mathcal{K}$, and $s_i \downarrow \beta$ as stated by Lemma 11.1.2. Therefore, $\langle \beta, \mathcal{K} \rangle$ is a failure distinguishing s_0 and s_1, i.e., $s_0 \neq_{must} s_1$.

(iii). Both sides converge but q^k_i can make a visible transition a which q^k_{1-i} cannot match. Hence, βa is a trace of q^k_i but not q^k_{1-i}. By the definition of *failure* $\langle \beta a, \emptyset \rangle$ is a failure for s_i but not for s_{1-i}, because βa is not a trace of s_{1-i} and $s_{1-i} \downarrow \beta a$.

■

Theorem 11.1.4 *Given a LTS $T = \langle S, A, \rightarrow \rangle$ with s_0 and s_1 being two distinguished states of T,*

1. *If $s_0 =_{must} s_1$, then* **II** *has a winning strategy in the failure equivalence game $GL(s_0, s_1)$.*

2. *If $s_0 \neq_{must} s_1$, then* **I** *has a winning strategy in the failure equivalence game $GL(s_0, s_1)$.*

Proof:

1. Since both T_s and T_f are deterministic and **I** always goes first in every round, **II** can only passively follows **I**'s choice. If $s_0 =_{must} s_1$, then this simple strategy is a winning strategy for **I** by Lemma 11.1.3.

2. We can find a shortest $\langle \beta, B \rangle$ such that it is a failure for s_i but not for s_{1-i}. W.l.o.g., we fix $i = 0$.

   We propose a strategy σ_I for **I** such that **I** sticks to the side 0 and dutifully follow β. Let $P = F(q_0^0 = \{s_0\}^\epsilon, q_1^0 = \{s_1\}^\epsilon)S(\cdots)\cdots S(\cdots)F(q_0^k, q_1^k)$ be a play under this strategy. We have to show **I** is the winner in both of following cases.

   (a) A player won the play before it reaches $F(q_0^k, q_1^k)$. We use the contraction to show that **II** cannot be the winner in this case. Suppose that **II** wins at $c_m = F(q_0^m, q_1^m)$. By the winning criteria, **II** can win iff one of following holds : (a) c_m has been repeated on P. Let c_l be the first occurrence of c_m, and rewrite β as $\beta_0\beta_1\beta_2$ such that β_0 and β_1 are the traces followed by P's segments $c_0 \cdots c_l$ and $c_l \cdots c_m$, respectively . Since configurations encode all the non-deterministic choices and c_l is repeated by c_m, if we trim out β_1 from β, $\langle \beta_0\beta_2, B \rangle$ is still a failure for s_0 but not for s_1, which contradicts the assumption that $\langle \beta, B \rangle$ is the shortest among such failures. (b) Both q_0^k and q_1^k converge and no one can make a visible action. Hence β is not a trace of s_0. Let β' be the trace followed so far, then $s_0 \downarrow \beta'$. Therefore, $\langle \beta, B \rangle$ cannot be the failure of s_0, which also contradicts our assumption.

   (b) The game reaches $F(q_0^k, q_1^k)$ without a winner. Hence, β is the trace for both side, and $q_1^k \downarrow$ because, otherwise, $\langle \beta, B \rangle$ is also a failure for s_1. There are two cases, depending on whether q_0^k converges,

      i. $q_0^k \uparrow$. Therefore, **I** wins because two sides have different convergence status.

      ii. $q_0^k \downarrow$. Then we can find a $D \in min\ Acc(q_0^k)$ such that $D \cap B = \emptyset$, and a stable state $s \in q_0^k$ such that D is the set of initial actions of s. But D cannot be an element of $min\ Acc(q_1^k)$ at the same time because, if so, there is a stable state $s' \in q_1^k$ such that D is also the set of initial actions of s', hence s' will refuse B and $\langle \beta, B \rangle$ is also a failure of s_1, which contradicts our assumption. Thus, the minimal acceptance sets of q_0^n and q_1^n cannot be same, and **I** wins the play.

■

From the proof of Theorem 11.1.4, the winning strategy for the computer can be constructed as below: if $s_0 =_{must} s_1$, then the computer will act as **II**. It just passively follows whatever action chosen by **I**. If $s_0 \neq_{must} s_1$, then the computer will act as **I**. It will choose to follow a shortest failure of one side.

11.2 Efficiently Constructing Winning Strategies

We have defined the games for bisimulation, language equivalence, and failure equivalence. To make these games usable as diagnostic routines, one needs to find an efficient way to construct winning strategies for the computer. Instead of building an algorithm exclusively for this purpose, we "recycle" the information already computed by equivalence checkers. This information is encoded as partition refinement trees (forests).

11.2.1 Winning Strategies in the Bisimulation Game

Assume that γ is a PRT for the problem $s_0 \sim s_1$. Recall that one side contains only one state in this game, we may speak of a side while actually refer to the state in this side. There are two cases, depending on whether the two states are related.

1. $s_0 \not\sim s_1$. Hence s_0 and s_1 are in different leaves of γ. We will form a winning strategy σ_I for the computer (as **I**) such that a play under σ_I holds the following invariant,

Two sides are always in different leaves of γ.

The initial configuration holds the invariant trivially. Now, let $F(q_0^k \equiv \{s_0^k\}, q_1^k \equiv \{s_1^k\})$ be the starting configuration of round k. By the invariant, s_0^k and s_1^k are in two different leaves, say d_0 and d_1. Let d be the closest common ancestor of d_0 and d_1. W.l.o.g., assume that s_i^k is in the left subtree rooted at d and s_{1-i}^k is in the right subtree. The computer will pick the next configuration $S(q_0^k, q_1^k, i, b, q' \equiv \{s_i^{k+1}\})$ such that $b = \ell_a(d)$ is the splitting action of d and s_i^{k+1} is a b-derivative of s_i^k in d, i.e., $s_i^{k+1} \in (\ell_s(d) \cap s_i^k \xrightarrow{b} \bullet)$. By Property (2) of Definition 8.1.1, such a choice is always available. Moreover, no matter which s_{1-i}^k the user chooses, it cannot be in the same leaf as s_i^{k+1}, per Property (2) of Definition 8.1.1.

Now we show that the user (as **II**) cannot win under σ_I. First, the play cannot reach a configuration in which both sides have no transitions, because the computer always has a valid move by the construction of σ_I. Second, the repetition of F-configuration under σ_I implies that there is no well-founded order on the nodes of γ; therefore such repetition cannot exist.

2. $s_0 \not\sim s_1$. Hence s_0 and s_1 are in the same leaf of γ and γ is stable. The winning strategy σ_{II} maintains the following invariant,

Two sides are in the same leaf of γ

The invariant holds trivially for the initial configuration. Let $F(s_0^k, s_1^k)$ be the starting configuration of round k, and the user (as **I**) chooses b-derivates s_i^{k+1} as the successor of s_i^k. Let b_i be the leaf which s_i^{k+1} is in. **II** chooses a b-derivative s_{1-i}^{k+1} as the successor of s_{1-i}^k such that s_{1-i}^{k+1} is in b_i. Following the argument similar to the above, one can show that this yields a winning strategy for **II**.

11.2.2 Winning Strategy in Other Equivalence Games

For other games, the winning strategies can be constructed by tracing PRTs on the account of reference transition systems. The idea is to establish an one-to-one mapping between sides in a play and states in the reference transition system. Whenever sides moves, the related states in the reference transition system also move accordingly. The position of these reference states in a PRT will help determine the next move for the computer: his next move should ensure that the reference states stay in the same leaf of the PRT if the equivalence-checking result is positive, and remain in different leaves otherwise. To elaborate on this idea, we need to define a one-to-one mapping h which relates sides with states in reference system, and extend the notion of side to a tuple $p_i^k \equiv \langle q_i^k, s_{ri}^k \rangle$, where q_i^k is the side in original form, and $s_{ri}^k = h(q_i^k)$ is a state in the reference system.

As we have identified that the technique of building winning strategies by tracing PRTs is common for all the equivalence games, it can be implemented as a part of equivalence game module (EQM). The strategy constructed by tracing PRTs is an abstract version of winning strategy, i.e., it is given on the account of reference systems without referring to targeted equivalence semantics or transition systems. It is part of semantics plugin's job to interpret states appearing this abstract winning strategy as sides in the targeting game. In other words, semantics plugins "translate" the abstract winning strategy to the winning strategy for the computer in destination games. The interpretation may be implemented in the form of a callback function $\langle q', b \rangle = T_c(p \equiv \langle q, s_r \rangle, s'_r, a)$, which indicates that the transition $s_r \xrightarrow{a} s'_r$ is interpreted as the move from q to q' via a symbolic action b.

The definition of h and T_c for the language equivalence game can be defined as below:

$$h(\langle q \rangle) = \langle q, \mathbf{0} \rangle$$

where $q \in 2^S$, and,

$$T_c(p \equiv \langle q, s_r \rangle, s'_r \equiv \langle q', \mathbf{0} \rangle, a) = \langle a, q' \rangle$$

where s_r, s'_r are states in $D(T)$, the acceptance graph of T for language equivalence.

The definition of h and T_c for the failure equivalence game is quite similar:

$$h(\langle q \rangle) = \langle q, q \uparrow \rangle$$

and,

$$T_c(p \equiv \langle q, s_r \rangle, s'_r \equiv \langle q', b \rangle, a) = \langle a, q' \rangle$$

where $b \in \{\mathbf{0}, \mathbf{1}\}$, and s_r, s'_r are the states of $ST(T)$, the acceptance graph of T fr language equivalence.

Let γ be a PRT interpreted on T_r. The details of forming a winning strategy is below, depending on whether two processes are related.

1. s_0 and s_1 are not related by the equivalence relation in question. Let $F(p_0^k \equiv \langle q_0^k, s_{r0}^k \rangle, p_1^k \equiv \langle q_1^k, s_{r1}^k \rangle)$ be the starting configuration of round k. s_{r0}^k and s_{r1}^k are in two different leaves of γ, say d_0 and d_1 respectively. The next move on the account of reference states is chosen exactly by the same approach used in bisimulation game: First, EQM will decide the common ancestor d for d_0 and d_1. Let a and d' be the splitting action and splitter for d, and assuming d_i is in the left subtree of d, then the computer chooses side i and the next reference state for side i is s_{ri}^{k+1} such that $s_{ri}^{k+1} = \ell(d')$. T_c will be called to determine q_i^k 's successor q_i^{k+1}. The next configuration will be $S(p_0^k, p_1^k, i, a, p' \equiv \langle q', s_{ri}^{k+1} \rangle)$ such that $\langle b, q' \rangle = T_c(p_i^k, a, s_{ri}^{k+1})$.

2. s_0 and s_1 are related by the equivalence relation in question. Let $S(p_0^k \equiv \langle q_0^k, s_{r0}^k \rangle, p_1^k \equiv \langle q_1^k, s_{r1}^k \rangle, i, a, p' \equiv= \langle q_i^{k+1}, s_{ri}^{k+1} \rangle)$ be the configuration after

I moves in round k. The computer chooses the successor $s^{k+1}_{r(1-i)}$ for $s^{k+1}_{r(1-i)}$ such that s^{k+1}_{ri} and $s^{k+1}_{r(1-i)}$ are in the same leaf of γ. The next configuration is $F(p^{k+1}_0 \equiv \langle q^{k+1}_0, s^{k+1}_{r0}\rangle, p^{k+1}_1 \equiv \langle q^{k+1}_1, s^{k+1}_{r1}\rangle)$, where q^{k+1}_{1-i} is the state related to $s^{k+1}_{r(1-i)}$ by T_c, i.e., $\langle a, q^{k+1}_{1-i}\rangle = T_c(p^{k+1}_i, a, s^{k+1}_{r(1-i)})$.

The correctness of the winning strategy can be proven similarly as in bisimulation game.

11.3 A Game Hierarchy for Preordering Checking*

Preordering checking games may also fit this hierarchy structure. Input systems are translated to reference systems, which is essentially same as acceptance graphs in Chapter 7; Then reference systems are fed into KA-PRT-friendly preordering checkers, which produce KA-PRTs as their results. The middle-level module, *Preordering Checking Module (PRM)*, will trace KA-PRT instead of PRT to form an abstract winning strategy as below. The basic mechanisms of tracing are quite similar: let $\gamma = \langle D, \ell, \ell_s, \ell_r\rangle$ be the KA-PRT submitted by checker, and s_{r0}, s_{r1} be the states in reference system related to the current configuration. PRM will keep the following invariant.

1. If two processes are related, then there is a leaf node $d \in D$ with $\langle B, X\rangle = d$ such that $s_{r0} \in B$ and $s_{r1} \in X$.

2. If two processes are not related, then there is a leaf node $d \in D$ with $\langle B, X\rangle = d$ such that $s_{r0} \in B$ but $s_{r1} \notin X$.

A play in a preordering games is still an alternating sequence of F-configuration and S-configuration; and **I** still moves first in each round. The difference is that **I** cannot freely choose sides. He will stick to side 0. Because of the similarity of EQM and PRM, it is possible to merge them as a single module.

Like in equivalence games, lower-level semantics plugins supply the semantics-oriented information including transition functions T_f, T_s, and T_c, and special winning criteria in the form of a preorder relation instead of an equivalence relation. The plugins are quite close to their counterpart in equivalence games, and can be weld together. For example, in our implementation on CWB-NC we use a single plugin to encode both the failure equivalence game and the failure preordering game.

Part III

System Work and Conclusion

Chapter 12

A Prototype on the Concurrency Workbench of the New Century

To assess the feasibility of this framework, we have implemented a prototype on the Concurrency Workbench of the New Century [13].

12.1 The Concurrency Workbench of the New Century

The Concurrency Workbench of the New Century (CWB-NC) [13] previously called the Concurrency Workbench of North Carolina, is a toolkit for verifying finite-state systems. It supports several formulations of verification problems, including those in the domain of equivalence checking, preordering checking, and model checking. The system also provides the following features,

1. Diagnostic information. CWB-NC provides the diagnostic information

to explain why two systems fail to be related by an equivalence or preordering relation.

2. Language flexibility. CWB-NC supports a variety of design languages. The input system encoded in some design language will be translated to an automaton. By localizing translation procedures, CWB-NC makes it easy to add support for new design languages. It also provides a compiler [12] to help generate new translation procedure from a specification for the syntax and semantics of the language.

Figure 12.1 shows the architecture of CWB-NC. In the CWB-NC, the input system is translated to an automaton encoded in an uniform data structure. The structure of automaton mixes LTS and Kripke Structure concepts: it is a labeled transition system whose states are decorated with sets of atomic propositions. Hence it can be used for encoding transition system for the purpose of both model checking and equivalence/preordering checking. The kernel, or the verification routines, takes automata and perhaps a temporal formula as input and computes answers and diagnostic information.

Like other existing tools, the diagnostic information generation is integrated with verification routines, so the information being computed during the verification can be used for generating diagnostic information. In CWB-NC version 1.2, if two processes are not related by a desired equivalence/preordering relation, the CWB-NC will return a textual-based formula which is satisfied by one process but not the other. The definition of this formula varies depending on the relation being checked.

Equivalence/Preordering Checking In bisimulation equivalence and simulation preordering, the distinguishing formula is presented as a Hennessy-Milner Logic formula (HML) [22]. A HML formula is constructed by the following grammar,

$$\Phi := \mathbf{tt} | \mathbf{ff} | \Phi \wedge \Phi | \Phi \vee \Phi | \langle a \rangle \Phi | [a] \Phi$$

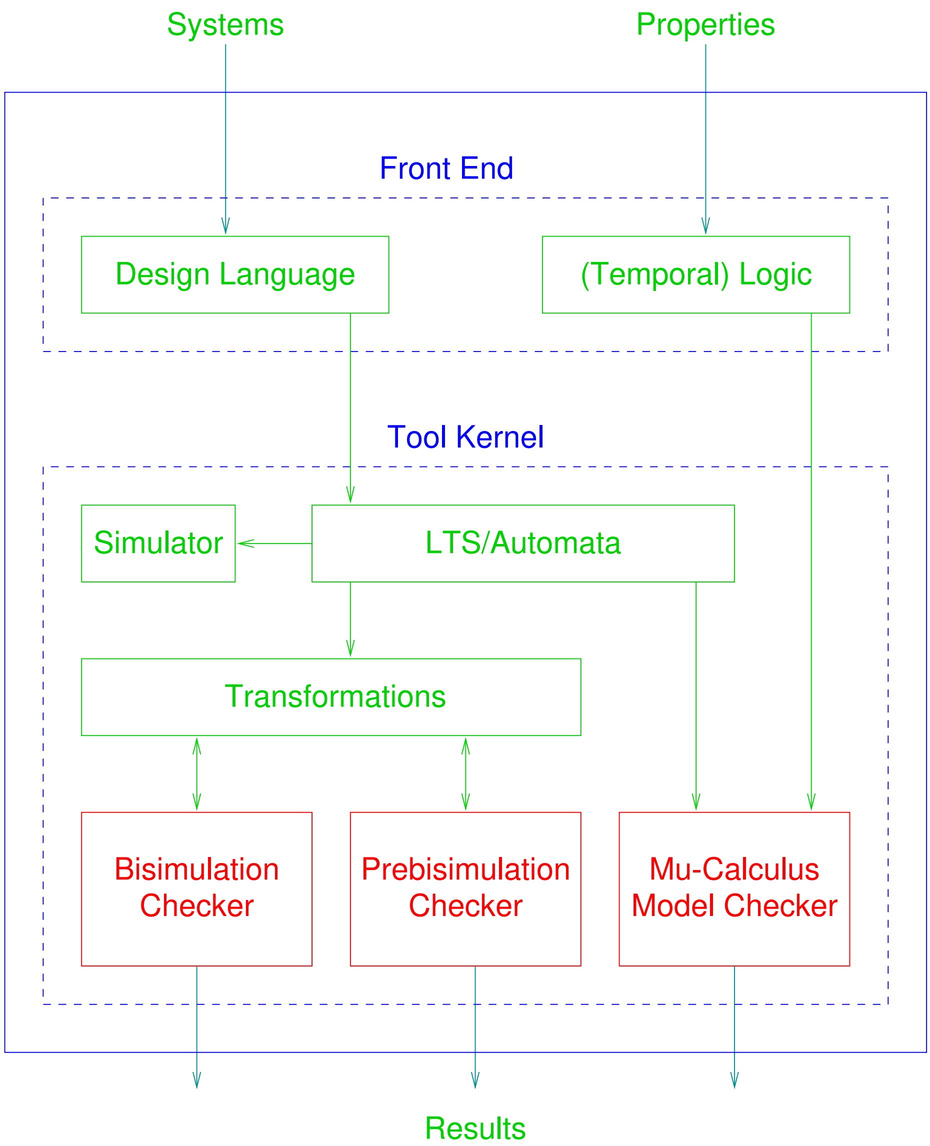


Figure 7: The architecture of CWB-NC [13]

It can be seen as a subset of μ-calculus formulas having no fixpoint operators. A HML formula is interpreted with respect to Kripke structure. A HML formula stands for the set of states where the formula holds. For example, $\langle a\rangle\Phi$ stands for the set of states which have some a-derivative at which Φ holds, and $[a]\Phi$ holds at a state if all its a-derivatives satisfy Φ. For bisimulation equivalence, the CWB-NC will return a process among two being checked and a HML formula Φ such that Φ holds exactly for this one but not the other. For the simulation preordering, HML produced by the CWB-NC is held by the lower process but not the higher one.

For observational bisimulation equivalence and observational simulation preordering, HML is slightly adapted to use weak transitions: $\langle a\rangle\Phi$ and $[a]\Phi$ are replaced by $\langle\langle a\rangle\rangle\Phi$ and $[[a]]\Phi$, respectively. The formula $\langle\langle a\rangle\rangle\Phi$ holds iff Φ holds at a state which has some weak a-derivatives where Φ holds, and similarly, $[[a]]\Phi$ holds at a state if all of its weak a-derivatives satisfies Φ.

For language equivalence and preordering, the diagnostic information return by CWB-NC is a visible sequence which is a trace of one process but not the other. For failure equivalence and preordering, the diagnostic information is a test that one process must pass while the other may fail.

In the CWB-NC, the equivalence/preorder checking is reduced to bisimulation/simulation checking on the corresponding reference system, per Chapter 7. Bisimulation checker used in CWB-NC is a modified Paige-Tarjan [40] adapted to compute Π-bisimulation: it starts with a partition induced by an equivalence relation Π, instead of the trivial partition. The simulation relation is computed by a variant of the Bloom-Paige algorithm [5].

Model Checking For reasons of efficiency, the CWB-NC has implemented more than one model checker, each of which is tailored for certain fragment of μ-calculus. The current release of CWB-NC includes a linear algorithm for the alternation-free fragment [14] and a L2 model checker [3]. The CWB-NC provides no diagnostic information for model checking.

The CWB-NC is implemented in SML of New Jersey and consists of approximately 18,000 lines of code. It supports Win32 and a variety of Unix platforms.

12.2 The Prototype

12.2.1 Objectives

By building a prototype, we have the chance to assess two important aspects of this framework.

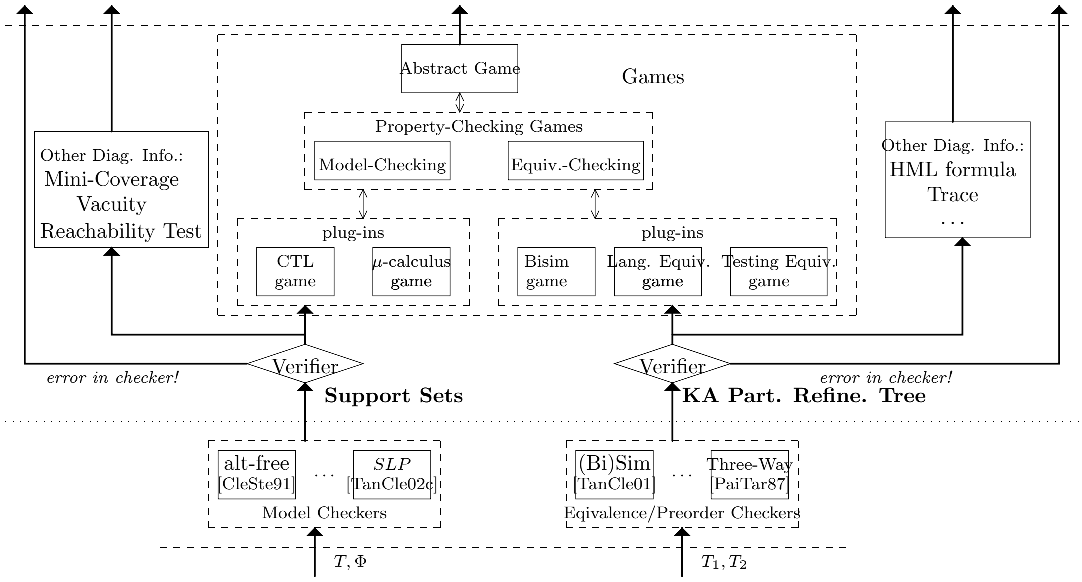
1. Feasibility of extracting *abstract proof structures*, especially the impact of extraction on the efficiency of checkers.

2. Usability of *abstract proof structures* in terms of generating diagnostic information, especially games.

The implementation is coupled with efforts of improving the efficiency of verification search engineering, and enhancing the diagnosis function with games.

12.3 Implementation

The implementation basically reshapes kernel module. Figure 8 shows the new kernel module. The diagnostic functions become checker-independent modules. Abstract proof structures are extracted from a variety of checkers and verified before being passed to diagnostic routines. The diagnostic information currently implemented in CWB-NC as well as property-checking games are constructed based on these abstract proof structures. Particularly, the design and implementation of games follow the semantics hierarchy introduced in Chapter 11. The hierarchy reduces the implementation cost, while adding the



A Prototype On the Concurrency Workbench

Figure 8: A Prototype On the Concurrency Workbench

flexibility and consistency to the implementation. To introduce a new game, one usually just adds the relatively small plugin to the lowest level of this hierarchy. Any changes made on higher levels will be applied to all games at lower levels. Since the user interface and overall control have been implemented as the part of abstract game module, the look and feel of various games are quite similar, which expedites the learning process for new users.

At a lower level, the introduction of two new model checkers extends the supported μ-calculus formula from the L2 fragment to the full μ-calculus. Individual simulation checker and bisimulation checkers are replaced by a single routine capable of computing both relations.

12.3.1 New APS-friendly Verification Algorithms

Model Checkers In addition to model checkers existing in version 1.2, we implement two local model checkers, LAFP [31] and SLP [50], for μ-calculus. We have special interest in local algorithms because of practical considerations: Local algorithms is driven by goals and constructs only the portion of transition system relevant to the problem. In practice they often reach the result faster than global ones, especially if the answer to the model-checking problem is negative. This makes them very useful for debugging, where the model checker may repeatedly be called to check revised designs. Nevertheless, local algorithms usually have the worse result on time complexity, compared with global ones. As other algorithm designed for the full μ-calculus, local checkers for the full μ-calculus may not perform well in some commonly-used fragments of μ-calculus. To overcome these difficulties, we designed and implemented SLP [50]. SLP is a linear-space local algorithm for the full μ-calculus which achieves the same complexity as the known best linear-space algorithm [32]. Table 2 shows some experimental results for SLP. All testing was done on a Sun Ultra SparcIIi with two 336 MHz processors and 3 GB of main memory. All times are reported in seconds of CPU time. $SCSIBus$ models the SCSI-II bus standard and is contained in the CWB-NC release. Our results show that SLP significantly outperforms the $LAFP$ and even has the competitive performance compared with a model checker specially designed for a fragment of μ-calculus. The L2 model checker refers to the one in [3].

As shown in 4, extracting support sets from these two checkers is straightforward: the support set is just another presentation of dependency sets already computed by LAFP. For SLP, support set is equivalence to direct dependent sets. The extraction requires no changes on the structure of both checkers.

Simulation/Bisimulation Checker The simulation and bisimulation checkers in this prototype has been replaced by a single algorithm which is capable of computing both relations [48]. [48] gives two algorithms to compute

Formula	# variables in μ-calculus	alternation Depth
Property1	38	1
Property2	14	2
Property3	15	1
Property4	11	1
Property5	14	2
Property6	15	2
NoDeadlock	6	1

Table 1: Properties in μ-calculus equation system

Agent Formula	*SLP*	*LAFP*	L2 model checker
SCSIBus Property1	513.82	N/A[1]	415.03
SCSIBus Property2	38.90	414.69	67.32
SCSIBus Property3	44.92	61.56	25.94
SCSIBus Property4	36.63	51.23	18.78
SCSIBus Property5	34.17	44.53	40.18
SCSIBus Property6	37.36	120.41	38.15
SCSIBus NoDeadlock	48.65	58.55	25.22

1. Memory allocation error after > 4 hour

Table 2: Test Result for *SLP*, *LAFP* and L2 model checker on SCSI-Bus

the simulation preordering as the relational coarsest KA-partition problem. The naive algorithm has been shown in Chapter 9. The one implemented in the CWB-NC uses the idea of three-way splitting introduced in [40] to expedite the process of splitting.

The algorithm has the same time complexity as Paige-Tarjan algorithm when used to compute bisimulation. In our experiment, this checker has the same speed as Paige-Tarjan algorithm in computing bisimulation relation. It preforms considerably better than the Bloom-Paige algorithm [5] when computing simulation. In Chapter 9 we discussed how to adapt the naive algorithm to use KA-PRTs as its internal representation of partitions. In our implementation, we use the exactly same idea to implement the three-way splitting algorithm for the KA-partition problem. Therefore, extracting KA-PRTs from this algorithm can be done at no cost.

12.3.2 The Games

We have implemented games for nine different verification approaches, including μ-calculus model checking, and the checking of different equivalence and preorderings. The design and implementation follows the three-level semantics hierarchy introduced in Chapter 11. We will give the implementation details by abstract levels of semantics hierarchy.

Abstract game The abstract game module is responsible for overall control, history maintenance, and user interface.

Overall controller takes the current configuration as input, then calls the *referee function* provided by lower level. The referee function encodes winning criteria and rules. It tells the controller whether someone already wins the game and if not, who is the next player. The controller then decides whether play should proceed, and if so, calls the strategy function of next player.

The strategy of a player is given by its strategy function. For the user, the strategy function simply prompts all the valid moves and let the user choose one as the next move. Hence, it is just part of user interface.

Configurations are handled abstractly, and the history is the sequence of abstract configurations while the detail of configurations are interpreted by the lower level. Therefore, the history maintenance can be implemented as part of abstract game.

Appendix B gives screenshots of model-checking games and bisimulation games. Readers shall note that the look and feel of these games are quite similar, despite the semantics of two games are completely different. This consistency results from implementing the user interface as part of abstract game module. These games also provide backtracking, which is especially useful for debugging purpose. Backtracking is part of history maintenance code, which in turn is part of the abstract game. This avoids the cost of implementing backtracking separately in each individual game.

Semantics game The middle of this hierarchy has two modules: the equivalence/preordering game module (EQM/PRM) and the model-checking game module (MCM). These modules implement the winning criteria in the form of *referee* function, and computer strategy function. These functions abstract the winning criteria and computer strategy of individual games in the domain of model checking and simulation/preordering checking. More precisely, the computer strategy function will construct an abstract version of its winning strategy. In EQM/PRM, this is done by tracing PRT or KA-PRT, and in MCM, the abstract winning strategy is constructed by analyzing support sets.

Semantics plugin On the lowest level of this hierarchy are those semantics definitions unique to each individual game. They are coded as plug-in for referee and strategy functions in semantics games. To add a new game, user only needs to supply the relative small plugins.

Table 3 shows the implementation cost in term of lines of code broken down by the levels of semantics hierarchy. The cost of game if implemented separately can be estimated by summing up the related components along the semantics hierarchy. For the bisimulation game, this would be about 1488 lines

Level in Hierarchy	Function	Lines of Code
Abstract game	History Maintainance	48
	Overall control/ User interface	587
Semantics game	EQM	547
	MCM	155
Plugins	Bisim./sim. plugin	306
	Weak bisim./sim. plugin	617
	Lang. equiv./preordering plugin	447
	Failure. Equiv./preordering plugin	604

Table 3: lines of code in each level of semantics hierarchy

of code. As indicated in Chapter 11, EQM/PRM can be merged into a single module because of the similarity shared by two and the fact that KA-PRT encodes the evidence for both equivalence checking and preordering checking. Equivalence and preordering plugins also can be merged. If we count in this factor, the semantics hierarchy helps reduce more than two thirds of the coding workload.

Chapter 13

Conclusion

We present a framework for supporting efficient and checker-independent post-verification analyses. The central idea is to define and use abstract proof structures as checker-independent evidence. This evidence encodes the proof by which a checker establishes its result. We provide the formalisms by which this evidence can be encoded uniformly independent of checkers being used. The choices of formalisms ensure that the evidence can be,

1. Extracted from a variety of existing checkers without compromising their complexities.

2. Verified within the time complexity of checkers which generates these evidence.

3. Used as the basis for performing a variety post-verification analyses.

The evidence we chose for model checking is *support set*, an characterized solution for boolean equation system. The evidence for equivalence checking is *partition refinement tree*. An extended version of partition refinement tree, Kernel-Auxiliary partition refinement tree, can also be used to encode the evidence for preordering checking.

The framework provide an checker-independent mechanism to certify verification results by checking the validity of evidence submitted by checkers. Since the certification can be performed without compromising overall complexities of verification tools, the framework provides us a fast and reliable way to check verification results.

Abstract proof structure can be used as the basis for a variety of post-verification analyses, including some traditional diagnostic schema, for example, counterexample and witness, and novel diagnostic schema, for example, vacuity detection. Among a wide range of diagnostic information which can be built on this framework, property-checking games have our special attention. Games provide an interactive and dynamic way to diagnose design systems. Nevertheless, implementing games separately turns out to be a very tedious work because (1) different verification semantic requires a unique definition of game (2) for every checker, games have to be adjusted to use the proof inside its unique infrastructure. We solve (2) by building games on abstract proof structures irrelevant to structures of individual checker. To solve (1), we introduce a semantics hierarchy. The features shared by various games have been placed on the proper level in this hierarchy. The hierarchy help reduce the implement cost by more than two-third, and provide users the consistent user interface, backtracking, and other good features.

The framework is designed to use a variety of checkers. We introduced two new algorithms for μ-calculus model checking and bisimulation/simulation checking. Both algorithms achieve superior performance in practice. The first algorithm is a local algorithm which have the same time and space complexities as the known best global algorithm. The second one introduces a partition refinement-based approach to compute simulation/bisimulation simultaneously.

The framework is introduced in the domain of model checking and equivalence checking. The sections with (*) extends the framework to handle preordering checking.

Related work The idea of retaining evidence during model checking as a basis for justifying the result has appeared in several recent publications. [34] uses a distinguished solution to alternation-free boolean equation system, called *extended boolean graphs* (EBGs), to encode the proof structures. EBGs can be viewed as a special case of support set in the alternation-free fragment of μ-calculus. Even closer to this work is that in [39], which uses deductive proofs to encode evidence for model-checking in the modal mu-calculus. That paper also discusses some of the same applications mentioned here for deductive information; a technical point of departure, however, is that deductive proofs in that setting require extra information in form of *ranking information* which records information on the number of "approximations" of outer variables that an inner variable depends on. This requirement plays the same role as the circularity restriction for support sets: in fact, the ranking information specifies the position of a variable in a dependency loop. With this extra information verifying the validity of proofs is easier than the for support sets. An obvious drawback is that storing ranking information requires additional space, and it also requires model checkers to maintain the information about numbers of approximations for variables. This information is not typically computed by on-the-fly (local) algorithms due to its top-down evaluation fashion. Therefore, it is not clear how ranking information can be collected for local algorithms. On the other hand, support sets require only dependency information, which is computed by both global and local algorithms.

On the theoretic aspect, researchers defined different *proof skeletons* for model-checking problem presented in different formalism, and indicate that solving model-checking problem is equivalent to establishing these proof skeleton. These skeletons include decision function in μ-calculus [46], tableau proof in tableau-based mode checkers [45], or a conjunctive (disjunctive) boolean equation system as a proof skeleton for a boolean equation system [33]. Nevertheless, few work has been done on practical aspects of these skeleton as evidences for justifying model-checking results. For example, the research emphasizes on the existences of these skeleton as the solution to model-checking problem, instead of using them to certify verification results and performing some useful analyses. Because of the same reason, these skeletons are "beyond doubt", meaning the correctness of these proof skeletons are not arguable. In practice, however, in order that the validity checking can be performed efficiently, these proof skeletons must be carefully characterized in term of constrains it has to hold.

In equivalence checking, refinement-based algorithms have been study for a long time [25, 40]. Encoding the evolution of partitions in these algorithms as a tree is a quite nature idea for implementations. For example, [19] uses binary tree as underlying data structure in discussing an extension of Paige-Tarjan algorithm. Although most of properties defined in definition 8.1.1 are familiar to those who have experience on refinement-based algorithm, the well-founded order required by (iii) gives the last piece we need to ensure the correctness of PRT as the evidence for equivalence checking. Spontaneous effort has been made to build diagnostic information on trees formed by equivalence checkers, for example, CWB-NC used this technique. Compared with tradition method in which distinguishing formula are formed during equivalence checking, our framework gives diagnostic routine more flexibility because diagnostic routine is relatively independent of checkers.

Recently games are used as an interactive diagnostic schema for verification. [43] shows that bisimularity and model checking can be characterized by games reminiscent of Ehrenfeucht-Fraissé games. [42] describes an abstract game, which is related to the top level of hierarchy. [42] is interested in an abstract game subsuming all the property-checking games, while our approach also explains how this abstract game can be interpreted as a real game.

[10] show how other equivalence-checking problems can be reduced to computing bisimularity. This work provides the theoretical base for equivalence game module which abstracts many other equivalence-checking games.

Future Directions

A Client-Server Model The ultimate goal of this research is to build a client/server architecture for diagnostic routines, such that model checkers are running on high-end servers and producing APS s as results, while user interface and post-verification analyses can run on low-cost workstations. Lots of interesting works can be done in this topic. For example, we need to standardize the data format of APSs, and determine what the mapping information should be retained during the translation of the verification problem to an "abstract" version whose solution can be captured by APS s. Here, the mapping information is important because we want diagnostics to be presented in the original form of the verification problem.

Evidence-based Symbolic model checking Symbolic model checking [35] has been shown as a successful approach for handling larger value-pass systems. Symbolic tools like SMV [35] have proven its practical importance in hardware design. Basically dependency sets are sets of states. Storing and processing sets of states is one of strong points of symbolic approach. It would be very interesting to see how symbolic approach help reduce the space usage of storing support sets, and how support sets can be extracted from symbolic model checkers.

Proof carrying code The idea of proof carrying code is that mobile code carries a formal proof to show that it satisfies a predefined set of safety rules. A code consumer then uses a validator to verify whether the proof is valid hence the code is safe to execute. At a technical level there are great similarities between proof carrying code and evidence-based verification. Currently compilers must be modified to produce proof attesting to a predefined set of safety rules. If we can introduce model-checking technique into compiler design, the result will be a compiler coupled with a model checker which can generate proofs attesting to any input property. Moreover, the new scheme will enjoy the richness of formalism used in model checking, which can encode other interesting properties besides safety.

Chapter 14

Appendix

A. Proof for the Soundness and Completeness of Support Set

Definition 14.0.1, lemma 14.0.2, 14.0.3, 14.0.4, and 14.0.5 are from [33].

Definition 14.0.1 *Let*

1. $\mathcal{E} = (\sigma_1 X_1 = f_1) \cdots (\sigma_m X_m = f_m)$
2. $\mathcal{E}' = (\sigma_1 X_1 = g_1) \cdots (\sigma_m X_m = g_m)$

Then $\mathcal{E} \leq \mathcal{E}'$ *if and only if* $f_i \leq g_i$ *for* $1 \leq i \leq m$, *where* $f_i \leq g_i$ *if and only if* $f_i(\theta) \leq g_i(\theta)$ *for all the environment* θ.

Lemma 14.0.2 *Suppose* $f(X)$ *is a monotonic boolean function in the single variable* X, *Then,* $\mu X.f(X) = f(\mathbf{0})$ *and* $\nu X.f(X) = f(\mathbf{1})$.

Lemma 14.0.3 *Let* $\mathcal{E}$ *and* $\mathcal{E}'$ *be two equation system such that* $\mathcal{E}' \leq \mathcal{E}$, *then* $[\![\mathcal{E}']\!](\theta) \subseteq [\![\mathcal{E}]\!](\theta)$ *for any environment* θ.

Lemma 14.0.4 *Let*

1. $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$ be boolean equation systems.

2. $\sigma_1 X_1 = f$, $\sigma_1 X_1 = f'$, $\sigma_2 X_2 = g$ be boolean equations, where $f' = f[X_2/g]$.

3. $[\![\mathcal{E}_1(\sigma_1 X_1 = f)\mathcal{E}_2(\sigma_2 X_2 = g)\mathcal{E}_3]\!]\theta \stackrel{def}{=} \theta_1$

4. $[\![\mathcal{E}_1(\sigma_1 X_1 = f')\mathcal{E}_2(\sigma_2 X_2 = g)\mathcal{E}_3]\!]\theta \stackrel{def}{=} \theta_2$

then $\theta_1 = \theta_2$.

Lemma 14.0.5 *Given a boolean equation system* $\mathcal{E}$ *and an environment* θ*, there exists a boolean equation system* $\mathcal{E}'$ *and* $\mathcal{E}''$ *with the properties:*

1. $\mathcal{E}'$ *is conjunctive.*

2. $\mathcal{E}' \leq \mathcal{E}$

3. $[\![\mathcal{E}']\!]\theta = [\![\mathcal{E}]\!]\theta$

For $\mathcal{E}''$ *the dual property hold:*

1. $\mathcal{E}''$ *is disjunctive,*

2. $\mathcal{E}'' \geq \mathcal{E}$

3. $[\![\mathcal{E}'']\!]\theta = [\![\mathcal{E}]\!]\theta$

where an boolean equation system is conjunctive (disjunctive) if the connectives on the right-hand side of an equation are $\wedge$ *(*$\vee$*).*

Lemma 14.0.6 *Given a close boolean equation system* $\mathcal{E} = \{\sigma_1 X_1 = f_1 \cdots \sigma_m X_m = m\}$ *and a canonical support set* $\Gamma = \langle r, X, \Xi\rangle$ *for* $\mathcal{E}$*, the following properties hold for every* $\langle X_i, \xi_i\rangle \in \Xi$*,*

1. $\xi_i \subseteq_r [\![\mathcal{E}^{(i+1)}]\!](\xi_i)$

2. $[\![\mathcal{E}^{(i)}]\!](\xi_i)(X_i) = r$

Proof: We consider case $r = \mathbf{1}$. Case $r = \mathbf{0}$ will hold dually. First, we construct a boolean equation system $\mathcal{E}_1 = (\sigma_1 X_1 = g_1) \cdots (\sigma_m = g_m)$ as follows,

1. g_k is a disjunct of f_k such that $g_k(\xi_k) = \mathbf{1}$, if $\langle X_k, \xi_k \rangle \in \Xi$.

2. Otherwise, g_k=f_k.

Property I of definition 3.1.1 guarantees the existence of such a disjunct in 1. Clearly $\mathcal{E}_1 \leq \mathcal{E}$, by Lemma 14.0.3 it's sufficient to just prove the following,

$(1')$ $\xi_i \subseteq [\![\mathcal{E}_1^{(i+1)}]\!](\xi_i)$

$(2')$ $X_i \in [\![\mathcal{E}_1^{(i)}]\!](\xi_i)$

for each $\langle X_i, \xi_i \rangle \in \Xi$. We use the induction on the structure of $\mathcal{E}_1$. Base case $i = m$ is straightforward. Let's take case $i = k$ in which $\langle X_k, \xi_k \rangle \in \Xi$, there are two scenarios,

1. $\sigma_k = \mu$. Hence $X_k \notin \xi_k$ by the circularity restriction. Denote $\theta = [\![\mathcal{E}_1^{(k+1)}]\!](\xi_k)$, then $[\![\mathcal{E}_1^{(j)}]\!](\theta) = \theta$ for all $j > k$ by the semantics of equation system. To prove property (1'), it's sufficient to show that $X_j \in \theta$ for every X_j such that $X_j \in \xi_k$ and $j > k$. Next we use another (inner) induction on $k + 1 \leq j \leq m$ to prove it,

   Base case (i.e., j=k+1): if $X_{k+1} \notin \xi_k$, then it stands trivially. Otherwise, $\xi_{k+1} \subseteq \xi_k$ because $X_{k+1} \in \xi_k$ and Γ is canonical. Hence $[\![\mathcal{E}_1^{(k+1)}]\!]\xi_{k+1} \subseteq [\![\mathcal{E}_1^{(k+1)}]\!]\xi_k$, therefore, by the (outer) induction hypothesis on $2'$, $X_{k+1} \in [\![\mathcal{E}_1^{(k+1)}]\!]\xi_{k+1} \subseteq [\![\mathcal{E}_1^{(k+1)}]\!]\xi_k = \theta$.

   Induction step: we examine only case $X_j \in \xi_k$ because induction step stands trivially if $X_j \notin \xi_k$. We have $\xi_j \subseteq \xi_k$ because $X_j \in \xi_k$ and

Γ is canonical. If we are able to establish that $X_l \in \theta$ whenever $l \leq j$ and $X_l \in \xi_j$, then we would have $[\![\mathcal{E}_1^{(j)}]\!](\xi_j) \subseteq [\![\mathcal{E}_1^{(j)}]\!](\theta)$ and hence $X_j \in [\![\mathcal{E}_1^{(j)}]\!](\theta)$ by the induction hypothesis on (2'). We need to prove both of following cases: $(1)(l < k+1 \wedge X_l \in \xi_j) \Rightarrow X_l \in \theta$. By the definition of θ, θ and ξ_k contain the same set of variables on $\{X_1, \cdots, X_k\}$, i.e., $(l < k+1 \wedge X_l \in \xi_k) \Rightarrow X_l \in \theta$. Since $X_j \in \xi_k$ and $j > k$, $\xi_j \subseteq \xi_k$. Hence we proved $(l < k+1 \wedge X_l \in \xi_j) \Rightarrow X_l \in \theta$. (2) $(k+1 \leq l < j \wedge X_l \in \xi_j) \Rightarrow X_l \in \theta$. The (inner) induction hypothesis states that $k+1 \leq l < j \wedge X_l \in \xi_k \Rightarrow X_l \in \theta$. Since $\xi_j \subseteq \xi_k$ as shown above, we have $(k+1 \leq l < j \wedge X_l \in \xi_j) \Rightarrow X_l \in \theta$. Now we have proven $[\![\mathcal{E}_1^{(j)}]\!](\xi_j) \subseteq [\![\mathcal{E}_1^{(j)}]\!](\theta) = \theta$. By the (outer) induction hypothesis on $2'$, $X_j \in [\![\mathcal{E}_1^{(j)}]\!](\xi_j) \subseteq [\![\mathcal{E}_1^{(j)}]\!]\theta = \theta$.

To show the property $(2')$, Note that $X_k \notin \xi_k$ by the circularity restriction and,

$$
\begin{aligned}
([\![\mathcal{E}_1^k]\!](\xi_k))(X_k) &= \mu X_k . g_k([\![\mathcal{E}_1^{(k+1)}]\!](\xi_k)) \\
&= g_k([\![\mathcal{E}_1^{(k+1)}]\!](\xi_k[X_k/\mathbf{0}])) && \text{(Lemma 14.0.2)} \\
&= g_k([\![\mathcal{E}_1^{(k+1)}]\!](\xi_k)) && (X_k \notin \xi_k) \\
&\geq g_k(\xi_k) && \text{(Induction on } 1'\text{)} \\
&= \mathbf{1}
\end{aligned}
$$

2. $\sigma_k = \nu$. Follows the same reasoning as in 1 to prove that $X_j \in \theta$ for every X_j such that $j{>}k$ and $X_j \in \xi_k$. To show the property $(2')$, Note that

$$
\begin{aligned}
([\![\mathcal{E}_1^k]\!](\theta_k))(X_k) &= \nu X_k . g_k([\![\mathcal{E}_1^{(k+1)}]\!](\xi_k)) \\
&= g_k([\![\mathcal{E}_1^{(k+1)}]\!](\theta_k[X_k/\mathbf{1}])) && \text{(Lemma 14.0.2)} \\
&\geq g_k([\![\mathcal{E}_1^{(k+1)}]\!](\theta_k)) \\
&\geq g_k(\theta_k) && \text{(Induction on } 1'\text{)} \\
&= \mathbf{1}
\end{aligned}
$$

■

14.0.2.1 Proof of Lemma 3.1.5

Proof: We will prove the lemma under the assumption that $\Gamma = \langle r, X, \Xi \rangle$ is canonical. However the result can be easily extended to the general case because we can always convert a support set Γ to a canonical one, and two support sets induce the same environment. We will consider only case $r = 1$, while case $r = 0$ can be proven similarly.

To prove that $\theta \subseteq [\![\mathcal{E}]\!]$, we use the induction on the order of index to prove that $X_i \in [\![\mathcal{E}]\!]$ for every $X_i \in \theta$. Base case $i = 1$ stands straightforwardly by (2) of lemma 14.0.6. Now suppose that the property holds for every X_i such that $i < k$, and $X_k \in \theta$. By semantics of fixpoint equation system we have $([\![\mathcal{E}^{(k)}]\!]([\![\mathcal{E}]\!])) = [\![\mathcal{E}]\!]$. We create a "masking" environment δ such that $\delta(X_j) = \mathbf{1}$ iff $j \geq k$. By the induction hypothesis we have,

$$\theta \cap \delta \subseteq [\![\mathcal{E}]\!] \cap \delta$$

By II of definition 3.1.1, we have $\xi_k \subseteq \theta$. Then,

$$\begin{aligned}
[\![\mathcal{E}^{(k)}]\!]\xi_k &\subseteq [\![\mathcal{E}^{(k)}]\!]\theta \\
&= [\![\mathcal{E}^{(k)}]\!](\theta \cap \delta) \\
&\subseteq [\![\mathcal{E}^{(k)}]\!]([\![\mathcal{E}]\!] \cap \delta) \\
&= [\![\mathcal{E}]\!]
\end{aligned}$$

and $([\![\mathcal{E}^{(k)}]\!](\xi_k))(X_k) = \mathbf{1}$ by lemma 14.0.6, therefore, $[\![\mathcal{E}]\!](X_k) = \mathbf{1}$ ∎

14.0.2.2 Proof of Theorem 3.1.7

Proof: We will prove case $r = \mathbf{1}$, while case $r = \mathbf{0}$ can be handled similarly. By lemma 14.0.5 there exists a conjunctive boolean equation system $\mathcal{E}'$ such that $[\![\mathcal{E}']\!] = [\![\mathcal{E}]\!]$. Denote $\theta = [\![\mathcal{E}']\!]$. We construct $\Gamma = \langle \mathbf{1}, X, \Xi \rangle$ as below. $\Xi(X_i) = \xi_i$ iff $[\![\mathcal{E}']\!](X_i) = \mathbf{1}$, where $rhs(X_i) = \wedge\xi_i$ for $\mathcal{E}'$. Now we want to show Γ is a support set.

Clearly X has been defined on Ξ, and I of definition 3.1.1 holds trivially. We can use the contradiction to show that II holds. Suppose not, then we can find a X_i and a X_j such that $\xi_i = \Xi(X_i)$, $X_j \in \xi_i$ but $\Xi(X_j)$ is not defined. By the construction of Γ $[\![\mathcal{E}']\!](X_j) = \mathbf{0}$, hence $(\wedge\xi_i)([\![\mathcal{E}']\!]) = \mathbf{0}$. It follows that $[\![\mathcal{E}']\!](X_i) = \mathbf{0}$ hence $\Xi(X_i)$ should also not be defined, which contradicts to the assumption on X_i. We can also use the contradiction to show that III holds. Suppose not, we can find a loop ρ on $\underset{\Xi}{\rightarrow}$ and a X_i on ρ such that X_i has the least index on ρ and $\sigma(X_i) = \mu$. Let $\xi_i = \Xi(X_i)$, then $(\mu X_i = \wedge\xi_i) \in \mathcal{E}'$ by the construction of Γ. We have,

$$
\begin{aligned}
[\![\mathcal{E}']\!](X_i) &= \mu X.(\wedge\xi_i)([\![\mathcal{E}'^{(i+1)}]\!]\theta[X_i/X]) \\
&= (\wedge\xi_i)([\![\mathcal{E}'^{(i+1)}]\!]\theta[X_i/\mathbf{0}]) && \text{(Lemma 14.0.2)} \\
&= \mathbf{0} && \text{(a)}
\end{aligned}
$$

To see (a), let's denote $\theta' = [\![\mathcal{E}'^{(i+1)}]\!]\theta[X_i/\mathbf{0}]$. Let X_j be the node before X_i on ρ and $\xi_j = \Xi(X_j)$, then $rhs(\mathsf{X}_j) = \wedge\xi_j$ and $X_i \in \xi_j$. Therefore $\theta'(X_j) = (\wedge\xi_j)(\theta') = \mathbf{0}$. Similarly using the induction on ρ, we can shown that $([\![\mathcal{E}'^{(i+1)}]\!]\theta[X_i/\mathbf{0}])(X_k) = \mathbf{0}$ for every variable X_k on ρ except X_i. Let X_l be the node next to X_i on ρ. Hence $X_l \in \xi_i$. Since $([\![\mathcal{E}'^{(i+1)}]\!]\theta[X_i/\mathbf{0}])(X_l) = \mathbf{0}$, (a) holds. Since $\theta(X_i) = \mathbf{0}$, X_i should not be defined on Ξ hence loop ρ cannot exist. ∎

B. Screenshot of Games

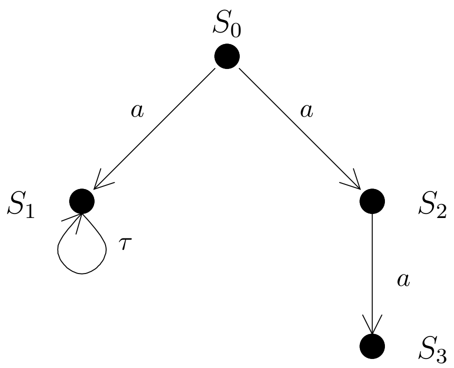


Figure 9: Sample processes for bisimulation game and μ-calculus model-checking game

```
Two agents are not related.
Computer will act as player I while user will act as player I.
################## Round 0####################
Starting configuration:
  Agent1: S0
  Agent2: S4
Referee: Computer go first to choose from either agent
Step 1: Computer chose agent 2 and made transition:
        --a-->S6
Available user options:
  Option 0: --a-->S2
  Option 1: --a-->S1
Which one do you want to choose?[(0-1) to choose, or (r)eview options]:0
Step 2: User matched computer's choice by the transition:
        --a-->S2
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 1####################
Starting configuration:
  Agent1: S2
  Agent2: S6
Referee: Computer go first to choose from either agent
Step 1: Computer chose agent 2 and made transition:
        --a-->S7
Available user options:
  Option 0: --a-->S3
Which one do you want to choose?[(0-0) to choose, or (r)eview options]:0
Step 2: User matched computer's choice by the transition:
        --a-->S3
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 2####################
Starting configuration:
  Agent1: S3
  Agent2: S7
Referee: No computer/user interfere needed.
 Reason: The second agent has action(s) a while the first agent didn't have
Computer won. Game over.
cwb-nc>
```

Figure 10: A play of bisimulation game on whether $S_0 \sim S_4$

```
Two agents are related.
User will act as player I while computer will act as player II.
################## Round 0####################
Starting configuration:
  Agent1: min X = [-]ff \/ <->X
  Agent2: S0
Referee: No computer/user interfere needed.
 Reason: Referee unrolls agent 1
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 1####################
Starting configuration:
  Agent1: [-]ff \/ <->X
  Agent2: S0
Referee: Computer go first to choose from agent 1
Step 1: Computer chose agent 1 and made transition:
        --Choose--><->X
Available user options:
  Option 0: --Stay-->S0
Which one do you want to choose?[(0-0) to choose, or (r)eview options]:0
Step 2: User matched computer's choice by the transition:
        --Stay-->S0
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 2####################
Starting configuration:
  Agent1: <->X
  Agent2: S0
Referee: User go first to choose from agent 1
Available user options:
  Option 0: --<->-->min X = [-]ff \/ <->X
Which one do you want to choose?[(0-0) to choose, or (r)eview options]:0
Step 1: User chose agent 1 and made transition:
        --<->-->min X = [-]ff \/ <->X
Step 2: Computer matched user's choice by the transition:
        --a-->S2
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 3####################
Starting configuration:
  Agent1: min X = [-]ff \/ <->X
  Agent2: S2
Referee: No computer/user interfere needed.
 Reason: Referee unrolls agent 1
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
```

```
################## Round 4####################
Starting configuration:
  Agent1: [-]ff \/ <->X
  Agent2: S2
Referee: Computer go first to choose from agent 1
Step 1: Computer chose agent 1 and made transition:
        --Choose--><->X
Available user options:
  Option 0: --Stay-->S2
Which one do you want to choose?[(0-0) to choose, or (r)eview options]:0
Step 2: User matched computer's choice by the transition:
        --Stay-->S2
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 5####################
Starting configuration:
  Agent1: <->X
  Agent2: S2
Referee: User go first to choose from agent 1
Available user options:
  Option 0: --<->-->min X = [-]ff \/ <->X
Which one do you want to choose?[(0-0) to choose, or (r)eview options]:0
Step 1: User chose agent 1 and made transition:
        --<->-->min X = [-]ff \/ <->X
Step 2: Computer matched user's choice by the transition:
        --a-->S3
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 6####################
Starting configuration:
  Agent1: min X = [-]ff \/ <->X
  Agent2: S3
Referee: No computer/user interfere needed.
 Reason: Referee unrolls agent 1
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
```

```
################## Round 7####################
Starting configuration:
  Agent1: [-]ff \/ <->X
  Agent2: S3
Referee: Computer go first to choose from agent 1
Step 1: Computer chose agent 1 and made transition:
        --Choose-->[-]ff
Available user options:
  Option 0: --Stay-->S3
Which one do you want to choose?[(0-0) to choose, or (r)eview options]:0
Step 2: User matched computer's choice by the transition:
        --Stay-->S3
Continue game?[(c)ontinue, e(x)it or (b)ack]:c
################## Round 8####################
Starting configuration:
  Agent1: [-]ff
  Agent2: S3
Referee: No computer/user interfere needed.
 Reason: Agent 2 apparently satisfies agent 1
Computer won. Game over.
```

Figure 11: A play of model checking on whether $S_0 \models \mu X.[-]ff \vee \langle - \rangle X$

Bibliography

[1] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1), 1994.

[2] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *CAV'97*, 1997.

[3] G. S. Bhat and R. Cleaveland. Efficient local model checking for fragments of the modal μ-calculus. In T. Margaria and B. Steffen, editors, *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* Vol. 1055 of *Lecture Notes in Computer Science*, pages 107–126. Springer-Verlag, Mar. 1996.

[4] G. S. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In E. M. Clarke, editor, *11th Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, NJ, July 1996. Computer Society Press.

[5] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.

[6] H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for temporal logic model checking. In *Proceedings of the Seventh International*

Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98), Lecture Notes in Computer Science. Springer-Verlag, Apr. 2001.

[7] E. Clarke, O. Grumberg, K. McMillian, and X. Zhao. Efficent generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automaion Conference*, 1995.

[8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[10] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. pages 11–23.

[11] R. Cleaveland and M. C. B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[12] R. Cleaveland, E. Madclaine, and S. Sims. A frond-end generator for the verification tools. In *Tools and Algorithms for the Construction and Analysis of Systems(TACAS'95).* Springer-Verlag, May 1995.

[13] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[14] R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In J. L. Albert, B. Monien, and M. R. Artalejo, editors, *Automata, Languages and Programming (ICALP '91)*, volume 510 of *Lecture Notes in Computer Science*, pages 127–138, Madrid, July 1991. Springer-Verlag.

[15] R. Cleaveland and B. U. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[16] M. Dam. CTL* and ECTL* as fragement of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[17] E. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science (LICS '86)*, pages 267–278, Cambridge, Massachusetts, June 1986. Computer Society Press.

[18] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings of the Seventh International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, Berlin, 1981. Springer-Verlag.

[19] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Sicence of Computer Programming*, 13:219–236, 1989/90.

[20] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.

[21] R. G.Bhat and A.Groce. Efficient model checking via büchi tableau automaton. In *??*, 2001.

[22] M. C. B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, Jan. 1985.

[23] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[24] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[25] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the 2nd ACM Symposium on the Principles of Distributed Computing*, Montreal, Canada, Aug. 1983.

[26] K. King, O. Kupferman, and M. Vardi. On the complexity of word automaton. In *FOSSACS'01*, 2001.

[27] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[28] O. Kupferman, H. Chockler, and M. Vardi. Coverage metrics for temporal logical model checking. In *TACAS proceedings*. Springer-Verlag.

[29] O. Kupferman and M. Vardi. Vacuity detection in temporal model checking, 1999.

[30] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, Mar. 2000.

[31] X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and*

Analysis of Systems (TACAS '98), Lecture Notes in Computer Science. Springer-Verlag, 1998.

[32] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In D. Dill, editor, *Proceedings of the Sixth International Conference on Computer Aided Verification (CAV '94),* Vol. 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[33] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems.* PhD thesis, Müchen, Techn-Univ., 1997.

[34] R. Mateescu. Efficient diagnostic generation for boolean equation system. Technical Report 3861, INRIA, 2000.

[35] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[36] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[37] R. Milner. Functions as processes. *Lecture Notes in Computer Science*, 443, 1990. In proceedings of International Colloquium on Algorithms, Languages and Programming 1990.

[38] Y. Moschovakis. *Descriptive Set Theory.* North Holland, 1980.

[39] K. Namjoshi. Certifying model checkers. In *Computer-Aided Verification.* LNCS 2102, 2001.

[40] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, Dec. 1987.

[41] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.

[42] P. Stevens. Abstract interpretations of games. In *VMCAI*, 1998.

[43] C. Stirling. Local model checking games. In I. Lee and S. Smolka, editors, I. Lee and S. A. Smolka, editors, *Proceedings of the Sixth International Conference on Concurrency Theory (CONCUR '95),* Vol. 962 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1995.

[44] C. Stirling. Games and modal μ-calculus. In *Proceedings of the Third International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996.

[45] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1), 1991.

[46] R. S. Streett and E. A. Emerson. An automaton theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81:249–264, 1989.

[47] L. Tan. An abstract schema for equivalence-checking games. In *The Third Internation Workshop on Verification, Model Checking, and Abstract Interpretation*, Venice, Italy, 2002.

[48] L. Tan and R. Cleaveland. Simulation revisited. In *TACAS'01*, 2001.

[49] L. Tan and R. Cleaveland. Evidence-based model checking. In *CAV'02*, 2002.

[50] L. Tan and R. Cleaveland. Two faster local algorithms for μ-calculus. In *FMSD(submitted)*, 2002.

[51] R. E. Tarjan. A hierarchical clusting algorithm using strong components. *Information Processing Letters*, 14:26–29, 1982.

[52] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.*, 1955.

[53] B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation system. In *ICALP'94*. Springer-Verlag, 1994.

Index