ABSTRACTION-BASED DEDUCTIVE-ALGORITHMIC VERIFICATION OF REACTIVE SYSTEMS

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

By
Tomás E. Uribe Restrepo
December 1998

© Copyright 1999 by Tomás E. Uribe Restrepo
All Rights Reserved

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

Zohar Manna
(Principal Adviser)

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

David L. Dill

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

Rob van Glabbeek

Approved for the University Committee on Graduate Studies:

To my parents, Fernando y Cecilia.

Abstract

This thesis presents a framework that combines deductive and algorithmic methods for verifying temporal properties of reactive systems, to allow more automatic verification of general infinite-state systems and the verification of larger finite-state ones. Underlying these *deductive-algorithmic* methods is the theory of *property-preserving assertion-based abstractions*, where a finite-state abstraction of the system is deductively justified and algorithmically model checked.

After presenting an abstraction framework that accounts for fairness, we describe a method to automatically generate finite-state abstractions. We then show how a number of other methods, including deductive rules, (Generalized) Verification Diagrams, and Deductive Model Checking, can also be understood as constructing finite-state versions of the system that are model checked.

This analysis leads to a better classification and understanding of these verification methods. Furthermore, it shows how the different abstractions that they construct can be combined. For this, we present an algorithmic *extended abstract model checking* procedure, which uses all the information produced by these methods, expressed in a finite-state format that can be easily and incrementally combined. Besides a standard safety component, the combined abstractions include extra bounds on fair transitions, well-founded orders, and constrained transition relations for the generation of counterexamples.

Thus, our approach minimizes the need for user interaction and maximizes the impact of the available automated deduction and model checking tools. Once proved, verification conditions are reused as much as possible, leaving the temporal and combinatorial reasoning to automatic tools.

This research was supported in part by the National Science Foundation under grants CCR-95-27927 and CCR-9804100, a gift from Intel Corporation, the Defense Advanced Research Projects Agency (DARPA) under NASA grant NAG2-892, ARO under grants DAAH04-95-1-0317, DAAH04-96-1-0122 and DAAG55-98-1-0471, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

Acknowledgements

Zohar Manna introduced me to the main subject of this thesis, the temporal verification of reactive systems, and supervised this research.

This thesis would not exist without the support, collaboration and feedback of Zohar and the REACT/STeP research team, present: Nikolaj Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Henny Sipma; and past: Anuchit Anuchitanukul, Eddie Chang, Luca de Alfaro, Arjun Kapur, Hugh McGuire, and Mark Pichora. Particular thanks are are due to Nikolaj Bjørner, Anca Browne, Michael Colón, and Henny Sipma as co-authors and fellow STeP developers. The results and insights obtained by working with them make up most of this thesis.

Amir Pnueli, Mark Stickel, and Richard Waldinger gave advice and encouragement. Mark Stickel provided support for a summer at SRI International, where I learned much about finite-domain satisfiability, theorem proving and verification.

As reading committee members, David Dill and Rob van Glabbeek made many corrections and suggestions for improving this thesis. Grigori Mints and Vaughan Pratt served on the orals committee and provided valuable feedback as well.

Serge Plotkin and Luca de Alfaro kept the espresso machine going. Farhad Shakeri helped keep the *other* machines going, which were also essential for this thesis. Phyllis Winkler and Maria Bharwada provided timely and efficient administrative support.

My interest in automated deduction developed thanks to Alan Frisch. Thanks are also due to Michael Genesereth, Peter Haddaway, David Page, Christian Prehofer, and Richard Scherl. For indispensable friendship and company during my years at Stanford, I thank Donald Aingworth, Marko Balabanovic, Rune Dahl, Brian Ferguson, T.K. Lakshman, Uri Lerner, Anna Patterson, Juan Carlos Quintero and Tessie Villalón, Zeynep Taspinar, and many other friends less likely to read this page, with apologies to those omitted.

My uncle, Rodrigo Restrepo, has always been an essential source of support and encouragement throughout the years. My brothers, sister, nephews, nieces and siblings-in-law, and my parents, Fernando and Cecilia, gave me the hope and support that made this work possible. If this thesis were worth the time spent away from them, each letter on every page would be priceless. For their love, generosity, patience and understanding, my gratitude is not expressible in any language.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis addresses the *formal verification of temporal properties of reactive systems.* A reactive system is a computer program, implemented in hardware or software, or a combination thereof, that maintains an ongoing interaction with its environment, which can include the physical world, human users, or other computer programs. Unlike purely functional programs, which compute a single output given a set of inputs, a reactive system does not necessarily terminate, so its computations can be viewed as infinite sequences of states.

Reactive systems feature *concurrency*: different components of the system may perform different tasks at the same time, and frequently or occasionally communicate with each other. They also feature *non-determinism*, where the next state of the system is under-specified or only partially known. The design of correct reactive systems is particularly challenging, since during their execution a large number of possibly unanticipated configurations and events can occur. *Formal verification* provides a rigorous, well-defined mathematical basis by which properties of all possible combinations of such configurations and events can be established.

Since the analysis of reactive systems requires reasoning about ongoing behavior over time, *temporal logic* has been found to be a very well-suited formalism for precisely specifying the properties that such systems should satisfy [Pnu77]. Formulas of temporal logic describe properties of infinite sequences of states. In its different varieties, temporal logic can express the ordering, inevitability, eventuality, or possibility of events. By proving that a system satisfies a temporal property, we show that *all* of its possible computations do.

Even though computations of reactive systems are infinite sequences of states, systems may be classified according to the cardinality of the set of possible states. For *finite-state* *systems*, the states are valuations of a finite number of finite-state variables, or equivalently, a finite number of bits. This includes, in particular, hardware systems of fixed size. *Infinite-state systems* feature variables with unbounded domains, typically found in software systems, such as integers, lists, trees, and other datatypes.

The verification of temporal properties for finite-state systems is decidable: *model checking* algorithms can automatically decide if a temporal property holds for a finite-state system, usually producing a *counterexample computation* if this is not the case. Counterexamples are of great use in identifying the corresponding fault in the system or its specification. (A counterexample is always more convincing than a proof, specially when the latter relies on exhaustive enumeration by computer.) However, model checking suffers from the *state explosion problem*, where the number of states to be explored grows exponentially in the size of the system description, particularly as the number of concurrent processes grows.

For general infinite-state systems, including real-time and hybrid systems, the verification problem is undecidable, and finite-state model checking techniques are not directly applicable.[1] *Deductive methods*, based on general-purpose theorem proving, apply to a wide class of finite- and infinite-state reactive systems. They provide *relatively complete* proof systems, which are guaranteed to prove any temporal property that indeed holds over the given system. Unfortunately, if the property does not hold, deductive methods do not normally provide much useful feedback to the user, who must then try to decide whether the property really fails or something is missing in the proof.

First-order logic, including its many fragments and extensions, is a natural and convenient language for expressing relationships over the unbounded data structures that make software systems infinite-state. *Automated deduction*, or *theorem proving*, provides computer-aided tools to reason about these relationships. Such tools are particularly welcome since the theorems needed are usually not particularly deep or interesting, but instead rather tedious to prove.

This thesis presents methods for combining the deductive and algorithmic approaches to verification, making model checking tools more applicable to general infinite-state systems. The goal is to make the verification of such systems more automatic (that is, less interactive), by reusing the results of deductive methods as much as possible and leaving the propositional

[1]Particular decidable classes of infinite-state systems can be model checked using specialized tools, and invariants can be automatically generated for infinite-state systems as well. These tools often use abstraction as well—see the discussion of related work in Chapter 7, particularly Section 7.3.

reasoning and temporal combinatorics to the finite-state model checking tools. The methods we present can also help verify large finite-state systems for which model checking is not feasible. Such systems may be treated as infinite-state ones, by parameterizing processes and introducing symbolic functions and constants to abstract data. Thus, very complex hardware systems might be best verified by modeling them as software.

One of the goals of this thesis is to show that the two hyphenated words in its title are mostly redundant: underlying *all* the methods for combining deductive and algorithmic verification we consider lies the notion of *abstraction*, which allows a quotient of the original state-space to be explored. We will see that the main difference between algorithmic and deductive methods is the degree of abstraction used. Many deductive methods can be understood as constructing an appropriate abstraction of the system for which (often simple) model checking can be performed. Formalizing this, we present an *assertion-based abstraction* framework that lies, implicitly, behind most deductive verification techniques (Chapter 3).

The main question that then arises is *how* such an abstraction may be found. The combinations of theorem proving and model checking discussed in this thesis differ in the ways in which the abstraction is constructed and analyzed, and are generally classified into two main groups. *Static* methods generate a system abstraction and then model check it. *Dynamic* methods construct the desired abstraction incrementally, interleaving it with the model checking process itself. (Chapter 7 classifies related work along these lines.)

We present an algorithm for automatically constructing finite-state assertion-based abstractions of possibly infinite-state systems, based on a description of its basic components provided by the user (Chapter 4). We then show how deductive verification rules and Generalized Verification Diagrams can also be understood as specifying an adequate abstraction, which is deductively justified and algorithmically model checked. Similarly, we show how in the related method of Deductive Model Checking, the abstraction is constructed and refined interactively, while the property is simultaneously model checked (Chapter 5).

We argue that viewing these methods as literal instances of abstraction has implications that are useful in practice. In Chapter 6 we show how the abstractions that these methods generate can be automatically combined and model checked, maximizing the utility of their construction, and minimizing the extra work done by the user. The verification conditions proved for the original system are reused as much as possible, using automatic tools to explore their consequences.

In all cases, the verification of infinite-state systems is still interactive (and undecidable in general), and not guaranteed to succeed or find a counterexample. However, we argue that the construction of abstractions provides an alternative to standard deductive verification that is more exploratory and incremental. Furthermore, user interaction, which will always be needed, is made less painful by maximizing the amount of model checking and deductive reasoning that is carried out automatically. We hope that it is also made more fun, by providing earlier and more interesting feedback to the user.

1.1 Model Checking vs. (?) Theorem Proving

	Algorithmic Methods	Deductive Methods	Combination
Automatic?	yes	no	sometimes
Counterexamples?	yes	no	sometimes
General infinite-state?	no	yes	yes

Table 1.1: Deductive vs. algorithmic verification

As mentioned above, model checking and deductive verification have different strengths and weaknesses, summarized in Table 1.1. Model checking [CE81, QS82] is based on the observation that checking that a formula is true in a particular model is generally easier than checking that it is true in all models (the *validity* test). In the case of reactive systems, the model of interest is the system itself, the temporal property is the formula to be checked over that model.

The provocatively titled manifesto by Halpern and Vardi [HV91] argues that model checking, rather than theorem proving, should be used in knowledge representation applications whenever possible—that is, whenever a particular model is available: a knowledge base is, in general, better described as a model than as a set of axioms, since its potential consequences can be model checked, rather than proved valid in a corresponding theory.[2]

However, we must point out that the use of theorem proving proposed in this thesis, which includes that found in most deductive verification frameworks, is *not* analogous to

[2]As [HV91] points out, for instance, *logical omniscience* is a non-problem from the model checking point of view: agents "know" all tautologies, not because they are expert logicians, but because they can check each one for the given model. (They will know many non-tautologies too, and are not expected to tell the difference.)

the application of theorem proving criticized in [HV91]: the system itself is not encoded as a logical formula. Rather, a succinct description of the system (a set of transitions) is used to generate a small number of theorems (*verification conditions*) that prove the truth of a temporal formula *relative to the particular system* in question. Thus, deductive verification is itself an indirect form of model checking. In Chapter 5, we justify this claim by showing how these deductive methods are equivalent to model checking a particular kind of abstraction.[3]

We argue that the approach described in this thesis takes advantage of the best that each method offers. Validity checking is used to prove facts about possibly infinite sets of states (under specialized theories, if applicable), where considering all the possible models of a particular formula does make a difference, for the best. Model checking is then used to check truth under the available model: a finite-state abstract system.

1.2 Sensible Uses of Formal Methods

It is generally agreed that a "*formal method*" requires three related components (e.g., [Rus93, MP95b]):

- A *mathematical model* of the systems being verified.
- A *specification language* for describing properties of systems.
- A *verification framework* for proving properties of systems.

In this way, formal methods can prove that a particular mathematical model of the system of interest satisfies certain mathematical properties (including, for instance, being equivalent to, or a refinement of, another mathematical model). By using formal methods, we increase the probability of finding subtle errors (also known as "*bugs*"). The task of specification itself can expose previously implicit and possibly misguided assumptions about the system, increasing the designer's understanding of it.

However, formal methods *cannot* prove that the mathematical model correctly reflects its real-world implementation, or, more seriously perhaps, that the properties that the model satisfies are indeed the ones that the system designers really want. Eventually, systems must

[3]This claim is not particularly surprising, but we argue that it gives a point of view that can be exploited in practice.

still be tested in the real world, as in the well-known quote from Knuth [Knu77]: *"Beware of bugs in the above code; I have only proved it correct, not tried it."*

For this reason, Henzinger [Hen96] argues that formal verification should be called "formal falsification" instead: *"The only sensible goal of formal methods is to detect the presence of errors,"* rather than certify their absence.[4] Rushby [Rus93] makes a similar point, quoting Lakatos: *"The virtue of a logical proof is not that it compels belief but that it suggests doubts."* Formal methods are valuable not because they give proofs, but because they uncover assumptions, and find bugs. We cannot prove that the real world will behave as we expect any more than we can prove that the law of gravity will continue to work tomorrow.

1.2.1 Which Airplane to Fly?

To illustrate the point more dramatically, consider the problem of choosing which of two airplanes, A and B, to use for a long trip.[5] Assume that both have been designed and built by equally competent teams of engineers, tested to the same degree, and are operated by the same crew. Assume also that several properties of the computer systems of plane B have been "verified" ("falsified" would be more accurate) using methods not unlike those proposed in this thesis. This author would then prefer to fly B, maybe even paying a (modest) fare increase proportional to the quality of the verification methods used.

However, if "formal methods" were the *only* method used to validate the systems in plane B, then this author would most definitely prefer A (or else walk, drive or swim to his destination). Formal methods should not give the designers of B any more certainty that it will behave "correctly" when flown. But they can help find more accurate notions of system "correctness" and provide more opportunities for uncovering problems in the design and bugs in the implementation.

We end this philosophical discussion with a good example of the dangers of indiscriminately applying formal reasoning to the real world, described ca. 1891:

> *Logic*, n. The art of thinking and reasoning in strict accordance with the limitations and incapacities of the human misunderstanding. The basic of logic is the syllogism, consisting of a major and a minor premise and a conclusion—thus:

[4]Perhaps the term "verification" can be retained when only mathematical objects are involved, and "falsification" or "validation" used whenever the physical world is involved.

[5]See [Rus93] for a discussion of formal methods and this particular domain. This section borrows from a discussion at a recent formal methods conference, where the very term "formal methods" was questioned. We keep it for lack of a better one.

> Major premise: Sixty men can do a piece of work sixty times as quickly as one man.
>
> Minor premise: One man can dig a posthole in sixty seconds; therefore—
>
> Conclusion: Sixty men can dig a posthole in one second.
>
> This may be called the syllogism arithmetical, in which, by combining logic and mathematics, we obtain a double certainty and are twice blessed.
>
> – Ambrose Bierce, *The Devil's Dictionary* [Bie72]

1.3 Outline

Chapter 2 presents the basic background material concerning reactive systems, temporal logic, model checking and automated deduction. Chapter 3 presents the necessary theoretical abstraction background (fairly simple, as far as these things are concerned), including fairness considerations.

Chapter 4 then presents an algorithm for generating finite-state abstractions of possibly infinite-state systems, following [CU98]. Chapter 5 discusses how the related methods of verification rules, Generalized Verification Diagrams, and Deductive Model Checking can be seen as constructing an appropriate assertion-based abstraction, making finer distinctions than those in Chapter 4. Based on this, Chapter 6 presents a practical approach to combining abstractions that benefits from the point of view and common abstraction framework presented in the previous chapters.

For Chapters 3–6, the most closely related work is discussed at the end of each chapter. Chapter 7 surveys and classifies more generally related work. Finally, Chapter 8 presents conclusions, and suggests some directions for future research.

Acknowledgements: I thank Saddek Bensalem, Jorge Cuéllar, Dennis Dams, Yassine Lakhnech and Hassen Saidi for valuable feedback, comments and discussion on subjects related to this thesis.

After many years of use, I am still delighted, amazed, and occasionally perplexed at the power and flexibility of TEX and LATEX, which make writing a document like this one possible. For this, I thank Donald Knuth and Leslie Lamport (if not for their contributions to concurrency, verification, automated deduction, the BAKERY algorithm and other such things).

Coauthorship: The work on generating finite-state abstractions (Chapter 4) was developed with Michael Colón and reported, in abbreviated form, in [CU98]. The work on Deductive Model Checking (Section 5.5) was developed with Henny Sipma and Zohar Manna [SUM99]. The presentation of Generalized Verification Diagrams in Section 5.3 is from [MBSU98], a collaboration with Anca Browne, Zohar Manna and Henny Sipma, based on their work in [BMS95]. Work with them and Luca de Alfaro [dAMSU97] clarified the connections between the different formalisms.

The use of context in the abstraction algorithm of Chapter 4 is inspired by Nikolaj Bjørner's formulation of contextual simplification in the STeP simplifier. The abstraction algorithm uses the STeP decision procedures and was implemented by Michael Colón, who also verified the examples of Chapter 4. The verification diagrams and abstract version of program BAKERY come courtesy of [BMSU98], a collaboration with Nikolaj Bjørner, Zohar Manna and Henny Sipma. Work on STeP was conducted in collaboration with Anca Browne, Nikolaj Bjørner, Eddie Chang, Michael Colón, Bernd Finkbeiner, Arjun Kapur, Zohar Manna, Mark Pichora and Henny Sipma.

Chapter 2

Background

This chapter presents the basic notions used in the rest of this thesis. First, we describe Kripke structures (Section 2.1.1), which model reactive systems, together with the fair transition systems that describe them (Section 2.1.2) and the temporal logic that specifies their properties (Section 2.2). Model checking and deductive verification are briefly presented in Sections 2.3–2.6. The STeP tool, which implements most of the verification methods described in this chapter and thesis, is described in Section 2.7. Since all of this material is quite well-covered elsewhere, we try to be brief, referring the reader to the appropriate sources for details.

2.1 Reactive Systems

2.1.1 Kripke Structures

A *reactive system* $\mathcal{S} : \langle \Sigma, \Theta, R \rangle$ is given by a set of *states* Σ, a set of *initial states* $\Theta \subseteq \Sigma$, and a *transition relation* $R \subseteq \Sigma \times \Sigma$. If $\langle s_1, s_2 \rangle \in R$, state s_2 is a successor of s_1, and the system can move from s_1 to s_2. A system $\mathcal{S}$ can be identified with the corresponding *Kripke structure*,[1] or *state-space*: this is the directed graph whose vertices are the elements of Σ and whose edges connect each state to its successor states. Each state in Σ will be identified by a valuation of a set of system variables, as we formalize below.

[1]This formalism has its origins in the semantics for modal logic given by Kripke, where each state is a *possible world*.

If Σ is finite, $\mathcal{S}$ is said to be *finite-state*. Describing large or infinite state-spaces explicitly is not feasible. Therefore, the state-space is *implicitly* represented in some other form: a hardware description, a program, or an ω-automaton. We now describe the basic representation for reactive systems we will use.

Relational Operators

We identify binary relations with sets of pairs, where $\langle s_1, s_2\rangle \in R$ iff $R(s_1, s_2)$ holds. In general, for a binary relation $R : \Sigma \times \Sigma$ and a set $S \subseteq \Sigma$, we define

$$\begin{aligned}
\mathit{post}(R, S) &\stackrel{\text{def}}{=} \{x \mid R(s, x) \text{ for some } s \in S\}\\
\mathit{pre}(R, S) &\stackrel{\text{def}}{=} \{y \mid R(y, s) \text{ for some } s \in S\}\\
\mathit{wpc}(R, S) &\stackrel{\text{def}}{=} \widetilde{\text{pre}}(R, S) \stackrel{\text{def}}{=} \overline{\mathit{pre}(R, \overline{S})}
\end{aligned}$$

where $\overline{S} = \Sigma - S$. Thus, *wpc* corresponds to the *weakest liberal precondition* predicate transformer, giving those states where the relation is *guaranteed* to reach a state in S, if a related state exists at all. In contrast, *pre* gives those states where it is *possible* to reach an element of S by the relation R.

2.1.2 Fair Transition Systems

Fair transition systems [MP91b, MP95b] are a convenient formalism for specifying finite- and infinite-state reactive systems. They are "low-level" in the sense that many other formalisms can be translated or compiled into them.[2] The representation relies on an *assertion language* to represent sets of states, usually based on first-order logic.

The state-space of the system is determined by a set of *system variables* $\mathcal{V}$, where each variable has a given domain (e.g., booleans, integers, recursive datatypes, or reals).

Definition 2.1.1 (Assertion) *A first-order formula whose free variables are a subset of $\mathcal{V}$ is an* assertion*, and represents the set of states that satisfy it. For an assertion φ, we say that $s \in \Sigma$ is a φ-*state *if $s \models \varphi$, that is, φ holds given the values of $\mathcal{V}$ at the state s.*

Assertions are sometimes called "state-formulas," but we avoid this term to prevent confusion with the CTL* state formulas of Section 2.2 (which are temporal formulas that

[2] In practice, higher-level languages are desirable since, among other things, they lessen the chance of inadvertently writing down vacuous specifications, as noted, e.g., in [McM93].

are evaluated over a state). We reserve the term *assertion* for formulas with no temporal operators.

We write the standard boolean connectives as $\wedge$, $\vee$, $\neg$, and $\leftrightarrow$ (equivalence). An *atomic formula* is one whose only subformula is itself.

Remark 2.1.2 (Assertion language) In practice, an assertion language other than first-order logic can be used. The basic requirements on the assertion language are the ability to represent predicates and relations, and automated support for validity and satisfiability checking (which need not be complete). $\square$

Besides first-order logic, examples of other suitable assertion languages include *ordered binary decision diagrams* (OBDD's) [Bry86] and their variants, for finite-state systems (see Section 2.3), and specialized constraint languages such as those used in *constraint logic programming* (CLP) [JL87].

The initial condition is now expressed as an assertion, characterizing the set of states the system can be in at the start of a computation. The transition relation R is described as a set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is described by its *transition relation* $\tau(\mathcal{V}, \mathcal{V}')$, a first-order formula over the set of system variables $\mathcal{V}$ and a *primed set* $\mathcal{V}'$, indicating their values at the next state.

Definition 2.1.3 (Transition system) *A* transition system $\mathcal{S} : \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ *is given by a set of* system variables $\mathcal{V}$*, an* initial condition Θ*, expressed as an assertion over* $\mathcal{V}$*, and a set of* transitions $\mathcal{T}$*, each an assertion over* $(\mathcal{V}, \mathcal{V}')$*.*

In the associated Kripke structure, each state in the state-space Σ *is a possible valuation of* $\mathcal{V}$*. We write* $s \models \varphi$ *if assertion* φ *holds at state* s*, and say that* s *is a* φ-state*. A state* s *is initial if* $s \models \Theta$*. There is an edge from* s_1 *to* s_2 *if* $\langle s_1, s_2 \rangle$ *satisfy* τ *for some* $\tau \in \mathcal{T}$*.*

Remark 2.1.4 (Edge labels in a Kripke structure) The edges of a Kripke structure can be labeled with transitions in one of two ways: In the first, there is at most one edge between any pair of states $\langle s_1, s_2 \rangle$, labeled by the set of transitions that can go from s_1 to s_2. In the second, each one of these transitions labels a separate edge from s_1 to s_2. In both cases, we say that a transition is *taken* along the edges it labels.

These two conventions are, of course, equivalent. We will use one or the other according to convenience, but not both at the same time. $\square$

The Kripke structure is also called the *state transition graph* for $\mathcal{S}$. Note that if the domain of a system variable is infinite, the state-space is infinite as well, even though the *reachable state-space*, the set of states that can be reached from Θ, may be finite.

The global transition relation is the disjunction of the individual transition relations: $R(s_1, s_2)$ iff $\tau(s_1, s_2)$ holds for some $\tau \in \mathcal{T}$. We also write $\mathcal{T}(s_1, s_2)$ in this case. The predicate transformers *pre*, *post* and *wpc* can now be expressed in terms of assertions:

$$\begin{aligned}
pre(\tau, \varphi) &\stackrel{\text{def}}{=} \exists \mathcal{V}'. (\tau(\mathcal{V}, \mathcal{V}') \wedge \varphi(\mathcal{V}')) \\
post(\tau, \varphi) &\stackrel{\text{def}}{=} \exists \mathcal{V}_0. (\tau(\mathcal{V}_0, \mathcal{V}) \wedge \varphi(\mathcal{V}_0)) \\
wpc(\tau, \varphi) &\stackrel{\text{def}}{=} \forall \mathcal{V}'. (\tau(\mathcal{V}, \mathcal{V}') \rightarrow \varphi(\mathcal{V}')) \ .
\end{aligned}$$

For assertions φ and ψ and transition τ, we write

$$\{\varphi\}\, \tau\, \{\psi\} \stackrel{\text{def}}{=} (\varphi(V) \wedge \tau(\mathcal{V}, \mathcal{V}')) \rightarrow \psi(\mathcal{V}') \ .$$

This is the *verification condition* that states that every τ-successor of a φ-state must be a ψ-state. For an expression φ, we define φ' to be the result of replacing each free variable x of φ with x'. Thus, $\psi(\mathcal{V}')$ can be written as ψ'. For a set of expressions S, let $S' \stackrel{\text{def}}{=} \{\varphi' \mid \varphi \in S\}$.

Note 2.1.5 (Implicit universal quantifiers) *Throughout this thesis, first-order formulas will be, implicitly, universally quantified. Thus, a verification condition f will stand for its* universal closure $\forall x_1. \ldots . \forall x_n. f$, *where $x_1, \ldots, x_n$ are the free variables in f. In contrast, existential quantifiers will always be explicit.*

A *run* of $\mathcal{S}$ is an infinite path through the Kripke structure that starts at an initial state, i.e., a sequence of states $(s_0, s_1, \ldots)$ where $s_0 \in \Theta$ and $R(s_i, s_{i+1})$ for all $i \geq 0$. If $\tau(s_i, s_{i+1})$ holds, then we say that transition τ is *taken* at s_i. A transition is *enabled* if it can be taken at a given state. The states where τ can be taken are characterized by the formula *enabled*, where

$$enabled(\tau) \stackrel{\text{def}}{=} \exists \mathcal{V}'. \tau(\mathcal{V}, \mathcal{V}') \ .$$

Note 2.1.6 (Assertions as sets) *Since an assertion can be identified with the set of states where it is true, we sometimes write $f_1 \subseteq f_2$ when f_1 implies f_2, for assertions f_1 and f_2. Similarly, we may write $f_1 \cap f_2$ for $f_1 \wedge f_2$, $f_1 \cup f_2$ for $f_1 \vee f_2$, or $\overline{f}$ for $\neg f$.*

$$\textbf{local} \quad y_1, y_2 \quad : \textbf{integer where } y_1 = 0 \ \wedge \ y_2 = 0$$

$$\left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{ll}\ell_0: & \textbf{noncritical}\\ \ell_1: & y_1 := y_2 + 1\\ \ell_2: & \textbf{await } (y_2 = 0 \vee y_1 \leq y_2)\\ \ell_3: & \textbf{critical}\\ \ell_4: & y_1 := 0\end{array}\right]\end{array}\right] \| \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{ll}m_0: & \textbf{noncritical}\\ m_1: & y_2 := y_1 + 1\\ m_2: & \textbf{await } (y_1 = 0 \vee y_2 < y_1)\\ m_3: & \textbf{critical}\\ m_4: & y_2 := 0\end{array}\right]\end{array}\right]$$

–P1– –P2–

Figure 2.1: Program BAKERY for 2-process mutual exclusion

2.1.3 Fairness and Computations

Our computational model represents concurrency by *interleaving*: at each step of a computation, a single action or transition is executed. The transitions from different processes are combined in all possible ways to form the set of computations of the system. *Fairness* expresses the constraint that certain actions cannot be forever prevented from occurring—that is, that some actions that the system can take do have a fair chance of being taken.

Describing R as a set of individual transition relations is convenient for modeling fairness:

Definition 2.1.7 (Fair transition system) *A* fair transition system (FTS) *is one where each transition is marked as* just *or* compassionate*. A just (or* weakly fair*) transition cannot be continually enabled without ever being taken; a compassionate (or* strongly fair*) transition cannot be enabled infinitely often but taken only finitely many times. A* computation *is a run that satisfies these fairness requirements (if any exist). Otherwise, the run is* unfair.

As usual, compassionate transitions are also considered to be just. Below, systems described using R but not $\mathcal{T}$ are assumed to have no fairness requirements.

To ensure that R is total on Σ, so that sequences of states can always be extended to infinite sequences, we assume an *idling transition*, with transition relation $\mathcal{V} = \mathcal{V}'$. The set of all computations of a system $\mathcal{S}$ is written $\mathcal{L}(\mathcal{S})$, suggesting a language of infinite strings whose alphabet is the set of states of $\mathcal{S}$.

Example 2.1.8 (BAKERY) *Figure 2.1 shows program* BAKERY*, which implements Lamport's* bakery *mutual exclusion algorithm, in the Simple Programming Language (SPL) of*

[MP95b]. SPL programs can be compiled into the corresponding fair transition systems, following the semantics of each of the SPL constructs, as done by STeP (Section 2.7). To each process corresponds a finite-domain control variable. *For* BAKERY*, the control variables π_1 and π_2 range over locations $\{\ell_0, \dots, \ell_4\}$ and $\{m_0, \dots, m_4\}$. All transitions are assumed to be just, except for those associated with the* **noncritical** *statements. Program* BAKERY *is an infinite-state system, since variables y_1 and y_2 can grow beyond any bound.*

We write m_i and ℓ_j to abbreviate the assertions $\pi_1 = m_i$ and $\pi_2 = \ell_i$, indicating where the control for each process resides.

2.1.4 Clocked Transition Systems

The reactive system model presented earlier assumes that concurrency can be modeled by interleaving, and that time advances by discrete steps of unspecified duration. However, fair transition systems can be also be used to model reactive systems with *real-time* constraints. A *clocked transition system* (CTS) [MP96] is a fair transition system $\mathcal{S} : \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$, whose system variables are partitioned into a set of *discrete variables* D and a set of real-valued *clock variables* C. Instead of an idling transition, $\mathcal{T}$ includes a *tick transition*, which is the only transition that can advance time. Clocked transition systems contain a *master clock* T, which can only be changed by the *tick* transition, where the initial condition Θ should imply $T = 0$.

The progress of time is restricted by a *time-progress condition* Π, an assertion over D and C. The transition relation for *tick* is:

$$\rho_{tick}: \quad \exists \Delta > 0. \quad \left(\begin{array}{c} (D' = D) \wedge (C' = C + \Delta) \\ \wedge \\ \forall t \in [0, \Delta]. \Pi(D, C + t) \end{array} \right) \quad .$$

Here, $C' = C + \Delta$ stands for

$$c_1' = c_1 + \Delta \wedge \ldots \wedge c_k' = c_k + \Delta \quad ,$$

indicating that all the clock variables are incremented by Δ. The expression $\Pi(D, C + t)$ stands for $\Pi(d_1, \dots, d_j, c_1 + t, \dots, c_k + t)$, where $D = \{d_1, \dots, d_j\}$, and is used to indicate that the progress condition should hold for the system at all times between the current

$$
\textbf{local } x : \{0,1,2\} \ \textbf{ where } x = 0
$$

$$
P_1 :: \begin{bmatrix} \ell_0\text{:} & \textbf{await } x = 0 \\ \ell_1\text{:} & x := 1 \\ \ell_2\text{:} & \textbf{skip} \\ \ell_3\text{:} & \textbf{await } x = 1 \\ \ell_4\text{:} & \textbf{critical} \end{bmatrix} \qquad || \qquad P_2 :: \begin{bmatrix} m_0\text{:} & \textbf{await } x = 0 \\ m_1\text{:} & x := 2 \\ m_2\text{:} & \textbf{skip} \\ m_3\text{:} & \textbf{await } x = 2 \\ m_4\text{:} & \textbf{critical} \end{bmatrix}
$$

Figure 2.2: Simple version of Fischer's mutual exclusion algorithm.

time T and $T+\Delta$. Clocked transition systems usually do not feature fairness constraints. Instead, upper bounds on the time that can pass before an enabled transition is taken can be specified as part of the time-progress condition.

The computations of a CTS are the runs where time grows beyond any bound. Incorrect specifications of real-time and hybrid systems may force time to stop, in the sense that it does not diverge. The property of *non-Zenoness* states that every run prefix can be extended to a computation where time diverges, and should be proved as a sanity check in specifications. This is an existential property (see Section 3.5). See [BMSU99] for a discussion of non-Zenoness in the STeP and CTS framework.

Example 2.1.9 (Fischer) *Figure 2.2 shows a fragment of Fischer's real-time mutual exclusion algorithm, from [MP95a] (the complete program is described in Section 4.6). The algorithm assumes uniform positive bounds L and U on the time each process can wait before executing its next statement: an enabled transition must wait at least L and at most U before being taken. If $2L > U$, the algorithm guarantees that both processes are never in their critical sections simultaneously.*

Two clock variables, c_1 and c_2, are used to measure the time that the current transition of each process has been enabled. The initial condition is the assertion

$$
\Theta : \ell_0 \wedge m_0 \wedge c_1 = 0 \wedge c_2 = 0 \wedge x = 0 \wedge T = 0 \ .
$$

As before, each statement corresponds to a transition. For example, statement ℓ_1 yields transition τ_{ℓ_1}, with relation

$$
\rho_{\ell_1} : \begin{pmatrix} \ell_1 \ \wedge \ c_1 \geq L \ \wedge \\ \ell_2' \ \wedge \ c_1' = 0 \ \wedge \ x' = 1 \end{pmatrix} \wedge \ \pi_2' = \pi_2 \ \wedge \ c_2' = c_2 \quad .
$$

This transition can be taken only if $c_1 \geq L$, enforcing the lower bound, and resets c_1 to 0.

The time-progress condition, which goes into the tick transition, is $\Pi : c_1 \leq U \wedge c_2 \leq U$, stating that time cannot advance if c_1 or c_2 would then grow beyond U. Intuitively, this forces the discrete transitions from each process to be taken within time U of the last time that the respective clock was reset.

Clocked transition systems illustrate the expressive power of the fair transition system formalism. However, after writing transition systems down, we must be able to specify their properties and reason about them.

2.2 Temporal Logic

So here are the questions:
Is time long, or is it wide?
. . . Are things getting better, or are they getting worse?

— *Laurie Anderson, "Same Time Tomorrow"*

We will use temporal logic to specify properties of reactive systems [Pnu77, MP91b]. STeP uses *linear-time temporal logic* (LTL) for temporal specifications, which we will use throughout most of this thesis. However, since we will also occasionally refer to existential properties, for generality we present here the temporal logic CTL* [EH86]. The logic CTL* includes both the branching-time *computation tree logic* (CTL) and LTL in a strictly more expressive language. The syntax of CTL* is given by a set of atomic predicates, p, and *path* and *state* formulas, recursively defined as:

$$\begin{array}{rcl} state & := & p \mid \neg p \mid state \vee state \mid state \wedge state \mid \mathsf{A}path \mid \mathsf{E}path \\ path & := & state \mid path \vee path \mid path \;\wedge\; path \\ & & \mid \bigcirc path \mid \diamondsuit path \mid \square path \mid (path \;\mathcal{U}\; path) \mid (path \;\mathcal{W}\; path) \end{array}$$

For simplicity, in the definition below we assume *negation-normal form*, where negation is only used at the level of atomic predicates. A *literal* is an atomic formula or its negation. For a literal l, the set of states for which l holds (l-states) is written as $||l||$. Technically, the $||l||$ function is part of the Kripke structure, but notice that it is fixed once the system variables $\mathcal{V}$ and the assertion language are given.

State formulas are evaluated for each node in a Kripke structure, while path formulas are evaluated over infinite sequences $(s_0, s_1, \ldots)$ of states. For a state s in structure K, we write paths(K, s) for the set of all infinite paths in K starting at s.

Definition 2.2.1 (CTL* temporal semantics) *For structure K and state s:*

$$
\begin{array}{lll}
(K,s) \vDash l & \text{if} & s \in ||l|| \textit{ for a literal } l \\
(K,s) \vDash \varphi_1 \wedge (\vee) \varphi_2 & \text{if} & (K,s) \vDash \varphi_1 \textit{ and (or) } (K,s) \vDash \varphi_2 \\
(K,s) \vDash \mathsf{A}\varphi & \text{if} & \pi \vDash \varphi \textit{ for all } \pi \in \text{paths}(K,s) \\
(K,s) \vDash \mathsf{E}\varphi & \text{if} & \pi \vDash \varphi \textit{ for some } \pi \in \text{paths}(K,s)
\end{array}
$$

Path formulas are evaluated as follows: For a path $\pi : (s_0, s_1, \ldots,)$, we write π^n for the suffix path $(s_n, s_{n+1}, \ldots,)$.

$$
\begin{array}{lll}
\pi \vDash \mathit{state} & \text{if} & (K, s_0) \vDash \mathit{state} \textit{ for state formula state} \\
\pi \vDash \varphi_1 \wedge (\vee) \varphi_2 & \text{if} & \pi \vDash \varphi_1 \textit{ and (or) } \pi \vDash \varphi_2 \\
\pi \vDash \bigcirc \varphi & \text{if} & \pi^1 \vDash \varphi \\
\pi \vDash \Box \varphi & \text{if} & \pi^k \vDash \varphi \textit{ for all } k \geq 0 \\
\pi \vDash \Diamond \varphi & \text{if} & \pi^k \vDash \varphi \textit{ for some } k \geq 0 \\
\pi \vDash \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{if} & \pi^j \vDash \varphi_2 \textit{ for some } i \geq 0 \textit{ and } \pi^k \vDash \varphi_1 \textit{ for all } 0 \leq k < i \\
\pi \vDash \varphi_1 \,\mathcal{W}\, \varphi_2 & \text{if} & \pi \vDash (\varphi_1 \,\mathcal{U}\, \varphi_2) \text{ or } \pi \vDash \Box \varphi_1
\end{array}
$$

$\mathcal{U}$ is the *until* operator, where $p \,\mathcal{U}\, q$ states that q will eventually hold, and p will be true at least up to (but not including) the point where q holds. The *await* formula $p \,\mathcal{W}\, q$ allows for the possibility that q never happens, in which case p must always be true.

CTL and LTL: *Computation tree logic* (CTL) is obtained by restricting the state and path modalities to always go together in pairs, where A and E must always be followed by $\bigcirc$, $\Diamond$, $\Box$ or $\mathcal{U}$. *Linear-time temporal logic* (LTL) is obtained by eliminating the A and E path quantifiers, leaving only path formulas that are evaluated over a single sequence of states. A CTL* formula is *universal* (resp. *existential*) if it only contains occurrences of the A (resp. E) path quantifier. $\forall$CTL* (resp. $\exists$CTL*) is the logic that contains only universal (resp. existential) CTL* formulas. An LTL formula φ holds for (K, s) if $(K, s) \vDash \mathsf{A}\varphi$, so LTL properties can be considered as universal.

The other boolean connectives such as $\rightarrow$ and $\leftrightarrow$ are defined as usual. Following [MP91b, MP95b], we sometimes write $p \Rightarrow q$ as an abbreviation for $\Box(p \rightarrow q)$. Past temporal operators, such as $\boxminus\, p$ ("So-far p") can be introduced as well [MP95b].

We can now define when a reactive system satisfies a temporal property φ:

Definition 2.2.2 (System validity) *For a system $\mathcal{S} : \langle \Sigma, \Theta, \mathcal{T} \rangle$ with underlying Kripke structure K, we say that φ is* $\mathcal{S}$-valid *, written $\mathcal{S} \models \varphi$, if $(K, n) \models \varphi$ for all $n \in \Theta$, where path formulas are evaluated over* fair paths *only, that is, those that are computations of the system (see Section 2.1.3).*

Remark 2.2.3 (Quantifiers and temporal logic) In *propositional temporal logic*, atomic formulas are treated as boolean variables. Since we will describe unbounded domains, we go beyond the propositional case by referring to possibly unbounded system variables, and allowing explicit quantifiers under the condition that temporal operators do not appear within the scope of a quantifier. Such formulas are called *state-quantified* temporal properties [MP95b].

The value of a *rigid variable* cannot change over time, while *flexible variables* can. System variables are the only flexible variables we will use, and we will not use flexible quantification in this thesis. However, for a temporal property φ, we will be able to prove the $\mathcal{S}$-validity of properties of the form $\forall x . \varphi[x]$, for a *rigid* auxiliary variable x, by proving $\varphi[N]$ for an arbitrary constant N of the same type as x. Similarly, we can prove $\exists x . \varphi[x]$ by proving $\varphi[x_0]$ for some particular value x_0. □

Example 2.2.4 (Non-Zenoness) *The* non-Zenoness *of a real-time system (Section 2.1.4) is expressed in CTL as:*

$$\forall \epsilon > 0 \;.\; \forall t \;.\; \mathsf{A}\,\Box(T = t \;\rightarrow\; \mathsf{E}\,\Diamond(T \geq t + \epsilon))$$

where ϵ and t are rigid *auxiliary variables. That is, for every $\epsilon > 0$ and at every reachable state, there is a possible future state where the global clock T has increased by at least ϵ. This property is equivalent to*

$$\exists \epsilon > 0 \;.\; \forall t \;.\; \mathsf{A}\,\Box(T = t \;\rightarrow\; \mathsf{E}\,\Diamond(T \geq t + \epsilon))\,,$$

so we can prove it by showing the $\mathcal{S}$-validity of

$$\mathsf{A}\,\Box(T = t \;\rightarrow\; \mathsf{E}\,\Diamond(T \geq t + \epsilon_0))$$

for arbitrary t and some fixed ϵ_0 [BMSU97].

For LTL, a convenient alternative definition of $\mathcal{S}$-validity can be formulated in terms of the *models* of φ:

Definition 2.2.5 (LTL models) *For an LTL formula, $\mathcal{L}(\varphi)$ is the set of all models of φ, that is, all paths π such that $\pi \vDash \varphi$.*

Proposition 2.2.6 (LTL system validity) *$\mathcal{S} \vDash \varphi$ for an LTL formula φ if and only if all the computations of $\mathcal{S}$ are models of φ, that is,*

$$\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi) \ .$$

Note 2.2.7 ($\mathcal{S}$-validity of verification conditions) *When we require that a premise of a rule or a verification condition be valid, we will mean for it to be $\mathcal{S}$-valid; thus, axioms and previously proven invariants concerning $\mathcal{S}$ can be used to establish it.*

Safety and Progress

Following [MP91b, MP95b], we distinguish between *safety* and *progress* properties. LTL safety properties have the following semantic characterization: if a safety property φ fails for a sequence π (that is, $\pi \vDash \neg\varphi$), then there is a finite prefix π_0 of π such that φ is false for any extension of π_0. Intuitively, safety properties state that certain "bad" states cannot be reached. Safety properties of the form $\Box\, p$ for an assertion p are called *invariances*. In this case, a prefix π_0 that violates the property is a sequence of states whose final state satisfies $\neg p$.

Progress properties, on the other hand, generally state that "good" things will happen, e.g., particular states are eventually reached, or appear infinitely often. Note that a system with no fairness requirements can idle forever, so it will not satisfy any non-trivial progress property unless fairness assumptions are encoded in the property itself. For more details on LTL, see [MP91b]. For a comprehensive treatment of CTL* and related logics, see [Eme90].

Example 2.2.8 (Temporal properties of BAKERY) *The main temporal properties of program* BAKERY *of Figure 2.1, written in LTL, are:*

- Mutual exclusion: *The two processes are never in their critical sections at the same time:*

$$\Box \neg(\ell_3 \wedge m_3) \ .$$

- One-bounded overtaking: *If one process is waiting to enter its critical section, the other process can only do so at most one before the first one does:*

$$\Box(\ell_2 \ \rightarrow \ \neg m_3 \, \mathcal{W} \, (m_3 \, \mathcal{W} \, (\neg m_3 \, \mathcal{W} \, \ell_3))) \ .$$

- Accessibility: *If one process is waiting to enter the critical section, it will eventually do so:*

$$\Box(\ell_1 \rightarrow \Diamond \, \ell_3) \ .$$

Mutual exclusion for the Fischer program of Figure 2.2 is expressed by the LTL formula $\Box \neg(\ell_4 \wedge m_4)$.

The first two properties are safety properties, and are valid even if the system decided to idle forever at any point. Accessibility is a progress property, and relies on the fairness of transitions.

2.3 Finite-State Model Checking

The question addressed in the rest of this thesis is how to establish $\mathcal{S} \vDash \varphi$ given a reactive system $\mathcal{S}$ and a temporal property φ. For finite-state systems, *model checking* answers this question by a systematic exploration of the Kripke structure (state-space) of $\mathcal{S}$: the Kripke structure is the model, and φ is the property being checked over that model.

Model checking for finite-state systems was introduced by Clarke and Emerson [CE81] and independently by Queille and Sifakis [QS82]. The classic model checking algorithms for CTL recursively label each system state with the formulas it satisfies, starting with the atomic propositions in φ and analyzing more complex subformulas of φ at each step. For example, once the states that satisfy a subformula f have been identified, the states that satisfy $\mathsf{E}\,\Box(f)$ (namely, the ones where f is always true in some possible computation) are identified by finding all the strongly connected components in the state-space where f is always true that are reachable by a path where f also holds (see [CG87]).

Model checkers such as SPIN [Hol91], COSPAN [Kur94], SMV [McM93], Murφ [MD93], and those in STeP (Section 2.7) take as input, essentially, a finite-state fair transition system and a temporal formula in some variety or subset of CTL*, and automatically check that the system satisfies the property.

2.3.1 Explicit-state LTL Model Checking

Much of the work we present only requires the "black box" use of model checkers, without having to change their internal workings. However, in Section 6.4 we describe a model checking procedure tailored to finite-state abstractions that can benefit from extra information about the concrete system. Therefore, we now briefly summarize an instance of automata-theoretic model checking, based on Chapter 5 of [MP95b], on which the procedure of Section 6.4 will be based.

The Formula Tableau

The *temporal tableau* T_φ of an LTL formula φ is a finite-state ω-automaton over infinite strings whose language is $\mathcal{L}(\varphi)$, the set of models of φ [Pra80, MP95b]. It is represented as a directed graph, where each node is labeled by an *atom*, which is a set of assertions and temporal formulas expected to hold whenever a model reaches this node. Two nodes n_1 and n_2 are connected with a directed edge (n_1, n_2) if the formulas in n_2 can hold at a state following one that satisfies the formulas in n_1. For example, if n_1 contains $\bigcirc p$ and $\square q$, then n_2 should contain p, q, and $\square q$. A tableau atom is *initial* if it describes a possible initial state of a model of φ. In particular, an initial atom should include φ itself. (When past temporal operators are used, an initial atom should not require the existence of previous states.)

A *strongly connected subgraph* (SCS) is a subgraph with at least one edge where each node is reachable from every other node. If φ is satisfiable, then there must be an SCS in T_φ that is reachable from an initial atom. Such an SCS must satisfy an additional property: if a model satisfies $\diamondsuit q$ or $p \,\mathcal{U}\, q$ at some state, it must in fact satisfy q at this or another state later on. Thus, we say that an SCS S is *fulfilling* if whenever a node in S contains $\diamondsuit \chi$ or $\varphi \,\mathcal{U}\, \chi$, then some node in S contains χ. This defines the *acceptance condition* of the tableau, when viewed as an ω-automaton (see Section 5.3).

Theorem 2.3.1 (Temporal Satisfiability and Tableaus) *φ is satisfiable iff there is a fulfilling, reachable SCS in T_φ.*

In practice, it suffices to consider only *maximal* strongly connected components: if an SCS is not fulfilling, all of its sub-SCS's are not fulfilling either. For details on LTL tableau constructions, see [KMMP93, MP95b, McG95].

The tableau serves as a guide for finding computations of $\mathcal{S}$ that satisfy $\neg\varphi$, that is, counterexamples to the $\mathcal{S}$-validity of φ. We do this by taking the *product* of the tableau and the Kripke structure of $\mathcal{S}$:

Definition 2.3.2 (Product graph) *Given two directed graphs*

$$\mathcal{G}_1 : \langle N_1, E_1 \rangle\,, \quad \mathcal{G}_2 : \langle N_2, E_2 \rangle\ ,$$

their product *is the graph*

$$\mathcal{G}_1 \times \mathcal{G}_2 : \langle N_1 \times N_2, E' \rangle$$

where E' is the set of edges

$$\{(\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle) \mid (a_1, b_1) \in E_1 \ \mathit{and}\ (a_2, b_2) \in E_2\}\ .$$

Edges in $\mathcal{G}$ are labeled with the corresponding edge-label pair. If $\mathcal{G}_1$ and $\mathcal{G}_2$ have initial nodes, the initial nodes in the product are those of the form $\langle n_1, n_2 \rangle$ where n_1 is initial in $\mathcal{G}_1$ and n_2 is initial in $\mathcal{G}_2$.

Figure 2.3 shows an outline of the classic model checking procedure for finite-state systems. The algorithm builds the product between the Kripke structure for $\mathcal{S}$ and $T_{\neg\varphi}$ and checks whether it contains a fulfilling SCS (with respect to the tableau atoms) that also satisfies the fairness requirements associated with $\mathcal{S}$. If such an SCS is reachable from an initial node a counterexample computation can be produced; otherwise, φ is $\mathcal{S}$-valid. Thus, the finite-state model checking algorithm is:

1. Compute the product graph $\mathcal{G} : \mathcal{S} \times T_{\neg\varphi}$, removing nodes $\langle s, A \rangle$ where the state s does not satisfy the assertions in the atom A. The graph $\mathcal{G}$ is sometimes called the $(\mathcal{S}, \neg\varphi)$-*behavior graph* [MP95b].

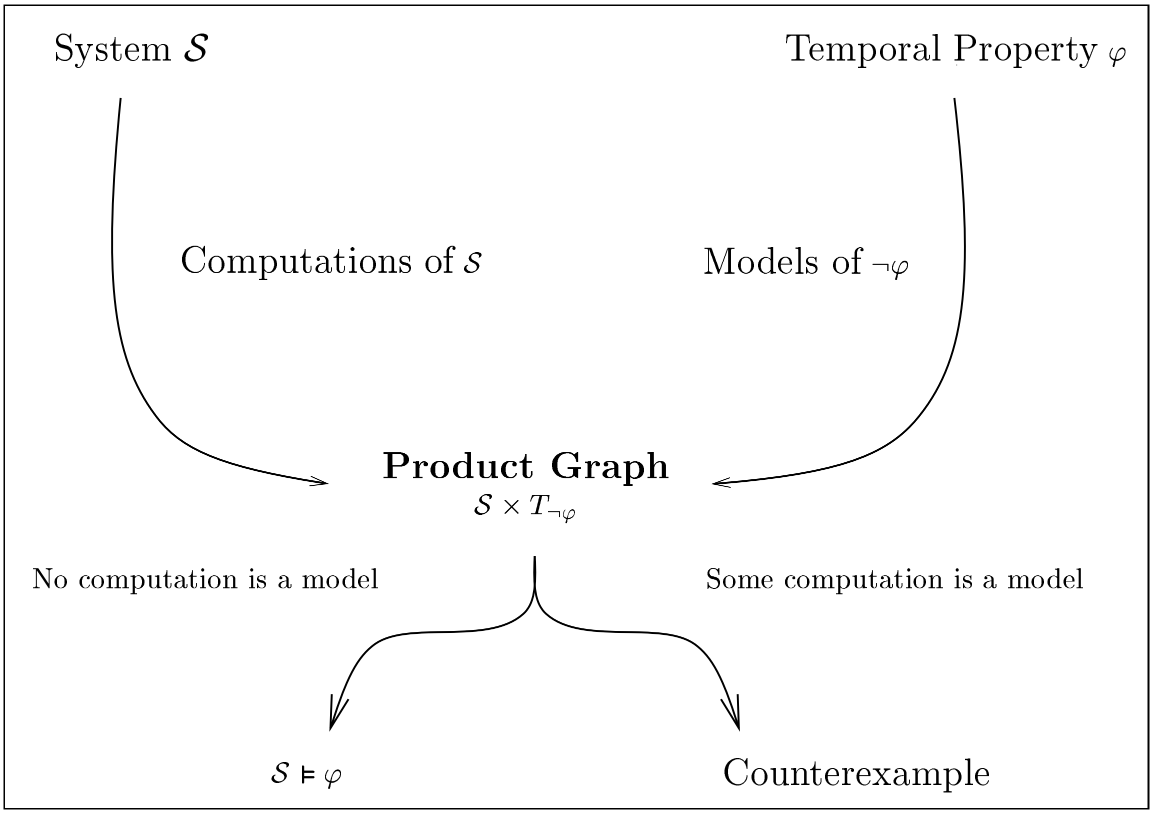

Figure 2.3: Outline of explicit-state LTL model checking

Edges in the product graph can be labeled by single or multiple transitions depending on the convention used (see Remark 2.1.4).

2. An SCS S is *accepting* if its underlying tableau SCS is accepting. An SCS is *just* (resp. *compassionate*) if every just (resp. compassionate) transition that is enabled at all (resp. some) nodes in S is taken along some edge in S.

3. If there is an accepting, just and compassionate SCS in $\mathcal{G}$ that is reachable from an initial node, return this SCS, and a path that leads to it, as a counterexample.

   Otherwise, report that $\mathcal{S} \models \varphi$.

Note that if an SCS is not just, its sub-SCS's are not just either, so it suffices to consider MSCS's. However, this is not the case for compassion: a sub-SCS can be compassionate while the larger one is not.

Many variants and improvements of this basic algorithm are possible, as we will see. For instance, if the product graph is to be explicitly constructed, this should be done starting with the initial states of both graphs, pruning incompatible $\langle s, A\rangle$ nodes as the reachable state-space is explored in a *forwards propagation* of the transition relation. When proving safety properties, *backwards propagation* is possible, constrained by previously proved system invariants, to check if the "bad" states are reachable from an initial state [Bjø98b].

The discussion of related work in Chapter 7 addresses model checking for infinite-state systems. The following sections present the other two main approaches to finite-state model checking.

2.3.2 Automata-theoretic Model Checking

The algorithm of the previous section is a simple instance of the *automata-theoretic* approach to model checking, where the system and its specification are viewed as finite-state ω-automata that describe languages over infinite trees or strings [Kur86, VW86, Kur94].

In the linear-time case, the system $\mathcal{S}$ is identified with an ω-automaton that recognizes the language $\mathcal{L}(\mathcal{S})$. Similarly, the property or specification to be verified can be expressed as an automaton that recognizes the language $\mathcal{L}(\varphi)$. To prove that φ holds for S, we must show that $\mathcal{L}(\varphi) \subseteq \mathcal{L}(\mathcal{S})$ (see Proposition 2.2.6). If both φ and $\mathcal{S}$ can be expressed as finite automata, this can be decided by purely algorithmic means. We will return to such automata in Section 5.3, where we discuss Generalized Verification Diagrams.

In the case of LTL, the automata describe languages over infinite words. For CTL and CTL*, automata over *infinite trees* can be used [Kup95]. For a survey on ω-automata, see [Tho90].

2.3.3 Symbolic Model Checking

For a finite-state system $\mathcal{S}$, the complexity of model checking depends on the size of the formula being checked (linear for CTL and exponential for LTL and CTL*) and the size of the state-space of $\mathcal{S}$ (linear for all three logics). While the temporal formulas of interest are usually small, the size of the state-space can grow exponentially in the size of its description, e.g., as a circuit, program, or fair transition system. This is known as the *state explosion problem*, where the number of system states limits the practical applicability of model checking algorithms.

Model checking techniques that construct and explore states of the system, one at a time, such as the one in Section 2.3.1, are called *explicit-state*. In contrast, *symbolic model checking* combats the state-explosion problem by using specialized formalisms to represent sets of states. Ordered Binary Decision Diagrams (OBDD's), introduced by Bryant [Bry86], are an efficient data structure for representing boolean functions and relations. McMillan [McM93] used OBDD's to represent the transition relation of finite-state systems, as well as subsets of the system's state-space. The efficient algorithms for manipulating OBDD's [BRB90] can be used to compute predicate transformations, such as pre- and post-condition operations, over the transition relation and large, implicitly represented sets of states.

The resulting model checking algorithms can be formulated using the *propositional μ-calculus* [Koz83], which expresses fixpoint relations over finite domains. The temporal operators can be given a fixpoint characterization; for instance, $\mathsf{E} \diamondsuit p$ is the least fixed point of the function $f(y) = p \vee \mathsf{E} \bigcirc y$ [BCM+92]. A survey of model checking for the μ-calculus is presented in [BCJM96]. Symbolic model checking algorithms for CTL can be adapted to LTL as well [CGH94].

The success of symbolic model checkers is often best measured in terms of the number of bits that describe the system state, rather than the number of reachable states. Symbolic model checking extends the size of finite-state systems that can be analyzed, and is particularly successful for hardware systems. However, it is still restricted to finite-state systems of fixed size; thus, for instance, a system may be model checked for up to 128 bits, but not for 129. The size of the OBDD representation of particular functions can grow exponentially in the number of boolean variables [Bry91], leading to what can be called the *OBDD explosion problem*, where the model checker runs out of memory before the user runs out of time. In these cases, efficient explicit-state model checkers are preferred, such as Murφ [Dil96]. To further extend the range of symbolic model checking, extensions of BDDs are being used to move from bit-level to word-level representations [CFZ95].

2.4 Deductive Verification

Figure 2.4 presents the *general invariance rule*, G-INV, which can be used to prove the $\mathcal{S}$-validity of formulas of the form $\square p$ for an assertion p. The premises of the rule are first-order verification conditions. If they can be proved to be valid, the temporal conclusion must hold for the system $\mathcal{S}$.

	For assertions φ and p,
I1.	$\Theta \rightarrow \varphi$
I2.	$\{\varphi\}\ \tau\ \{\varphi\}$ for each $\tau \in \mathcal{T}$
I3.	$\varphi \rightarrow p$
	$\mathcal{S} \vDash \Box p$

Figure 2.4: General invariance rule G-INV

An assertion is *inductive* if it is preserved by all the system transitions and holds at all initial states. The invariance rule relies on finding an inductive auxiliary assertion φ that *strengthens* p, that is, φ implies p. The invariant p may not be inductive, but φ must be. Thus, some transitions might be able to make p false, but the rule ensures that the states where this happens do not appear in any run of $\mathcal{S}$.

The soundness of the rule is obvious: if φ holds initially and is preserved by all transitions, it will hold for every reachable state of $\mathcal{S}$. If p is implied by φ, then p will also hold for all reachable states. Rule G-INV is also *relatively complete*, that is, complete relative to the underlying first-order reasoning: if p is an invariant of $\mathcal{S}$, then the strengthened assertion φ always exists [MP95b]. Assuming that we have a complete system for proving valid assertions, then we can prove the $\mathcal{S}$-validity of any $\mathcal{S}$-valid temporal property. Note, however, that proving invariants is undecidable for general infinite-state systems, and finding such a φ can be non-trivial.

Other verification rules can be used to verify different classes of temporal formulas, ranging from safety to progress properties [MP91a]. These deductive methods are relatively complete and yield a direct proof of any $\mathcal{S}$-valid temporal property. However, they may require substantial user guidance to succeed, and do not produce counterexample computations when the property fails.

Graphical formalisms facilitate the task of guiding and understanding a deductive proof. *Verification diagrams* [MP94, BMS95] provide a graphical representation of the verification conditions needed to establish a particular temporal formula. In Chapter 5 we will discuss deductive verification, including rules and diagrams, in the context of abstraction.

2.4.1 Well-founded Orders

Not all properties of infinite-state systems can be proved by simple finite-state abstractions or rules such as G-INV: well-foundedness arguments must be used to reason about loops that depend on unbounded data variables.

For example, consider the BAKERY program of Figure 2.1 and the property

$$\varphi : \Diamond \Box \neg((\ell_0 \vee \ell_1) \wedge (m_0 \vee m_1)) \;\rightarrow\; \Diamond(\max(y_1, y_2) > N)$$

where N is an arbitrary positive integer [Sip98, BMSU98]. This property states that $\max(y_1, y_2)$ will grow beyond N, provided the two processes are never at locations ℓ_0 or ℓ_1 (resp. m_0 or m_1) at the same time. BAKERY has no finite-state abstraction that can prove φ for an arbitrary N, unless we use well-foundedness arguments to add extra fairness constraints, as we will do in Chapters 5 and 6. We will return to this property in Chapter 6, where we describe a model checking algorithm for finite-state abstractions that can use well-founded orders to reduce the number of abstract computations that are potential counterexamples.

To express well-founded relations, we use *ranking functions*:

Definition 2.4.1 (Ranking functions) *A* binary relation $\succ$ *over a domain* $\mathcal{D}$ *is a subset of* $\mathcal{D} \times \mathcal{D}$*, where we write* $s_1 \succ s_2$ *iff* $\langle s_1, s_2 \rangle \in \succ$ *.*

A binary relation $\succ$ *is* well-founded over $\mathcal{D}$ *if there are no infinite sequences of elements* $e_1, \ldots, e_n, \ldots$ *in* $\mathcal{D}$ *such that*

$$e_1 \succ e_2 \succ \ldots \succ e_n \succ \ldots$$

A ranking function δ *is a mapping from system states into a* well-founded domain $(\mathcal{D}, \succ)$*. We write* $x \succeq y$ *iff* $x \succ y$ *or* $x = y$ *.*

We say that an assertion over unprimed and primed system variables $\rho(\mathcal{V}, \mathcal{V}')$ *is* well-founded *if it characterizes a well-founded relation over* $\Sigma \times \Sigma$ *.*

Ranking functions can be described using an extended or specialized assertion language, e.g. [SdRG89].

2.5 Invariant Generation

Invariants state that some assertion p holds at all the reachable states of the system. Once established, they can be very useful in all forms of deductive and algorithmic verification.

In the deductive case, verification conditions need not be valid in general, but only valid with respect to the invariants of the system. Therefore, deductive verification usually begins by proving a series of invariants of increasing strength, where each one is used as a lemma to prove subsequent ones. In the algorithmic case, invariants can constrain the set of states explored in symbolic model checking [PS96] and backwards explicit-state model checking [Bjø98b], for extra efficiency.

Finally, invariants can also be used to produce better system abstractions in deductive-algorithmic verification, as we will see in Chapter 4.

2.6 Automated Deduction

Automated deduction ("theorem proving") is used to formally check the validity of the verification conditions generated by deductive verification rules and diagrams. As mentioned above, the validity of these formulas can be established with respect to invariants of the system, which can be previously proved or generated automatically. In general, axioms and lemmas about the system or the domain of computation can also be used. We now summarize the main types of automated deduction we will use:

2.6.1 Decision Procedures

Establishing the validity of first-order formulas is, in general, itself undecidable. However, most verification conditions refer to particular theories that describe the domain of computation, such as linear arithmetic, lists, arrays and other data types.

Decision procedures provide efficient, specialized validity checking for particular theories. For instance, equality need not be axiomatized, but its consequences can be efficiently derived by specialized methods based on congruence closure. Similarly, specialized methods can efficiently reason about integers, lists, bit vectors and other datatypes frequently used in system descriptions [Bjø98a].

Thus, using appropriate decision procedures can be understood as specializing the assertion language to the particular domains of computation used by the system being verified.

2.6.2 Validity Checking

Given an assertion language, an essential operation is to determine the validity of particular assertions, possibly with respect to given axioms, previously proven invariants or lemmas.

Decision procedures usually operate only at the *ground* level, where no quantification is allowed. Program features such as parameterization and the *tick* transition (Section 2.1.4) introduce quantifiers in verification conditions. Fortunately, the required quantifier instantiations are often "obvious" in that they use instances that can be provided by the decision procedures themselves. [BSU97, Bjø98a] presents an integration of first-order reasoning and decision procedures that can automatically prove many verification conditions that would otherwise require the use of an interactive prover.

Throughout this thesis, we will use validity checkers as "black boxes," and will not need to change or access their internal workings. Thus, we could also use systems such as the Stanford Validity Checker (SVC) [BDL96], an efficient checker specialized to handle large ground formulas that occur in hardware verification.

2.6.3 Interactive Theorem Proving

Automatic proof methods for first-order logic and certain specialized theories are necessarily incomplete, not guaranteed to terminate, or both. In practice, automatic verification tools abandon completeness and focus instead on quickly deciding particular classes of problems that are both tractable and likely to appear when verifying realistic systems.

To achieve completeness, *interactive theorem proving* is used. The user selects rules that reduce complex subgoals to simpler ones, and provides instantiations for first-order quantifiers that the automated system would otherwise have difficulty finding. The automatic decision procedures are still used whenever possible (e.g., to simplify formulas and close off branches). *Tactics* are used to automate repetitive sequences of steps.

PVS [OSR93, ORR$^+$96] is an interactive theorem prover that includes higher-order logic, decision procedures, and tactics. Other interactive theorem provers (with varying degrees of automated support) that have been used for reactive system verification include the Boyer-Moore prover [BM88], HOL [GM93] and the STeP system, described below.

2.7 The STeP System

The Stanford Temporal Prover (STeP) implements the verification methodologies described in this chapter, including explicit-state and symbolic LTL model checking, automatic invariant generation, verification rules and diagrams, and automatic and interactive theorem proving.

Fair transition systems are the basic system description formalism of STeP. Real-time systems can be represented as clocked transition systems (see Section 2.1.4); *hybrid systems*, where discrete and continuous components interact, can be described as *phase transition systems*. STeP automatically translates SPL programs such as BAKERY from Figure 2.1 into the underlying transition systems, using control variables to represents the locations of the program. STeP includes a number of automatic invariant generation methods, based on abstract interpretation [BBM97] (see Section 7.1.2).

STeP is described in [BBC$^+$95, BBC$^+$96, MBB$^+$98]. Cases studies for STeP are presented in [BMSU99] (a real-time system, modularly described), [BLM97] (a parameterized fault-tolerant system), and [MS98, MCF$^+$98, Sip98] (hybrid systems). STeP is being extended to include modular specification and verification [FMS98, MCF$^+$98].

Chapter 3

Abstraction

Abstraction reduces the verification of a system property to the verification of a related property over a simpler system. Abstraction is often proposed to allow the verification of infinite-state systems, by constructing finite-state abstract systems that can be model checked. Abstraction can also mitigate the state explosion problem in the finite-state case, by constructing an abstract system with a more manageable set of states.

The general framework of *abstract interpretation* (see Section 7.1.1) defines an abstraction whose state-space can be represented, manipulated and approximated in specialized ways that could not be directly applied to the original system. Originally designed for deriving safety properties in static program analysis, this framework has recently been extended to include reactive systems and general temporal logic, e.g., [BBLS92, CGL94, LGS$^+$95, DGG94, CIY95, Dams96]. Below, we follow the notation of [Dams96] as much as possible. We discuss this and other related work in Chapter 7.

3.1 An Abstraction Framework

In general terms, abstraction reduces the verification of a temporal property φ over a *concrete system* $\mathcal{S}$, to checking a related property $\varphi^{\mathcal{A}}$ over a simpler, *abstract system* $\mathcal{A}$.

Definition 3.1.1 (Generic abstraction) *Given a class of temporal properties $\mathcal{P}$ and two systems $\mathcal{S}$ and $\mathcal{A}$, we say that:*

- $\mathcal{A}$ *is a* weakly preserving *abstraction of* $\mathcal{S}$ *for* $\mathcal{P}$ *if for any* $\varphi^{\mathcal{A}} \in \mathcal{P}$,

$$\text{If } \mathcal{A} \models \varphi^{\mathcal{A}} \text{ then } \mathcal{S} \models \varphi$$

- $\mathcal{A}$ *is a* strongly preserving *abstraction of* $\mathcal{S}$ *for* $\mathcal{P}$ *if for any* $\varphi^{\mathcal{A}} \in \mathcal{P}$,

$$\mathcal{A} \models \varphi^{\mathcal{A}} \text{ if and only if } \mathcal{S} \models \varphi$$

Strong preservation does not leave much room for generating simpler systems. For instance, if all CTL properties are to be preserved, the two systems must be *bisimilar* (see, e.g., [BCG88, vG90, dNV95, LGS+95]). Thus, weak preservation is more often used.

Based on the framework of *abstract interpretation* [CC77], the abstract system is defined in terms of an *abstract domain*, a set of states $\Sigma_{\mathcal{A}}$ that includes a partial order $\preceq$, where $a_1 \preceq a_2$ if a_1 is a "more precise" abstract state than a_2. Such abstractions are often presented in terms of *Galois connections*. In a frequently used case, abstract states represent sets of concrete states, and the two posets being connected are the *power set* of concrete states, $\mathcal{P}(\Sigma_{\mathcal{C}})$, ordered by set inclusion, and an abstract domain $\Sigma_{\mathcal{A}}$ ordered by $\preceq$ which, being abstract, will remain under-specified for the time being. This is the framework we will use as well:

Definition 3.1.2 (Galois connection) *A* Galois connection *from* $(\mathcal{P}(\Sigma_{\mathcal{C}}), \subseteq)$ *to* $(\Sigma_{\mathcal{A}}, \preceq)$ *is a pair of functions*

$$(\alpha : \mathcal{P}(\Sigma_{\mathcal{C}}) \to \Sigma_{\mathcal{A}},\ \gamma : \Sigma_{\mathcal{A}} \to \mathcal{P}(\Sigma_{\mathcal{C}}))$$

such that $\alpha(x) \preceq y$ *iff* $x \subseteq \gamma(y)$.

The *abstraction function* α maps each set of concrete states to an abstract state that represents it. The *concretization function* $\gamma : \Sigma_{\mathcal{A}} \to \mathcal{P}(\Sigma_{\mathcal{C}})$ maps each abstract state to the set of concrete states that it represents.

Definition 3.1.3 (Concretization of sets and relations) *For a set of abstract states* S*, we define*

$$\gamma(S) \stackrel{\text{def}}{=} \bigcup_{a \in S} \gamma(a) \ .$$

For a relation $\rho_{\mathcal{A}}$ over $\Sigma_{\mathcal{A}} \times \Sigma_{\mathcal{A}}$, we define

$$\gamma(\rho_{\mathcal{A}}) = \{\langle s_1, s_2\rangle \mid s_1 \in \gamma(a_1) \text{ and } s_2 \in \gamma(a_2) \text{ for some } \langle a_1, a_2\rangle \in \rho_{\mathcal{A}}\} \ .$$

We say that $\rho_{\mathcal{A}}$ over-approximates *a concrete relation $\rho_{\mathcal{C}}$ if $\rho_{\mathcal{C}} \subseteq \gamma(\rho_{\mathcal{A}})$.*

As an alternative to Definition 3.1.2, α and γ should be monotonic, $\alpha(\gamma(a)) \preceq a$ for all $a \in \Sigma_{\mathcal{A}}$, and $c \subseteq \gamma(\alpha(c))$ for all $c \in \mathcal{P}(\Sigma_{\mathcal{C}})$. The functions α and γ determine each other, and are usually expressed in terms of γ. Given γ, the abstraction function can be uniquely defined as

$$\alpha(c) = glb\,\{a \in \Sigma_{\mathcal{A}} \mid c \subseteq \gamma(a)\} \ ,$$

where *glb* is the greatest lower bound with respect to $\preceq$ in the abstract domain. This, provided that such greatest lower bounds exist and γ is *continuous*:

$$glb(\gamma(S)) = \gamma(glb(S)) \ .$$

Conversely, given an abstraction function α, the concretization function is uniquely defined as

$$\gamma(a) = \bigcup \{c \subseteq \Sigma_{\mathcal{C}} \mid \alpha(c) \preceq a\} \ .$$

If $\alpha(\gamma(a)) = a$ for all $a \in \Sigma_{\mathcal{A}}$, then we have a *Galois insertion.*

This framework can be generalized by allowing arbitrary concrete domain posets—the particular one we use here is the *power-set construction*—as well as weakening the Galois connection requirements, replacing them by constraints on a more general *description relation* $\rho \subseteq \mathcal{P}(\Sigma_{\mathcal{C}}) \times \Sigma_{\mathcal{A}}$ [Dams96]. Another approach is to express abstractions using an *abstraction mapping* $h : \Sigma_{\mathcal{C}} \to \Sigma_{\mathcal{A}}$, where the abstract model is a homomorphic image of the concrete one [Kur94, CGL94, RSS95, HS96]. Such homomorphisms preserve ∀CTL* properties, as we also do below. However, the Galois connection framework is sufficient for our purposes.

Remark 3.1.4 In general, we assume that we have different assertion languages for the abstract and concrete systems, which can be specialized to the respective domains of computation, and say that an assertion or temporal formula is *abstract* or *concrete* depending on the language that it belongs to. □

3.2 Abstract Property Satisfaction

We now discuss how to relate abstract and concrete temporal properties. One way to define the abstract satisfaction of a concrete property, given an abstract domain $\Sigma_{\mathcal{A}}$, is to define a special $||l||^{\mathcal{A}}$, which tells us which elements of the abstract domain satisfy literals of the *concrete* assertion language, as follows:

Definition 3.2.1 (Abstract pointwise satisfaction $||l||^{\mathcal{A}}$ [Dams96]) *For an abstract domain $\Sigma_{\mathcal{A}}$, concretization function $\gamma : \Sigma_{\mathcal{A}} \to \mathcal{P}(\Sigma_{\mathcal{C}})$, and concrete literal l, we define*

$$||l||^{\mathcal{A}} \stackrel{\text{def}}{=} \{a \mid \gamma(a) \subseteq ||l||\} \ .$$

Note that the $||l||^{\mathcal{A}}$ function ranges over concrete literals; the domain $\Sigma_{\mathcal{A}}$ includes a separate $||l^{\mathcal{A}}||$ function over abstract ones. (We regard the $||l||$ function as a fixed attribute of the domain, even though technically it is part of the Kripke structure—see Section 2.2.) While the $||l^{\mathcal{A}}||$ that corresponds to $\Sigma_{\mathcal{A}}$ satisfies $||l^{\mathcal{A}}|| \cup ||\neg l^{\mathcal{A}}|| = \Sigma_{\mathcal{A}}$, it is not the case that $||l||^{\mathcal{A}} \cup ||\neg l||^{\mathcal{A}} = \Sigma_{\mathcal{A}}$. This new $||l||^{\mathcal{A}}$ is used to directly define what it means for an abstract system to satisfy an arbitrary *concrete* temporal property:

Definition 3.2.2 ($\models^{\mathcal{A}}$ for $\forall$CTL*) *An abstract system $\mathcal{A}$ satisfies a* concrete *$\forall CTL^*$ property φ (over concrete assertions), written $\mathcal{A} \models^{\mathcal{A}} \varphi$, if $\mathcal{A}$ satisfies φ using the $\models$ of Definition 2.2.1, but replacing the $||p||$ native to $\mathcal{A}$ by the $||p||^{\mathcal{A}}$ of Definition 3.2.1.*

While the above definition is quite general, an alternative and more intuitive approach is to abstract and concretize temporal properties themselves. The following definition applies to universal properties, assumes negation-normal form, and is relative to a fixed abstraction domain (that is, a Galois connection as in Definition 3.1.2).

Definition 3.2.3 (Abstraction and concretization of CTL* properties) *For a concrete CTL^* temporal property φ, its* abstraction *$\alpha^t(\varphi)$ is obtained by replacing each assertion f in φ by an abstract assertion $\alpha^-(f)$ that characterizes the set of abstract states*

$$\alpha^-(f) : \bigvee\nolimits^{\mathcal{A}} \{a \in \Sigma_{\mathcal{A}} \mid \gamma(a) \subseteq f\} \ .$$

Conversely, given an abstract temporal property $\varphi^{\mathcal{A}}$, its concretization *$\gamma(\varphi^{\mathcal{A}})$ is obtained by replacing each atom a in $\varphi^{\mathcal{A}}$ by an assertion that characterizes $\gamma(a)$.*

The particular abstraction framework we use (Section 3.8) ensures that these assertions exist. Intuitively, this means that $\alpha^t(\varphi)$ should under-approximate assertions at the state-level as exactly as possible (the definition chooses the best under-approximation for each). On the other hand, concretization will be *exact*, since we will always have concrete assertions to characterize $\gamma(a)$ (if this were not possible, we should choose the smallest over-approximation instead).

Lemma 3.2.4 (Relating α^t and $\models^{\mathcal{A}}$) *Given a concrete property φ and abstract system $\mathcal{A}$, then $\mathcal{A} \models^{\mathcal{A}} \varphi$ (using Definition 3.2.2) if and only if $\mathcal{A} \models \alpha^t(\varphi)$ (using Definition 3.2.3).*

This gives rise to three equivalent definitions of weak preservation:

Definition 3.2.5 (Weak preservation) *$\mathcal{A}$ is a* weakly preserving abstraction *of $\mathcal{S}$ relative to a class of concrete temporal properties $\mathcal{P}$ if for any property $\varphi \in \mathcal{P}$,*

I. If $\mathcal{A} \models^{\mathcal{A}} \varphi$ then $\mathcal{S} \models \varphi$. Or, equivalently:

II. If $\mathcal{A} \models \alpha^t(\varphi)$ then $\mathcal{S} \models \varphi$. Or, equivalently:

III. For any abstract temporal property $\varphi^{\mathcal{A}}$ where $\gamma(\varphi^{\mathcal{A}}) \in \mathcal{P}$, if $\mathcal{A} \models \varphi^{\mathcal{A}}$ then $\mathcal{S} \models \gamma(\varphi^{\mathcal{A}})$.

3.3 Property-Preserving Abstractions

Naturally, we now would like to establish conditions on the concrete and abstract systems that guarantee weak preservation of different kinds of temporal properties. We first consider the simple case of universal properties. (Existential properties are discussed in Section 3.5.)

Definition 3.3.1 (Concretizing sequences $\gamma(\pi)$) *We say that a sequence $c_0, c_1, \ldots$ of $\mathcal{S}$-states* corresponds *to a sequence $a_0, a_1, \ldots$ of $\mathcal{A}$-states if $c_i \in \gamma(a_i)$ for all $i \geq 1$. For a sequence of abstract states $\pi : a_0, a_1, \ldots$, we define $\gamma(\pi)$ as the set of all sequences of concrete states that correspond to it. We extend this to a set of sequences S as*

$$\gamma(S) \stackrel{\text{def}}{=} \bigcup_{\pi \in S} \gamma(\pi) \ .$$

Then we can say:

Proposition 3.3.2 (LTL preservation) *System $\mathcal{A}$ is an LTL property-preserving abstraction of $\mathcal{S}$ if*

$$\mathcal{L}(\mathcal{S}) \subseteq \gamma(\mathcal{L}(\mathcal{A})) \ .$$

If $\mathcal{S}$ has no fairness constraints, then $\mathcal{L}(\mathcal{S})$ is equal to the set of runs of $\mathcal{S}$. Fairness constraints are discussed in Section 3.4. Different versions of the following theorem are presented and proved in [Dams96, LGS$^+$95, CGL94]:

Theorem 3.3.3 (Weakly preserving $\forall$CTL* abstraction) *Consider two systems $\mathcal{S} : \langle \Sigma_\mathcal{C}, \Theta_\mathcal{C}, R_\mathcal{C} \rangle$, and $\mathcal{A} : \langle \Sigma_\mathcal{A}, \Theta_\mathcal{A}, R_\mathcal{A} \rangle$ such that for a concretization function*

$$\gamma : \Sigma_\mathcal{A} \to \mathcal{P}(\Sigma_\mathcal{C})$$

the following hold:

1. INITIALITY*: $\Theta_\mathcal{C} \subseteq \gamma(\Theta_\mathcal{A})$.*
2. CONSECUTION*: If $R_\mathcal{C}(s_1, s_2)$ for some $s_1 \in \gamma(a_1)$ and $s_2 \in \gamma(a_2)$, then $R_\mathcal{A}(a_1, a_2)$.*

Then $\mathcal{A}$ is a weakly preserving abstraction of $\mathcal{S}$ for $\forall CTL^*$.

Note that the consecution requirement can be expressed using Definition 3.1.3 as $R \subseteq \gamma(R_\mathcal{A})$. Informally, the conditions ensure that $\mathcal{A}$ can do everything that $\mathcal{S}$ does, and perhaps some more.[1]

The proof [Dams96] proceeds by induction on the structure of the formula φ, and relies on the following lemma:

Lemma 3.3.4 *If $\pi : s_0, s_1, \ldots$ is a path in K, then there is a path $\pi_\mathcal{A} : a_0, a_1, \ldots$ in $K^\mathcal{A}$ such that $\pi \in \gamma(\pi_\mathcal{A})$.*

> **Proof:** We can choose any $a_i \in \alpha(s_i)$ for each i (see Definition 3.2.3). The consecution requirement ensures that, since $R(s_i, s_{i+1})$, then $R_\mathcal{A}(a_i, a_{i+1})$. $\square$

[1] It is also said that $\mathcal{S}$ *γ-simulates* $\mathcal{A}$ (see Section 3.6): the conditions of Theorem 3.3.3 ensure that $\rho \stackrel{\text{def}}{=} \{(c, a) \mid c \in \gamma(a)\}$ is a simulation from $\mathcal{S}$ to $\mathcal{A}$. As noted in [Dams96], this is confusing, and the terminology should probably be reversed in this context. We will simply say that *there is* a simulation between $\mathcal{A}$ and $\mathcal{S}$.

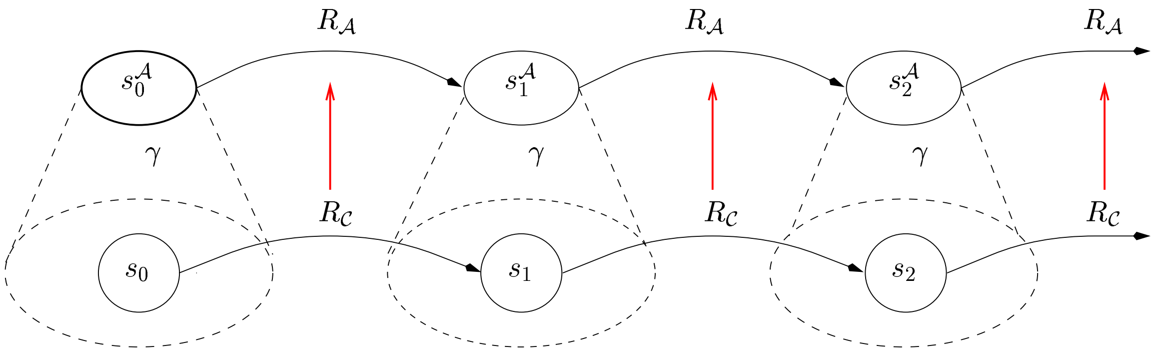

Figure 3.1: $\forall$CTL* weak preservation (γ-simulation)

This theorem applies to the weak preservation of LTL, being a subset of $\forall$CTL*. Figure 3.1 illustrates the proof for the case of LTL. The two conditions ensure that every run of $\mathcal{S}$ corresponds to some run of $\mathcal{A}$. Therefore, by Proposition 3.3.2, if every run of $\mathcal{A}$ satisfies an LTL temporal property $\varphi^{\mathcal{A}}$, every run of $\mathcal{S}$ will satisfy the corresponding property $\gamma(\varphi^{\mathcal{A}})$.

We would like $\mathcal{A}$ to satisfy as many properties as possible, which is the case (for all of CTL*) if $\Theta_{\mathcal{A}}$ is as small as possible. This is the case if $\Theta_{\mathcal{A}}$ is defined as $\{\alpha(c) \mid c \in \Theta_{\mathcal{C}}\}$. Similarly, the transition relation should be as small as possible—while still correct—in order to preserve the maximum number of universal properties. The best abstraction, then, is the one where $R_{\mathcal{A}}(a_1, a_2)$ *only* when there exist $s_1 \in \gamma(a_1)$ and $s_2 \in \gamma(a_2)$ such that $R_{\mathcal{C}}(s_1, s_2)$.

Example 3.3.5 (Abstract BAKERY) *Figure 3.2 shows an abstract version of the* BAKERY *program of Figure 2.1. Three bits, b_1, b_2 and b_3, are used to eliminate the infinite-domain concrete variables y_1 and y_2, while the finite-state control variables have been retained. The correspondence between these bits and the original system defines the abstract domain and the concretization function, in a way that we will formalize in Section 3.8.*

The transitions of this abstract system retain the fairness constraints of the concrete ones. This is justified in the following section.

$$\textbf{local} \quad b^{y_1=0}, b^{y_2=0}, b^{y_1\le y_2} \quad : \textbf{boolean where } b^{y_1=0}, b^{y_2=0}, b^{y_1\le y_2}$$

$$\left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{ll}\ell_0^{\mathcal{A}}: & \textbf{noncritical}\\ \ell_1^{\mathcal{A}}: & (b^{y_1=0}, b^{y_1\le y_2}) := (\mathit{false}, \mathit{false})\\ \ell_2^{\mathcal{A}}: & \textbf{await } (b^{y_2=0} \vee b^{y_1\le y_2})\\ \ell_3^{\mathcal{A}}: & \textbf{critical}\\ \ell_4^{\mathcal{A}}: & (b^{y_1=0}, b^{y_1\le y_2}) := (\mathit{true}, \mathit{true})\end{array}\right]\end{array}\right] \; || \; \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{ll}m_0^{\mathcal{A}}: & \textbf{noncritical}\\ m_1^{\mathcal{A}}: & (b^{y_2=0}, b^{y_1\le y_2}) := (\mathit{false}, \mathit{true})\\ m_2^{\mathcal{A}}: & \textbf{await } (b^{y_1=0} \vee \neg b^{y_1\le y_2})\\ m_3^{\mathcal{A}}: & \textbf{critical}\\ m_4^{\mathcal{A}}: & (b^{y_2=0}, b^{y_1\le y_2}) := (\mathit{true}, b^{y_1=0})\end{array}\right]\end{array}\right]$$

–P1– –P2–

$$\begin{array}{ll} b^{y_1=0}: & y_1 = 0\\ b^{y_2=0}: & y_2 = 0\\ b^{y_1\le y_2}: & y_1 \le y_2\end{array}$$

Figure 3.2: Finite-state abstraction of program BAKERY

3.4 Preservation under Fairness

If we consider fairness, we must ensure that unfair paths in $\mathcal{A}$ correspond only to unfair paths in $\mathcal{S}$. Otherwise, some computations (fair runs) of $\mathcal{S}$ might not be represented as computations of $\mathcal{A}$. Therefore, we must add an extra condition for fairness:

Theorem 3.4.1 *Consider systems $\mathcal{S} : \langle \mathcal{V}_\mathcal{C}, \Theta_\mathcal{C}, \mathcal{T}_\mathcal{C}\rangle$, and $\mathcal{A} : \langle \mathcal{V}_\mathcal{A}, \Theta_\mathcal{A}, \mathcal{T}_\mathcal{A}\rangle$ and concretization function $\gamma : \Sigma_\mathcal{A} \to \mathcal{P}(\Sigma_\mathcal{C})$. If the initiality and consecution conditions of Theorem 3.3.3 hold, and:*

3. FAIRNESS*: If an abstract transition $\tau^\mathcal{A}$ is marked as just (resp. compassionate), then there is a just (resp. compassionate) concrete transition $\tau^\mathcal{C}$ such that*

   (a) $\gamma(\mathit{enabled}(\tau^\mathcal{A})) \subseteq \mathit{enabled}(\tau^\mathcal{C})$
   (resp. $\gamma(\mathit{enabled}(\tau^\mathcal{A})) = \mathit{enabled}(\tau^\mathcal{C})$*), and*

   (b) $\mathit{post}(\tau^\mathcal{C}, \gamma(\mathit{enabled}(\tau^\mathcal{A}))) \subseteq \gamma(\mathit{post}(\tau^\mathcal{A}, \mathit{enabled}(\tau^\mathcal{A})))$.

Then $\mathcal{A}$ is a weakly preserving abstraction of $\mathcal{S}$ for $\forall CTL$.*

Note that (b) is guaranteed if $\tau^\mathcal{A}$ over-approximates $\tau^\mathcal{C}$, in which case (a) is always

$$\gamma(\mathit{enabled}(\tau^\mathcal{A})) = \mathit{enabled}(\tau^\mathcal{C}) \ .$$

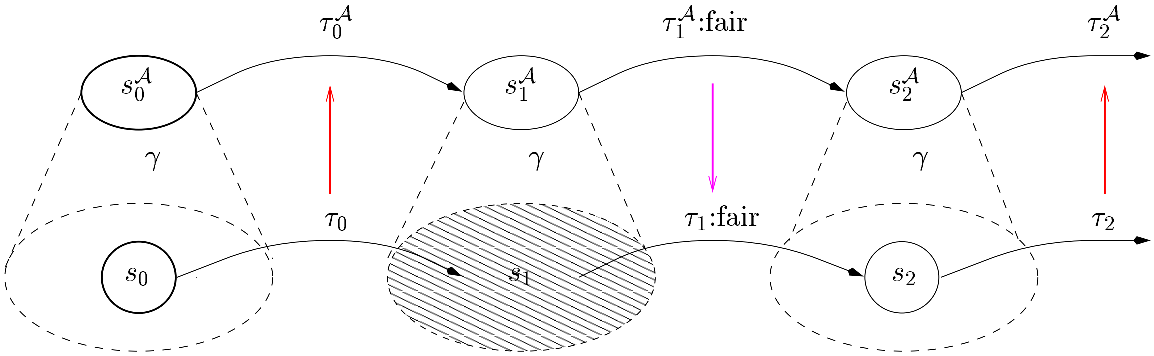

Figure 3.3: $\forall$CTL* weak preservation (γ-simulation) under fairness

Proof: The path quantification for $\forall$CTL* satisfaction now considers only *fair paths*. Thus, the proof proceeds as for Theorem 3.3.3, except that we now must also show that if a sequence $\pi_{\mathcal{A}} : a_0, a_1, \ldots$ is not fair for $\mathcal{A}$, then any sequence in $\gamma(\pi_{\mathcal{A}})$ is not fair for $\mathcal{S}$.

If $\pi_{\mathcal{A}}$ is not fair, then there is a fair abstract transition $\tau^{\mathcal{A}}$ that is enabled infinitely often (continuously in the case of justice) but never taken. Consider then the transition $\tau^{\mathcal{C}}$ guaranteed by the fairness clause of Theorem 3.4.1. By the $\subseteq$ requirement in condition (a), $\tau^{\mathcal{C}}$ is enabled in the respective states in $c_1, c_2, \ldots$ and, in the case of compassion, disabled at all other states, thanks to the equality requirement in (a). Condition (b) means that if $\tau^{\mathcal{A}}$ is not taken at an abstract state $s^{\mathcal{A}}$, then $\tau^{\mathcal{C}}$ cannot be taken at any state in $\gamma(s^{\mathcal{A}})$. Therefore $c_1, c_2, \ldots$ is not fair towards $\tau^{\mathcal{C}}$, since $\tau^{\mathcal{C}}$ can never be taken, a contradiction. $\Box$

Figure 3.3 illustrates this theorem, where $\tau_1^{\mathcal{A}}$ is an abstract fair transition; the corresponding concrete transition τ_1 should be enabled at all the states that correspond to those where $\tau_1^{\mathcal{A}}$ is enabled, and reach corresponding states.

Remark 3.4.2 If we weaken condition (a) to $\gamma(\mathit{enabled}(\tau^{\mathcal{A}})) \subseteq \mathit{enabled}(\tau^{\mathcal{C}})$ for the case of compassion, then Theorem 3.4.1 no longer holds: in the proof, it could be the case that the concrete transition $\tau^{\mathcal{C}}$ was taken at a state corresponding to one where $\tau^{\mathcal{A}}$ was *not* enabled.

Figure 3.4 shows this case. At abstract states A and B, compassionate transition $\tau^{\mathcal{A}}$ is enabled and not taken. Transition $\tau^{\mathcal{C}}$ cannot be taken at the corresponding concrete states either. However, $\tau^{\mathcal{A}}$ is not enabled at abstract state C, but could, under the weaker

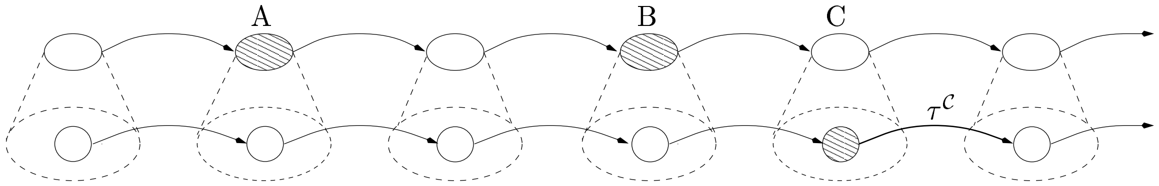

Figure 3.4: The problem with under-approximating the enabling condition of a compassionate transition

$\subseteq$ requirement, be enabled and taken at some concrete state that it represents. Thus, the abstract run may be uncompassionate towards $\tau^{\mathcal{A}}$ even though the concrete one is compassionate towards $\tau^{\mathcal{C}}$.

This is not a problem in the case of justice: if $\tau^{\mathcal{A}}$ is continuously enabled but not taken, then the weaker $\subseteq$ requirement ensures that $\tau^{\mathcal{C}}$ is also continuously enabled but not taken. $\square$

The conditions of Theorem 3.4.1 limit the fairness constraints that can be imposed on transitions in $\mathcal{A}$. Note that the more fairness constraints $\mathcal{A}$ has, the more CTL* properties it will satisfy. In Chapter 6, we will describe how partial information about the enabling condition of fair transitions can be used at the abstract level, which gives an alternative and more flexible approach.

3.4.1 Uniform Compassion

Uniform compassion is presented in [BLM97] to more accurately model compassion in parameterized distributed systems. This is a stronger version of compassion: if a transition τ is declared to be *uniformly compassionate*, then it cannot be enabled at infinitely many φ-states without being taken infinitely often *at a φ-state*, for any assertion φ. If uniform compassion enters the picture, we can add the following clause:

Theorem 3.4.3 (Preservation of Uniform Compassion) *$\tau^{\mathcal{A}}$ can be marked as uniformly compassionate if there is a uniformly compassionate concrete transition $\tau^{\mathcal{C}}$ such that*

(a) $\gamma(enabled(\tau^{\mathcal{A}})) \subseteq enabled(\tau^{\mathcal{C}})$, and

(b) $post(\tau^{\mathcal{C}}, \gamma(enabled(\tau^{\mathcal{A}}))) \subseteq \gamma(post(\tau^{\mathcal{A}}, enabled(\tau^{\mathcal{A}})))$.

	$\tau^{\mathcal{A}}$ just	$\tau^{\mathcal{A}}$ compassionate	$\tau^{\mathcal{A}}$ uniformly comp.
$\tau^{\mathcal{C}}$ unfair	—	—	—
$\tau^{\mathcal{C}}$ just	$\subseteq$	—	—
$\tau^{\mathcal{C}}$ compassionate	$\subseteq$	$=$	—
$\tau^{\mathcal{C}}$ uniformly comp.	$\subseteq$	$\subseteq$	$\subseteq$

Table 3.1: Justification of fairness of abstract transitions

Proof: Again we must show that if $\pi_{\mathcal{A}} : a_0, a_1, \ldots$ is not fair, then every computation in $\gamma(\pi_{\mathcal{A}})$ is unfair. Assume $\pi_{\mathcal{A}}$ is not fair towards the uniformly compassionate transition $\tau^{\mathcal{A}}$. Then $\tau^{\mathcal{A}}$ is infinitely often enabled but not taken at $\varphi^{\mathcal{A}}$-states, for some abstract assertion $\varphi^{\mathcal{A}}$. By condition (a), $\tau^{\mathcal{C}}$ is infinitely often enabled at all the corresponding $\gamma(\varphi^{\mathcal{A}})$-concrete states. By condition (b), $\tau^{\mathcal{C}}$ cannot be taken at those states. Consider now the other states in the concrete computation. These states *do not* satisfy $\gamma(\varphi)$, so (unlike the normal compassion case) whether $\tau^{\mathcal{C}}$ is taken at those states is irrelevant: $\tau^{\mathcal{C}}$ is enabled at infinitely many $\gamma(\varphi)$-states, but not taken at a $\gamma(\varphi)$-state, so any any computation in $\gamma(\pi)$ is unfair towards the uniformly compassionate $\tau^{\mathcal{C}}$. □

We can consider compassionate transitions to be just, and uniformly compassionate transitions to be both compassionate and just. Thus, for instance, a uniformly compassionate concrete transition can justify the justice of an abstract transition.

Table 3.1 summarizes the conditions under which $\tau^{\mathcal{A}}$ can be marked as fair based on the fairness of $\tau^{\mathcal{C}}$. A "—" entry indicates that the given fairness assignment to $\tau^{\mathcal{A}}$ cannot be justified by the fairness of $\tau^{\mathcal{C}}$. A "$\subseteq$" entry indicates that the weaker condition *3(a)* must hold, and "$=$" indicates that equality must hold.

Note that for the "$\subseteq$" entries, we can generate a *distinct* abstract fair transition $\tau^{\mathcal{A}}_{\varphi}$ for each set of states φ where we know that $\tau^{\mathcal{C}}$ is enabled.

Definition 3.4.4 (Transition-mapped abstraction) *Following [KMP94], we say that a* transition-mapped abstraction *is one where every abstract transition $\tau^{\mathcal{A}}$ over-approximates a concrete transition τ, that is, $\tau \subseteq \gamma(\tau^{\mathcal{A}})$. The mapping and the abstraction are* fairness-reducing *if the conditions for fairness preservation of Table 3.1 are observed. In this case, the $\subseteq$ requirements imply the $=$ one.*

Thus we have:

Proposition 3.4.5 *A fairness-reducing transition-mapped abstraction is weakly-preserving for $\forall CTL^*$.*

These are sufficient but not necessary conditions for the fairness of $\tau^{\mathcal{A}}$. Section 3.6 presents a more general formulation and, as mentioned before, Chapter 6 presents ways in which more flexible fairness constraints at the abstract level can be used.

The following Lemma is a special case of Theorem 3.4.1, expressed in terms of first-order assertions at the concrete level:

Lemma 3.4.6 *An abstract system $\mathcal{A} : \langle \mathcal{V}_{\mathcal{A}}, \Theta_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}} \rangle$ is a weakly preserving abstraction of $\mathcal{S} : \langle \mathcal{V}_{\mathcal{C}}, \Theta_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}} \rangle$ for $\forall CTL^*$ if:*

1. INITIALITY*: $\Theta \rightarrow \gamma(\Theta_{\mathcal{A}})$ is valid.*
2. CONSECUTION*: For every transition $\tau^{\mathcal{C}} \in \mathcal{T}_{\mathcal{C}}$ there is a set of transitions $\left\{\tau_1^{\mathcal{A}}, \ldots, \tau_k^{\mathcal{A}}\right\}$ in $\mathcal{T}_{\mathcal{A}}$ such that*
$$\tau^{\mathcal{C}} \rightarrow \gamma(\tau_1^{\mathcal{A}}) \vee \ldots \vee \gamma(\tau_k^{\mathcal{A}})$$
is valid.
3. FAIRNESS*: For every just (resp. compassionate) transition $\tau^{\mathcal{A}} \in \mathcal{T}_{\mathcal{A}}$ there is a just (resp. compassionate) transition $\tau \in \mathcal{T}$ such that the following are valid:*
   (a) $\gamma(enabled(\tau^{\mathcal{A}})) \rightarrow enabled(\tau)$
   (resp. $\gamma(enabled(\tau^{\mathcal{A}})) \leftrightarrow enabled(\tau)$) and
   (b) $post(\tau, \gamma(enabled(\tau^{\mathcal{A}}))) \rightarrow \gamma(post(\tau, enabled(\tau)))$.

Example 3.4.7 *In the abstract* BAKERY *program of Example 3.3.5 (Figure 3.2), we can label abstract transitions ℓ_2 and m_2 as just, since $\gamma(b^{y_2=0} \vee b^{y_1 \leq y_2})$ is equivalent to $y_2 = 0 \vee y_1 \leq y_2$ and $\gamma(b^{y_1=0} \vee \neg b^{y_1 \leq y_2})$ is equivalent to $y_1 = 0 \vee y_2 < y_1$ (c.f. Example 2.1.8, Figure 2.1).*

For this program, we can automatically model check the mutual exclusion, bounded overtaking and accessibility properties (Example 2.2.8), which are then guaranteed to hold of the original infinite-state BAKERY *program of Figure 2.1.*

3.5 Preserving Existential Properties

The previous lemmas apply to the preservation of *universal* properties ($\forall$CTL*) only. The logics CTL and CTL* include *existential* quantification as well, which expresses the possibility of certain computations.

A footnote in [CGL94] tells us that "[one of the authors] prefers systems that will do something useful to those that might." The paper by Lamport [Lam95], titled "Proving Possibility Properties," begins by stating: *"Proving possibility properties provides no useful information about a system."* However, it goes on to discuss how proving existential properties of *specifications* can be useful to validate and debug them (and shows how this can be done in an LTL framework, even though such properties cannot be directly expressed in LTL). A special case of this is the *non-Zenoness* of real-time and hybrid system specifications (see Section 2.1.4 and Example 2.2.4): real systems will never be Zeno, but faulty specifications might be.

It is also important to note that techniques for proving existential properties can also find *counterexamples* to universal ones. Combining the search for a proof with the search for a counterexample can help overcome one of the main drawbacks of traditional deductive verification (see Section 1.1).

Defining abstractions that preserve both universal and existential properties is more complex. This is done in [CIY95, DGG97, Dams96] by defining Kripke structures with two different transition relations, each one used to interpret existential and universal path quantification respectively.

For reasons that will soon become clear, we write $R_{\mathcal{A}}^{\forall\exists}$ to indicate the extra abstract transition relation. [Dams96] calls this the *constrained* transition relation, while the universal-preserving relation $R_{\mathcal{A}}$ we have been using so far is the *free* one. In [CIY95], they are called the *conservative* and *liberal* transition relations, respectively (the expanded Kripke structure is a *democratic* one). To preserve properties that combine existential and universal properties, *both* transition relations are used to give semantics to CTL* formulas, depending on the temporal modality being analyzed. We briefly formalize this as follows:

Definition 3.5.1 ($\forall\exists$-subgraph, $\text{paths}^{\forall\exists}(K, s)$**)** *Given a Kripke structure K and a* constrained transition relation $R_{\mathcal{A}}^{\forall\exists}$, *the* $\forall\exists$-subgraph *of K is the subgraph whose edges are exactly those where $R_{\mathcal{A}}^{\forall\exists}$ holds. For a node s in K, we write* $\text{paths}^{\forall\exists}(K, s)$ *for the set of all infinite paths in K starting at s in the $\forall\exists$-subgraph.*

We can now formally define what it means for an abstract system to satisfy a concrete CTL* temporal property:

Definition 3.5.2 ($\models^{\mathcal{A}}$ for CTL*) *An abstract system $\mathcal{A}$ with extra constrained transition relation $R_{\mathcal{A}}^{\forall\exists}$ satisfies a* concrete *CTL* property φ (over concrete assertions), written $\mathcal{A} \models^{\mathcal{A}} \varphi$, if $\mathcal{A}$ satisfies φ using the $\models$ of Definition 2.2.1, where (1) we replace the $||p||$ native to $\mathcal{A}$ by the $||p||^{\mathcal{A}}$ of Definition 3.2.1, and (2) we modify the clause for existential path quantification as follows:*

$$(K, s) \models \mathsf{E}\varphi \quad \text{if} \quad \pi \models \varphi \textit{ for some } \pi \in \text{paths}^{\forall\exists}(K, s)$$

See [Dams96] for details. The following theorem gives the $R_{\mathcal{A}}^{\forall\exists}$ relation its name:

Theorem 3.5.3 (CTL* preservation) *Given a concrete system $\mathcal{S} : \langle \Sigma_{\mathcal{C}}, \Theta_{\mathcal{C}}, R_{\mathcal{C}} \rangle$, and abstract system*

$$\mathcal{A} : \left\langle \Sigma_{\mathcal{A}}, \Theta_{\mathcal{A}}, R_{\mathcal{A}}, R_{\mathcal{A}}^{\forall\exists} \right\rangle \ .$$

Assume that the conditions of Theorem 3.3.3 hold, and

(iii) If $R_{\mathcal{A}}^{\forall\exists}(a_1, a_2)$ then for all $s_1 \in \gamma(a_1)$ there exists $s_2 \in \gamma(a_2)$ *such that $R_{\mathcal{C}}(s_1, s_2)$.*

Then all properties of CTL are weakly preserved, that is, if $\mathcal{A} \models^{\mathcal{A}} \varphi$, then $\mathcal{S} \models \varphi$.*

To maximize the number of existential properties we can prove over the abstract system, we should maximize the $R_{\mathcal{A}}^{\forall\exists}$ relation. This is achieved if $R_{\mathcal{A}}^{\forall\exists}(a_1, a_2)$ holds *exactly when* for all $s_1 \in \gamma(a_1)$ there exists $s_2 \in \gamma(a_2)$ such that $R_{\mathcal{C}}(s_1, s_2)$. Note that under-approximating each concrete transition $\tau^{\mathcal{C}}$ to a corresponding abstract $\tau_{\forall\exists}^{\mathcal{A}}$ such that

$$\gamma(\tau_{\forall\exists}^{\mathcal{A}}) \subseteq \tau^{\mathcal{C}}$$

is a sufficient but not necessary condition for Theorem 3.5.3 above; that is, it does not always give the best possible existential-preserving abstraction over the assertion-based abstraction domain. We return to this in Section 4.9.

The abstract initial condition should still be an over-approximation of the concrete one; to maximize the number of properties that hold, we should make it as small as possible (while still a correct over-approximation): the fewer computation trees the system has, the more CTL* properties will hold for all of them.

Fairness: We now briefly address the question of how a CTL* model checker for the abstract system should handle fairness. Satisfaction of CTL* should only consider *fair paths* through the Kripke structure when existentially or universally quantifying over them. For the concrete system, a path is fair if it is a computation of the system.

In the case of an abstract system, we again need sound fairness constraints to identify unfair paths. If $\pi_{\mathcal{A}}$ is deemed unfair, then all computations of $\gamma(\pi_{\mathcal{A}})$ should be unfair too. We now note that if the existential version of τ is taken, the universal version can be considered taken too. Thus, our criteria for eliminating unfair paths can be the same as before, using fairness constraints on the standard $\exists\exists$ free transition relation.

On the other hand, to ensure that paths that are witnesses for existential quantification are indeed fair, all transitions that are just (resp. compassionate) at the concrete level should be disabled infinitely often (resp. continuously) with respect to the $\exists\exists$ abstract relation, or else taken infinitely often, with respect to the $\forall\exists$ relation. We will return to this in Chapter 6. Expressed in terms of first-order transition relations, we have:

Proposition 3.5.4 *If the conditions of Lemma 3.4.6 hold for the $\mathcal{T}_{\mathcal{A}}$ relations, and for all abstract constrained transition relations $\tau_{\forall\exists}^{\mathcal{A}}$ and all abstract states a*

$$\gamma(a) \rightarrow \exists\mathcal{V}'.\left(\mathcal{T}_{\mathcal{C}}(\mathcal{V},\mathcal{V}') \wedge \gamma(post(a,\tau_{\forall\exists}^{\mathcal{A}}))\right)$$

is valid, then $\mathcal{A} : \left\langle \mathcal{V}_{\mathcal{A}}, \Theta_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}}, \left\{\tau_{\forall\exists}^{\mathcal{A}}\right\}\right\rangle$ is a CTL weakly-preserving abstraction of $\mathcal{S}$.*

Recall that first-order formulas such as the above are implicitly universally quantified throughout this thesis.

3.6 A General Simulation Rule

A general proof rule for simulation between systems is presented in [KMP94]. This rule is rephrased in Figure 3.5, to include abstraction. The temporal formula $taken(\tau)$ characterizes, not surprisingly, the states in a computation where τ is taken (see [MP91b], p. 255). Expressing *taken* at the concrete (infinite-state) level requires, in general, *rigid quantification* over auxiliary variables, or a *next value* operator. This is less inconvenient at the abstract finite-state level, as we will see in Chapter 6.

For systems $\mathcal{S} : \langle \mathcal{V}_\mathcal{C}, \mathcal{T}_\mathcal{C}, \Theta_\mathcal{C} \rangle$ and $\mathcal{A} : \langle \mathcal{V}_\mathcal{A}, \mathcal{T}_\mathcal{A}, \Theta_\mathcal{A} \rangle$,

S1. INITIALITY: $\Theta_\mathcal{C} \subseteq \gamma(\Theta_\mathcal{A})$

S2. CONSECUTION: $\mathcal{S} \models \tau^\mathcal{C} \rightarrow \bigvee_{\tau^\mathcal{A} \in \mathcal{T}_\mathcal{A}} \gamma(\tau^\mathcal{A}) \quad$ for every $\tau \in \mathcal{T}$

S3. JUSTICE: $\mathcal{S} \models \gamma(enabled(\tau^\mathcal{A})) \Rightarrow \Diamond \left(\neg\gamma(enabled(\tau^\mathcal{A})) \vee \gamma(taken(\tau^\mathcal{A})) \right)$
for every just $\tau^\mathcal{A} \in \mathcal{T}_\mathcal{A}$

S4. COMPASSION: $\mathcal{S} \models \Box \Diamond \gamma(enabled(\tau^\mathcal{A})) \Rightarrow \Diamond \gamma(taken(\tau^\mathcal{A}))$
for every compassionate $\tau^\mathcal{A} \in \mathcal{T}_\mathcal{A}$

$\mathcal{L}(\mathcal{S}) \subseteq \gamma(\mathcal{L}(\mathcal{A}))$

Figure 3.5: General simulation rule

Theorem 3.6.1 (General LTL Preservation Rule)
The general simulation rule of Figure 3.5 is sound.

Proof: (Outline) Premises S1 and S2 are the usual initiality and consecution conditions. Premise S3 states that if an abstract just transition is enabled, then it eventually must be taken or disabled—when translated back to the computations of the original system. Similarly, premise S4 requires that compassionate abstract transitions be taken if they are infinitely often enabled. $\Box$

3.7 Unobservable Abstract Variables

Refinement between systems can be understood as the inverse of the abstraction process. Here, a high-level (abstract) specification is shown to correspond to a low-level (concrete) implementation, which is then guaranteed to satisfy the properties of the specification.

Refinement complicates the abstraction framework by allowing abstract variables that do not correspond to any concrete variables. We do not address such refinement in this thesis, but in Chapter 5 we will consider the case where abstract system variables are not reflected in the concrete states. Such variables are deemed *unobservable* in the concrete system.

The crux of the following definitions is that an abstract state represents exactly the same set of states it would represent if the variables in U were not part of $\mathcal{A}$.

Definition 3.7.1 (U-variants) *A state s is a U-*variant *of a state s' if s and s' coincide in the values of variables* not *in U. For a sequence of states $\pi' : s_0, s_1, \ldots$, a sequence $\pi' : s_0, s_1, \ldots$ is a U-*variant *of π if each s_i is a U-variant of s'_i.*

For a state or sequence X, the set of U-variants of x is written $X[U]$. For a set of states S, we write $x[U]$ for the set $\{s' | s' \in s[U]$ and $s \in S\}$.

Definition 3.7.2 (Unobservable abstract variables) *Abstract system variables can be marked as* unobservable*. Let U be the set of unobservable variables of $\mathcal{A}$. Then for a state, set of states, sequence or set of sequences X, the new concretization function γ_{New} is defined as:*

$$\gamma_{New}(X) \stackrel{\text{def}}{=} \gamma(X[U]) \ .$$

Thus, an abstract state represents the same set of states that all of its U-variants do, and an abstract computation corresponds to the same set of computations that all of its U-variants do.

An alternative is to consider the unobservable variables U as concrete variables as well, which have the same value in corresponding abstract and concrete states. Then we would have $\gamma(X[U]) = (\gamma(X))[U]$.

Proposition 3.7.3 *If $\mathcal{A}$ has unobservable variables, all the theorems in this chapter still hold when γ is replaced by γ_{New}.*

3.8 Assertion-based Abstraction

We will use a particularly simple instance of the previous abstraction framework.

Definition 3.8.1 (Boolean algebra) *A* complete boolean algebra *$\mathcal{BA}(S)$, given a set S, is a structure $\left\langle S, \wedge^{\mathcal{A}}, \vee^{\mathcal{A}}, \neg, \mathit{false}^{\mathcal{A}}, \mathit{true}^{\mathcal{A}} \right\rangle$ which satisfies the usual properties of Boolean logic, that is, $\left\langle S, \wedge^{\mathcal{A}}, \vee^{\mathcal{A}} \right\rangle$ a distributive lattice, with top element $\mathit{true}^{\mathcal{A}}$, bottom element $\mathit{false}^{\mathcal{A}}$, and complement $\neg$, where $a \wedge^{\mathcal{A}} \neg a = \mathit{false}^{\mathcal{A}}$ and $a \vee^{\mathcal{A}} \neg a = \mathit{true}^{\mathcal{A}}$ (see [DP90]).*

As we will see, the following abstract domain is often implicitly used in deductive verification:

Definition 3.8.2 (Assertion-based abstraction) *Given a finite set of assertions B, the* assertion-based abstract domain *with* basis B *has the complete boolean algebra $\mathcal{BA}(B)$ (using $\wedge^{\mathcal{A}}, \vee^{\mathcal{A}}, \neg^{\mathcal{A}}$) as its abstract domain $\Sigma_{\mathcal{A}}$, where*

— $s_1^{\mathcal{A}} \preceq s_2^{\mathcal{A}}$ *iff* $s_1^{\mathcal{A}}$ *implies* $s_2^{\mathcal{A}}$,

— $\gamma(f) \stackrel{\text{def}}{=} \{s \in \Sigma \mid s \vDash f\}$, *and*

— $\alpha(S) \stackrel{\text{def}}{=} \bigwedge^{\mathcal{A}} \left\{s^{\mathcal{A}} \in \mathcal{BA}(B) \mid S \subseteq \gamma(s^{\mathcal{A}})\right\}$.

That is, the concretization $\gamma(f)$ of an assertion f is simply the set of concrete states that satisfies it. The abstraction $\alpha(S)$ of a set of states S is the smallest point in the abstract lattice whose concretization includes all the elements of S. Note that γ is continuous, since

$$\gamma(f_1 \wedge^{\mathcal{A}} f_2) = \gamma(f_1) \cap \gamma(f_2) \ .$$

This is a *Galois insertion* from $(\mathcal{P}(\Sigma), \subseteq)$ to $(\Sigma_{\mathcal{A}}, \preceq)$, since $\alpha(\gamma(f)) = f$ for all $f \in \mathcal{BA}(B)$. The converse, $\gamma(\alpha(f)) \supseteq f$, is not always the equality, since there will be sets of concrete states for which there is no exact corresponding abstract state; in this case, $\gamma(\alpha(f))$ will be the intersection of the states represented by all the abstract points a such that $\gamma(a) \supseteq f$.

Our results for assertion-based abstraction will hold of any abstract lattice for which the boolean operations are available.

Note 3.8.3 (Abstract vs. concrete assertions) *We will continue to characterize sets of concrete states using assertions, which need not be points in the abstract state-space. To distinguish abstract points and assertions from concrete ones, we use $\wedge^{\mathcal{A}}, \vee^{\mathcal{A}}, \neg^{\mathcal{A}}, \rightarrow^{\mathcal{A}}$ for operations in the abstract domain, while $\wedge, \vee, \neg, \rightarrow$ are the usual connectives in the concrete assertion language (occasionally, the superscripts will be dropped when the context is clear). Note that the abstract domain serves as its own (propositional) assertion language.*

Proposition 3.8.4 (γ as a boolean homomorphism) *If $s^{\mathcal{A}}$ is an abstract assertion (or, equivalently, an abstract state), then $\gamma(s^{\mathcal{A}})$ is characterized by the concrete assertion obtained from $s^{\mathcal{A}}$ by replacing $\wedge^{\mathcal{A}}$, $\vee^{\mathcal{A}}$ and $\neg^{\mathcal{A}}$ by $\wedge$, $\vee$ and $\neg$. The boolean variables in $s^{\mathcal{A}}$, which are elements of B, appear as corresponding subformulas in $\gamma(s^{\mathcal{A}})$. That is, γ is a* boolean homomorphism *between the two assertion languages.*

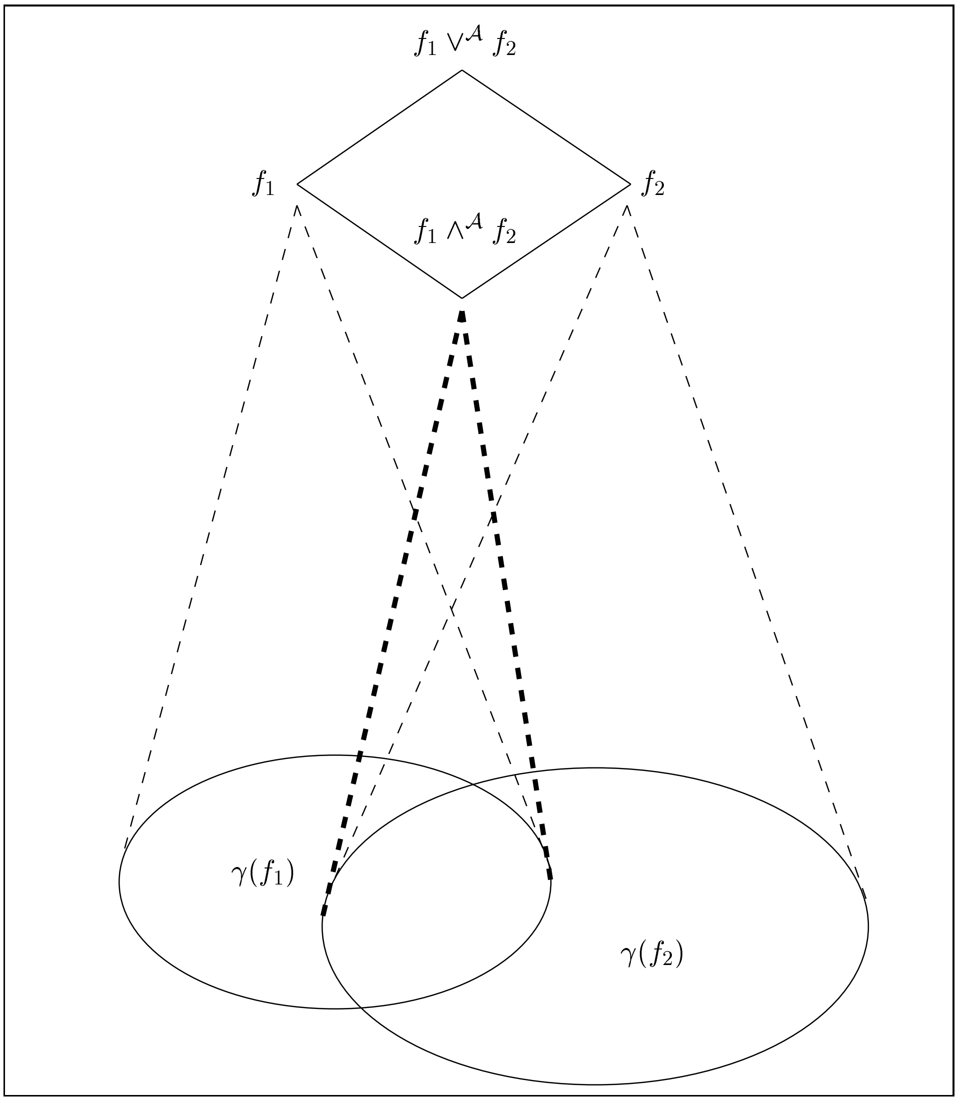

Figure 3.6: Assertion-Based Abstraction; $\gamma(f_1 \wedge^{\mathcal{A}} f_2) = \gamma(f_1) \cap \gamma(f_2)$

For a formula $s^{\mathcal{A}} \in \mathcal{BA}(B)$, we will write $\gamma(s^{\mathcal{A}})$ to characterize the set of states it represents, rather than simply $s^{\mathcal{A}}$, to highlight the fact that $s^{\mathcal{A}}$ is an abstract state, while $\gamma(s^{\mathcal{A}})$ is a concrete assertion representing a set of concrete states (see Figure 3.6). In this way, the extension of γ to sets can be defined as the assertion

$$\gamma(S) \stackrel{\text{def}}{=} \bigvee_{a \in S} \gamma(a) \ .$$

Another reason for insisting on the $\gamma(s^{\mathcal{A}})$ notation is that the abstract assertion language can be replaced by a less familiar one.

Example 3.8.5 (Abstract Bakery as an ABA) *The abstract Bakery program of Figure 3.2 is an assertion-based abstraction, with basis $B : \{b^{y_1=0}, b^{y_2=0}, b^{y_1 \le y_2}\}$. Any set of abstract states can be described as an assertion $f^{\mathcal{A}}$ over $b^{y_1=0}$, $b^{y_2=0}$, $b^{y_1 \le y_2}$ and the finite-state control variables. If we let $\gamma(b^{y_1=0}) = \{s \mid s \vDash y_1 = 0\}$, $\gamma(b^{y_2=0}) = \{s \mid s \vDash y_2 = 0\}$, and $\gamma(b^{y_1 \le y_2}) = \{s \mid s \vDash y_1 \le y_2\}$, then $\gamma(f^{\mathcal{A}})$ is characterized by the corresponding concrete assertion (see Proposition 3.8.4 above).*

The preservation of a finite-state concrete variable v can be easily but tediously formalized by introducing a set of abstract variables to encode the possible values of v.

3.9 Related Work

Most abstraction-based property-preservation frameworks do not account for fairness. However, Pnueli and Kesten [PK98, KP98] have recently described an assertion-based ∀CTL*-preserving abstraction framework that includes fairness considerations, expressed in terms of *fair Kripke structures*.

This formalism adds fairness directly to Kripke structures: the Kripke structure now include *justice requirements*, which are sets of states J_i such that a computation should have infinitely many J_i-states, and *compassion requirements*, which are pairs $\langle P_i, Q_i\rangle$ such that infinitely many P_i states then there are infinitely many Q_i states. This representation brings the Kripke structure closer to an ω-automata, and replaces the need for individual transition relations.

Over-approximating the J_i's and Q_i's, and under-approximating the P_i's, (weakly) preserves ∀CTL* properties, similarly to Theorem 3.4.1. Progress properties can be model checked for a finite-state abstract Kripke structure and transferred to the original, possibly infinite-state one, as we will also do.

In Chapter 6, we will release abstract transitions from the requirement of having to be marked as just or compassionate, and will use extra information about the concrete transitions instead, expressed in terms of the abstract states. Informally, this representation can be thought as lying in between (abstract) Fair Kripke Structures and (abstract) Fair Transition Systems.

Summary: Simulation and refinement rules such as those described in this chapter can be used provided that the abstract system $\mathcal{A}$ is given beforehand, together with an abstraction mapping between the two systems. However, they do not address the question of how abstract systems can be found or generated. This is the subject of the rest of this thesis.

Chapter 4

Automatic Generation of Abstract Systems

In the terminology of Chapter 3, one way to establish properties for an infinite-state system $\mathcal{S}$ is to find a finite-state abstraction $\mathcal{S}^{\mathcal{A}}$ that preserves these properties. $\mathcal{S}^{\mathcal{A}}$ can then be model checked using automatic, algorithmic tools.

The theorems of Chapter 3 describe *when* an abstraction is property-preserving, and can be used to check that a given abstraction is correct, but do not address the issue of *constructing* such abstractions. One approach is to construct the abstraction manually, and separately prove the conditions of Theorem 3.3.3 or Theorem 3.4.1. This offers the most flexibility, but is time-consuming and is subject to errors if not done entirely within a computer-aided verification environment.

Another approach is to construct the abstract system based on the concrete one and a description of the abstraction function. This is the approach we present in this chapter, automatically generating assertion-based abstractions from the concrete fair transition system and the basis of the desired abstraction. The algorithm we present has the following features:

- Uses a validity checker for the assertion language as a black box.
- Generates abstract assertions from concrete assertions, and is compositional in the structure of the formulas that describe the concrete initial condition and transition relations. This allows us to generate an abstract FTS from a concrete FTS, and then let a model checker explore the state-space of the abstraction.

- Is thus relatively efficient: its performance is polynomial in the size of the transition relations and not the size of the abstract state-space.

- Can be applied to any assertion language and abstract domain for which the corresponding boolean algebra and validity checker exist.

The (relative) efficiency of the algorithm makes it easier to try out different alternatives. This is particularly useful when systems are being debugged, and when different checks require different abstractions. The generated abstractions also lend themselves to refinement, can be incrementally constructed, and can be easily combined, as we will see in Chapter 6.

Note 4.0.1 (Assertion Basis) *In the following, let $B : \{b_1, \ldots, b_n\}$ be a fixed assertion basis.*

The basis B defines an assertion-based abstract domain, as given by Definition 3.8.2. As mentioned in Section 3.8, the concretization function $\gamma(f)$ is trivial to compute: for any abstract state s, the corresponding set of concrete states is characterized by the formula obtained by replacing each bit in s by its corresponding assertion. The same applies to relations, replacing primed bits by primed assertions. However, computing α is more problematic: for a set of concrete states S, the best abstraction of S is the greatest lower bound (that is, the conjunction) of all the abstract states whose concretization includes S. Finding all such abstract points is impractical, even for a singleton set S, and much more so when S is infinite and itself characterized by an assertion.

Thus, we will be content with *approximating* α. Instead of replacing concrete sets of states and relations by their best possible abstraction, which may be too expensive or impossible to compute, we will replace them by a conservative approximation that will still preserve the class of properties we are interested in. (That is, the definition of assertion-based abstraction tells us what the best possible abstraction is, but in general we will find a sub-optimal one that, hopefully, will still be good enough to prove the property of interest.)

Note 4.0.2 (On α and γ) *In the rest of this chapter, α will refer to a function used to* approximate *the abstraction function of the same name in Chapter 3, while γ is the exact concretization function.*

We will use validity checking to relate abstract and concrete (sets of) states. The abstraction procedure assumes the availability of a *validity checker*, which will be used as a black box:

Definition 4.0.3 (Validity checker) *We assume we have a procedure, checkValid, which is a sound, but not necessarily complete, validity checker for the assertion language (see Section 2.6.2): if checkValid(p) succeeds, then p is valid.*

Validity checkers are provided by theorem provers such as PVS, verification systems such as STeP, and specialized tools such as SVC (see Section 2.6).

The abstraction algorithm is based on a procedure that approximates assertions over $\mathcal{V}$ and $\mathcal{V}'$ as assertions over B and B'. The procedure descends through the boolean structure of the formula, building an assertion to serve as a *context* and keeping track of the polarity of subexpressions until it reaches the atoms. The procedure then over- or under-approximates each atom using an element of $\mathcal{BA}(B \cup B')$, where the context is used to improve the abstraction.[1]

We show how to abstract single atoms in Section 4.1. This procedure is then used in Section 4.2 to abstract assertions, which are boolean combinations of atoms. Abstract transition systems and temporal properties can then be easily generated, as shown in Sections 4.3 and 4.4.

4.1 Abstracting Atoms

Atoms are abstracted by testing them, given a context C, against a set of *points*

$$P \subseteq \mathcal{BA}(B \cup B') \ .$$

Default sets of points for pure and mixed atoms are defined below; these sets of points can be redefined or expanded by the user.

$$\begin{array}{lcll}
\alpha_{atom}(+, C, a) & \stackrel{\text{def}}{=} & \bigwedge^{\mathcal{A}} \{p \in P \mid \mathit{checkValid}((C \wedge a) \rightarrow \gamma(p))\} & \text{(over-approximation)} \\
\alpha_{atom}(-, C, a) & \stackrel{\text{def}}{=} & \bigvee^{\mathcal{A}} \{p \in P \mid \mathit{checkValid}((C \wedge \gamma(p)) \rightarrow a)\} & \text{(under-approximation)}
\end{array}$$

[1] While it is possible to push all negations down to the level of atoms and avoid polarity considerations, we will both under- and over-approximate formulas (at the topmost level) later on, so the above presentation is more convenient to use.

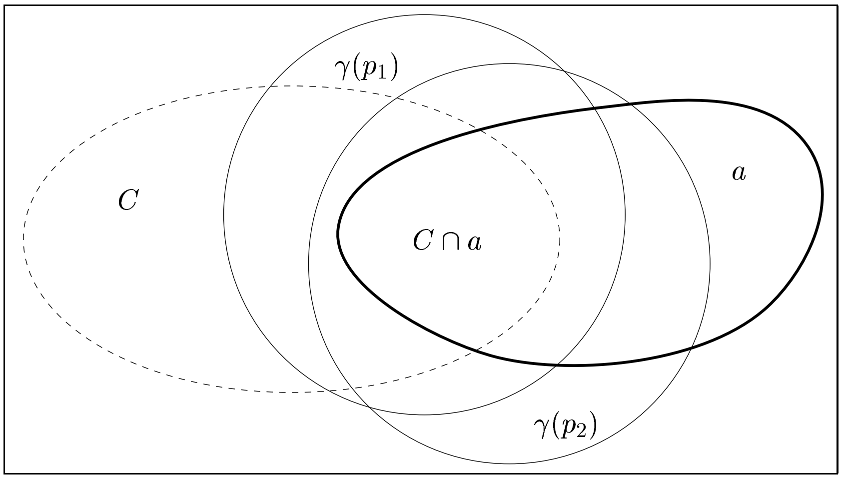

Figure 4.1: Over-approximations (relative to C) are conjoined

Intuitively, the context C indicates that we only need to correctly approximate the part of a that lies within C. Thus, when over-approximating a in context C, we can over-approximate $a \wedge C$ instead, a smaller set than a. This yields a smaller result, and hence a more precise over-approximation. Similarly, when under-approximating a in context C, we can under-approximate $a \vee \neg C$ instead, which is a larger set than simply a. This will give a larger result, and hence a better overall under-approximation.

Lemma 4.1.1 (Soundness of Approximations) *For any atom a,*

$$C \rightarrow (a \rightarrow \gamma(\alpha(+, C, a))) \quad \textit{and} \quad C \rightarrow (\gamma(\alpha(-, C, a)) \rightarrow a)$$

are valid.

Proof: In set-theoretic notation, this can be expressed as:

$$C \cap a \ \subseteq \ \gamma(\alpha(+, C, a))$$

and

$$C \cap \gamma(\alpha(-, C, a)) \ \subseteq \ a \ .$$

Let T be the set of points for which *checkValid* returns *true*.

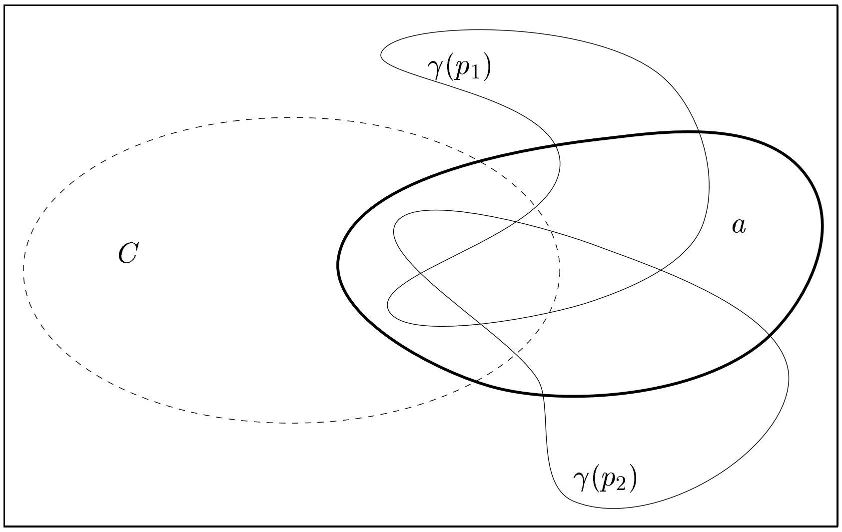

Figure 4.2: Under-approximations (relative to C) are disjoined

$+$ case: For every point p in T, $(C \cap a) \subseteq \gamma(p)$. Thus, $C \cap a \subseteq \bigcap_{p \in T} \gamma(p)$. This is illustrated in Figure 4.1. Since γ is continuous,

$$\bigcap_{p \in T} \gamma(p) = \gamma(\bigwedge\nolimits^{\mathcal{A}}_{p \in T} p)$$

and thus

$$C \cap a \ \subseteq\ \gamma(\bigwedge\nolimits^{\mathcal{A}}_{p \in T} p)\,,$$

which we wanted to show.

$-$ case: For every point p for which *checkValid* returns *true*, $(C \cap \gamma(p)) \subseteq a$. Thus, $C \cap \bigcup_{p \in P} \gamma(p) \subseteq \gamma(a)$. This is illustrated in Figure 4.2. Similarly to the previous case, this means that

$$\bigcup_{p \in P} \gamma(p) = \gamma(\bigcup_{p \in P} (p))$$

and thus

$$C \cap \gamma(\bigvee\nolimits^{\mathcal{A}}_{p \in P} (p)) \ \subseteq\ \gamma(a) \ .$$

$\square$

4.2 Abstracting Assertions

Note 4.2.1 (Polarity) *For* $\pi \in \{+,-\}$, *we define* π^{-1} *as* $+^{-1} \stackrel{\text{def}}{=} -$ *and* $-^{-1} \stackrel{\text{def}}{=} +$.

We extend α_{atom} to a function α that abstracts assertions as follows:

$$\begin{array}{rcl}
\alpha(\pi, C, a) & = & \alpha_{atom}(\pi, C, a), \text{ if } a \text{ is an atom} \\
\alpha(\pi, C, \neg q) & = & \neg^{\mathcal{A}} \alpha(\pi^{-1}, C, q) \\
\alpha(+, C, q \wedge r) & = & \textbf{let } \hat{q} = \alpha(+, C, q) \textbf{ in } \hat{q} \wedge^{\mathcal{A}} \alpha(+, C \wedge \gamma(\hat{q}), r) \\
\alpha(+, C, q \vee r) & = & \textbf{let } \hat{q} = \alpha(+, C, q) \textbf{ in } \hat{q} \vee^{\mathcal{A}} \alpha(+, C \wedge \neg\gamma(\hat{q}), r) \\
\alpha(-, C, q \wedge r) & = & \textbf{let } \hat{q} = \alpha(-, C, q) \textbf{ in } \hat{q} \wedge^{\mathcal{A}} \alpha(-, C \wedge \gamma(\hat{q}), r) \\
\alpha(-, C, q \vee r) & = & \textbf{let } \hat{q} = \alpha(-, C, q) \textbf{ in } \hat{q} \vee^{\mathcal{A}} \alpha(-, C \wedge \neg\gamma(\hat{q}), r) \ .
\end{array}$$

An assertion f is thus abstracted using $\mathrm{O}(|P| \cdot |f|)$ validity checks.

The intuition is as follows: when over-approximating $q \wedge r$, if we first over-approximate q to $\hat{q}$, then we know that a correct result can be forced to lie within $\hat{q}$. Thus, we add this to the context when over-approximating r. Similarly, when over-approximating $q \vee r$, if we first over-approximate q to $\hat{q}$, we then can commit to a result that includes $\hat{q}$; thus, when over-approximating r, we can consider only the part of r that lies outside $\hat{q}$, that is, add $\neg\hat{q}$ to the context. Similar reasoning lies behind the $-$ case.

The main claim that justifies the correctness of the algorithm is:

Theorem 4.2.2 *For assertions* C *and* f,

$$C \to (\gamma(\alpha(-, C, f)) \to f) \quad \textit{and} \quad C \to (f \to \gamma(\alpha(+, C, f)))$$

are valid, that is,

$$C \cap \gamma(\alpha(-, C, f)) \subseteq f \quad \textit{and} \quad C \cap f \subseteq \gamma(\alpha(+, C, f)) \ .$$

Proof: We prove this by induction on the structure of the formula f.[2] The base case, where the formula is an atom, is covered by Lemma 4.1.1. Assume now that the

[2] Researchers in formal verification are sometimes criticized for not verifying their own algorithms and verification systems, and indeed there is a small typo in [CU98]. We therefore include this proof in rather excruciating detail.

theorem holds for formulas q and r.

• $(\neg q)$: Complementing a correct over- (resp. under-) approximation for $\neg q$ produces a correct under- (resp. over-) approximation of q. More formally: by the induction hypothesis, $C \rightarrow (\gamma(\alpha(-, C, q)) \rightarrow q)$ is valid. Then

$$C \rightarrow (\neg q \rightarrow \neg\gamma(\alpha(-, C, q)))$$

is valid. By the definition of α, we have $\alpha(+, C, \neg q) = \neg^{\mathcal{A}}\alpha(-, C, q)$, so $\alpha(-, C, q) = \neg^{\mathcal{A}}\alpha(+, C, \neg q)$; thus

$$C \rightarrow (\neg q \rightarrow \neg\gamma(\neg^{\mathcal{A}}\alpha(+, C, \neg q)))$$

is valid. Since $\neg\gamma(f) = \gamma(\neg f)$, this means that

$$C \rightarrow (\neg q \rightarrow \gamma(\alpha(+, C, \neg q)))$$

is valid, which we wanted to show. The proof for $\alpha(-, C, \neg q)$ is dual.

• $\alpha(+, C, q \wedge r)$: As in the algorithm, let $\hat{q} \stackrel{\text{def}}{=} \alpha(+, C, q)$. By induction hypothesis,

$$(C \cap q) \subseteq \gamma(\hat{q}) \quad \text{and} \quad (C \wedge \gamma(\hat{q})) \cap r \ \subseteq\ \gamma(\alpha(+, C \wedge \gamma(\hat{q}), r)) \ .$$

Then:

$$\begin{aligned}
\gamma(\alpha(+, C, q \wedge r)) &= \gamma(\hat{q} \wedge^{\mathcal{A}} \alpha(+, C \wedge \gamma(\hat{q}), r)) && \text{(definition of } \alpha\text{)} \\
&= \gamma(\hat{q}) \wedge \gamma(\alpha(+, C \wedge \gamma(\hat{q}), r)) \\
&\supseteq \gamma(\hat{q}) \wedge (C \wedge \gamma(\hat{q})) \cap r && \text{(induction hypothesis)} \\
&= C \wedge \gamma(\hat{q}) \cap r \\
&\supseteq C \wedge (C \cap q) \cap r && \text{(induction hypothesis)} \\
&= C \cap (q \wedge r)
\end{aligned}$$

Thus,

$$C \cap (q \wedge r) \subseteq \gamma(\alpha(+, C, q \wedge r)),$$

which we wanted to show.

• $\alpha(+, C, q \vee r)$: Again, let $\hat{q} \stackrel{\text{def}}{=} \alpha(+, C, q)$. By induction hypothesis,

$$(C \wedge \neg\gamma(\hat{q})) \cap r \ \subseteq\ \gamma(\alpha(+, C \wedge \neg\gamma(\hat{q}), r)) \quad \text{and} \quad (C \wedge q) \subseteq \gamma(\hat{q}) \ .$$

Then:

$$\begin{array}{rcll}
\gamma(\alpha(+, C, q \vee r)) & = & \gamma(\hat{q} \vee^{\mathcal{A}} \alpha(+, C \wedge \neg\gamma(\hat{q}), r)) & \text{(definition of } \alpha) \\
& = & \gamma(\hat{q}) \vee \gamma(\alpha(+, C \wedge \neg\gamma(\hat{q}), r)) & \\
& \supseteq & \gamma(\hat{q}) \vee (C \wedge \neg\gamma(\hat{q}) \wedge r) & \text{(induction hypothesis)} \\
& = & \gamma(\hat{q}) \vee (C \wedge r) & \\
& \supseteq & (q \wedge C) \vee (C \wedge r) & \text{(induction hypothesis)} \\
& = & C \cap (q \vee r) &
\end{array}$$

- $\alpha(-, C, q \wedge r)$: Let $\hat{q} \stackrel{\text{def}}{=} \alpha(-, C, q)$. By induction hypothesis,

$$(C \wedge \gamma(\hat{q})) \cap \ \gamma(\alpha(-, C \wedge \gamma(\hat{q}), r)) \subseteq r \quad \text{and} \quad (C \wedge \gamma(\hat{q})) \subseteq q \ .$$

Since $Z \wedge X \subseteq Y$ implies $Z \subseteq Y \cup \neg X$, we know that

$$C \cap \gamma(\alpha(-, C \wedge \gamma(\hat{q}), r)) \subseteq r \cup \neg\gamma(\hat{q}) \tag{4.1}$$

Then:

$$\begin{array}{rcll}
C \cap \gamma(\alpha(-, C, q \wedge r)) & = & C \cap \gamma(\hat{q} \wedge^{\mathcal{A}} \alpha(-, C \wedge \gamma(\hat{q}), r)) & \text{(definition of } \alpha) \\
& = & C \cap (\gamma(\hat{q}) \ \wedge \ \gamma(\alpha(-, C \wedge \gamma(\hat{q}), r))) & \\
& = & (C \cap \gamma(\hat{q})) \ \wedge \ (C \cap \gamma(\alpha(-, C \wedge \gamma(\hat{q}), r))) & \\
& \subseteq & (C \cap \gamma(\hat{q})) \ \wedge \ (r \cup \neg\gamma(\hat{q})) & \text{(4.1 above)} \\
& = & (C \cap \gamma(\hat{q})) \ \wedge \ r & \\
& \subseteq & (q \wedge r) & \text{(induction hypothesis)}
\end{array}$$

□

As a check, note that $a \cap C \subseteq X$ is equivalent to $(\neg X) \cap C \supseteq (\neg a)$:

$$\begin{array}{rcl}
a \cap C \ \subseteq \ X & \leftrightarrow & \neg(a \cap C) \supseteq \neg X \\
& \leftrightarrow & (\neg a \cup \neg C) \supseteq \neg X \\
& \leftrightarrow & (\neg a) \supseteq (\neg X) \cap C
\end{array}$$

Hence, the condition $\alpha(-, C, X) = a$ is equivalent to $\neg\alpha(+, C, \neg X) = a$. Using the equation

$$\alpha(-, C, X) = \neg\alpha(+, C, \neg X)$$

we can derive the negative cases from the positive ones:

$$\begin{aligned}
\alpha(-,C,q\wedge r) &= \neg\alpha(+,C,\neg q\vee\neg r) \\
&= \neg(\underbrace{\alpha(+,C,\neg q)}_{X}\ \vee\ \alpha(+,C\wedge\neg X,\neg R)) \\
&= (\neg\alpha(+,C,\neg q))\ \wedge\ \neg\alpha(+,C\wedge\neg X,\neg r) \\
&= \alpha(-,C,q)\wedge\alpha(-,C\wedge\neg X,r)
\end{aligned}$$

But $\alpha(-,C,q)=\neg\alpha(+,C,\neg q)=\neg X$. Similarly,

$$\begin{aligned}
\alpha(-,C,q\vee r) &= \neg\alpha(+,C,\neg q\wedge\neg r) \\
&= \neg(\underbrace{\alpha(+,C,\neg q)}_{X}\ \wedge\ \alpha(+,C\wedge X,\neg r)) \\
&= (\neg\alpha(+,C,\neg q))\ \vee\ \neg\alpha(+,C\wedge X,\neg r) \\
&= \alpha(-,C,q)\ \vee\ \alpha(-,C\wedge X,r)
\end{aligned}$$

Again, $\alpha(-,C,q)=\neg\alpha(+,C,\neg q)=\neg X$.

Notice that this algorithm applies to any abstract domain that is a boolean algebra, provided the operations for $\wedge^{\mathcal{A}}$, $\vee^{\mathcal{A}}$, $\neg^{\mathcal{A}}$ and γ are available. Similarly, it applies to any assertion language for which a validity checker is available.

The stronger the validity checker, the better over- and under- approximations we will get, and the more temporal properties we will be able to prove, for a fixed basis. Note that if an assertion f contains only atoms in B, then we can expect its abstraction $f^{\mathcal{A}}$ to be equivalent to f modulo invariants. (More precisely, $\mathcal{S}\vDash\gamma(f^{\mathcal{A}})\leftrightarrow f$.)

4.3 Abstracting Systems

Given a concrete transition system

$$\mathcal{S}^{\mathcal{C}}:\left\langle\mathcal{V},\Theta,\mathcal{T}:\left\{\tau_1^{\mathcal{C}},\ldots,\tau_k^{\mathcal{C}}\right\}\right\rangle\quad,$$

its ∀CTL*-preserving abstraction is

$$\mathcal{S}^{\mathcal{A}}:\left\langle B:\{b_1,\ldots,b_n\},\Theta_{\mathcal{A}},\mathcal{T}_{\mathcal{A}}:\left\{\tau_1^{\mathcal{A}},\ldots,\tau_k^{\mathcal{A}}\right\}\right\rangle$$

where $\Theta_{\mathcal{A}}$ is the result of over-approximating Θ, and each abstract transition $\tau_i^{\mathcal{A}}$ is the result of over-approximating the corresponding $\tau_i^{\mathcal{C}}$.

System $\mathcal{S}^{\mathcal{A}}$ is an n-bit finite-state system. Since $\Theta \subseteq \gamma(\Theta_{\mathcal{A}})$ and $\tau \subseteq \gamma(\tau^{\mathcal{A}})$ for all $\tau \in \mathcal{T}$, the initiality and consecution conditions of Lemma 3.4.1 are satisfied. We satisfy the fairness condition by propagating the fairness of τ to $\tau^{\mathcal{A}}$ only if we can establish the validity of $\gamma(\mathit{enabled}(\tau^{\mathcal{A}})) \rightarrow \mathit{enabled}(\tau)$. In this case, the two enabling conditions are equivalent.[3] If the basis includes the atoms in the guard of τ, this is guaranteed to be the case (c.f. the abstract BAKERY of Figure 3.2). In the worst case no fairness carries over and only safety properties of $\mathcal{S}^{\mathcal{A}}$ (and hence $\mathcal{S}^{\mathcal{C}}$) can be proved.

Following [KMP94], we call this a *transition-mapped abstraction*, since there is a one-to-one mapping between abstract and concrete transitions. The mapping is *fairness-reducing*, since each transition $\tau^{\mathcal{C}}$ is at least as fair as the corresponding $\tau^{\mathcal{A}}$ (see Table 3.1).

The initial context can contain known invariants of $\mathcal{S}^{\mathcal{C}}$. When abstracting the atoms of an initial condition or the assertions of a temporal property (see below), we test against the set of unprimed points

$$P_U \stackrel{\text{def}}{=} B \cup \{\neg b_i \mid b_i \in B\} \quad .$$

For transition relations, we test against the set of mixed points

$$P_M \stackrel{\text{def}}{=} P_U \cup P'_U \cup \{p_1 \rightarrow p_2 \mid p_1 \in P_U \text{ and } p_2 \in P'_U\} \quad .$$

Thus, the algorithm abstracts a transition relation ρ_τ using $\mathrm{O}(n^2|\rho_\tau|)$ validity checks, where $n = |B|$. For an assertion f with no primed variables, $\mathrm{O}(n|f|)$ validity checks are needed. Enlarging these point sets can increase the quality of the abstraction, as discussed in Section 4.5; however, these relatively small sets were sufficient to verify most of the examples in Section 4.6 below.

4.4 Abstracting Temporal Properties

A temporal property φ is abstracted by under-approximating the positive polarity assertions that it contains and over-approximating those with negative polarity. This gives an abstract property $\varphi^{\mathcal{A}}$ that implies the optimal abstraction $\alpha^t(\varphi)$ of Definition 3.2.3, which assumes

[3] Strictly speaking, the two could differ on unreachable states, since known invariants of $\mathcal{S}^{\mathcal{C}}$ can be used to establish the conditions of Lemma 3.4.1 and to generate the abstractions.

negation-normal form.

This guarantees that every model of the abstract property $\varphi^{\mathcal{A}}$ corresponds to a model of the concrete one. Informally, consider the temporal tableau $\mathcal{G} : T_\varphi$ for φ (see Section 2.3.1). Let $\mathcal{G}'$ be the tableau obtained by replacing the assertion $\mu(n)$ in each node n of $\mathcal{G}$ by an under-approximation of $\mu(n)$. Any computation accepted by $\mathcal{G}'$ will also be accepted by $\mathcal{G}$. (This also justifies Definition 3.2.3, which uses the best possible approximation.)

Thus, if all computations of the abstract system satisfy $\varphi^{\mathcal{A}}$, all computations of the concrete system will satisfy φ. If the basis includes all of the assertions appearing in the property, the property approximation is exact.

Of course, if we can prove a property $\varphi^{\mathcal{A}}$ for $\mathcal{S}^{\mathcal{A}}$, then the concrete system $\mathcal{S}$ will satisfy $\gamma(\varphi^{\mathcal{A}})$, obtained by replacing each abstract assertion f in $\varphi^{\mathcal{A}}$ by its corresponding concretization $\gamma(f)$ (see Definition 3.2.3).

4.5 Optimizations

There are many possible improvements to the basic abstraction procedure described above, to increase efficiency and generate better approximations. We now describe some of them:

Preserving concrete variables: For convenience, we let $\mathcal{S}^{\mathcal{A}}$ retain some of the finite-domain variables of $\mathcal{S}$, as indicated by the user. (This is the case, for example, for the control variables of the abstract BAKERY of Figure 3.2). The finite-state model checker can then represent them explicitly or, for instance, encode them as bits (as an OBDD-based symbolic model checker would). We implement this by letting α be the identity on expressions with finite-domain whose free variables do not appear in the basis. Note, however, that the algorithm can always be used to abstract finite-state systems to smaller abstract ones.

Enlarging the test point set: Additional points must sometimes be tested to obtain a sufficiently precise abstraction. For example, τ may imply $(b_i \wedge b_j) \rightarrow b'_k$, but imply neither $b_i \rightarrow b'_k$ nor $b_j \rightarrow b'_k$. In our implementation, the user can specify additional points to test for particular transitions when specifying the basis. Alternatively, the user may enlarge the basis, but this will in general not only increase the abstraction time, but also increase the time and space used at model-check time.

4.6 Examples

Consider the BAKERY program from Figure 2.1 and the basis

$$B : \left\{ \begin{array}{ll} b_1 : & y_1 = 0, \\ b_2 : & y_2 = 0 \\ b_3 : & y_1 \leq y_2 \end{array} \right\}$$

obtained from the guards of the **await** statements (that is, their enabling conditions). Using this basis, and preserving the finite-state (control) variables, the finite-state abstraction of BAKERY, shown in Figure 3.2 is generated. This abstract program can be model checked to verify the basic safety properties of the original system, including mutual exclusion and one-bounded overtaking (see Example 2.2.8).

The atoms in the guards of transitions ℓ_2 and m_2 are included in the basis (negated, in the case of $y_2 < y_1$); thus, they are abstracted exactly. By Theorem 3.4.1, the abstract transitions can inherit the fairness properties of the original ones. The same trivially applies to $\ell_1, \ell_3, \ell_4, m_1, m_3, m_4$, since their guards are *true*. Thus, we can also prove the response property of accessibility,

$$\square(\ell_1 \rightarrow \diamondsuit\, \ell_3) \ ,$$

by model checking the abstract system.

Example 4.6.1 (Abstracting Fischer) *For the Fischer program of Figure 2.2, we used the following basis [CU98]:*

$$\begin{array}{llll} b_1: & c_1 \geq L & b_4: & c_2 \geq c_1 \\ b_2: & c_2 \geq L & b_5: & c_1 \geq c_2 + L \\ b_3: & c_1 \geq c_2 & b_6: & c_2 \geq c_1 + L \end{array}$$

The initial context contained assumptions $\{L > 0,\ U > 0,\ U \geq L,\ 2L > U\}$ *and invariants* $\{c_1 \geq 0,\ c_2 \geq 0\}$. *The initial condition was abstracted to*

$$\pi_1 = \ell_0 \wedge \pi_2 = m_0 \wedge x = 0 \wedge \neg b_1 \wedge \neg b_2 \wedge b_3 \wedge b_4 \wedge \neg b_5 \wedge \neg b_6$$

$$\textbf{local } x : \{0,1,2\} \textbf{ where } x = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \textbf{skip} \\ \ell_3\text{: } \textbf{while } x \neq 1 \textbf{ do} \\ \quad \left[\begin{array}{l} \ell_4\text{: } \textbf{await } x = 0 \\ \ell_5\text{: } x := 1 \\ \ell_6\text{: } \textbf{skip} \\ \ell_7\text{: } \textbf{if } x = 1 \textbf{ then} \\ \quad \left[\ell_8\text{: } \textbf{critical}\right] \end{array}\right] \\ \ell_9\text{: } \textbf{skip} \\ \ell_{10}\text{: } x := 0 \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{noncritical} \\ m_2\text{: } \textbf{skip} \\ m_3\text{: } \textbf{while } x \neq 2 \textbf{ do} \\ \quad \left[\begin{array}{l} m_4\text{: } \textbf{await } x = 0 \\ m_5\text{: } x := 2 \\ m_6\text{: } \textbf{skip} \\ m_7\text{: } \textbf{if } x = 2 \textbf{ then} \\ \quad \left[m_8\text{: } \textbf{critical}\right] \end{array}\right] \\ m_9\text{: } \textbf{skip} \\ m_{10}\text{: } x := 0 \end{array}\right] \end{array}\right]$$

Figure 4.3: Fischer's mutual exclusion algorithm (complete version)

(where we now write $\wedge, \vee, \neg$ rather than $\wedge^{\mathcal{A}}, \vee^{\mathcal{A}}, \neg^{\mathcal{A}}$). Transition ℓ_1 was abstracted to

$$\rho^{\mathcal{A}}_{\ell_1} : \left(\begin{array}{c} \pi_1 = \ell_1 \wedge \pi_1' = \ell_2 \wedge x' = 1 \wedge \pi_2' = \pi_2 \wedge \\ b_1 \wedge \neg b_1' \wedge b_4' \wedge \neg b_5' \wedge (b_2 \rightarrow \neg b_3') \wedge (b_2 \rightarrow b_6') \wedge \\ (\neg b_3 \rightarrow \neg b_3') \wedge (\neg b_3 \rightarrow b_6') \wedge (b_4 \rightarrow \neg b_3') \wedge (b_4 \rightarrow b_6') \wedge \\ (\neg b_5 \rightarrow \neg b_3') \wedge (b_6 \rightarrow \neg b_3') \wedge (b_6 \rightarrow b_6') \end{array}\right) .$$

The other transitions were similarly abstracted. (The tick *transition, which contains quantifiers, was treated as a single literal when abstracted.) With our STeP implementation, the abstract system was generated in 28 seconds, and mutual exclusion was automatically model checked in one second.*

Figure 4.3 shows a more complete version of Fischer's algorithm, for which mutual exclusion can be verified using the same abstraction basis; even though the control structure is more complex, the relevant relationships between the data variables remain the same.

Table 4.1 shows abstraction times for several other examples, from [CU98], where the main safety property of each system was model checked.[4] In practice, finding the right abstraction is an iterative process, where different assertion bases are tested, and extra test points are used. Section 5.5 addresses these issues.

[4] Obtained on a Sun SPARC 2 workstation, approx. 200 Mhz. These are *measurements*, in the sense of [Hol98], and not intended to validate or refute any particular scientific hypothesis.

System	# transitions	Basis size	Abstraction time	Model check time
Bakery	14	3	3s	<1s
Fischer (fragment)	11	6	28s	1s
Alternating-bit	7	4	14s	<1s
Bounded Retransmission	13	7	70s	4s

Table 4.1: Abstraction and model checking times

4.7 Minimizing Validity Checks

The most expensive part of the abstraction algorithm is calling *checkValid*. The performance can be greatly improved if the number of calls is minimized. We now present a few simple strategies that eliminate trivial or redundant test points.

4.7.1 Terminating the Recursion

When a subexpression consists solely of conjunctions of literals, we eliminate redundant validity checks by testing each point once for the entire subexpression. That is, we terminate the recursion early, since testing the points for each atom will not improve the quality of the abstraction.

This assumes that *checkValid* is monotonic, in the sense that if $\mathit{checkValid}(C \rightarrow p)$ succeeds and $C' \subseteq C$, then $\mathit{checkValid}(C' \rightarrow p)$ should also succeed.

Theorem 4.7.1 *If φ and ψ are pure atoms and the methods of Section 4.1 and 4.2 are used, then $\alpha(+, \varphi \wedge \psi)$ and $\alpha(-, \varphi \vee \psi)$ can be computed directly without loss of precision.*

Proof: Consider $\alpha(+, C, a \wedge b)$ where a and b are atoms. If computed directly, this would be the conjunction $\bigwedge S$ of all the elements of

$$S \stackrel{\text{def}}{=} \{f \in P \mid \mathit{checkValid}((a \wedge b \wedge C) \rightarrow f)\} \quad .$$

We claim that the same result would be obtained if we recurse into the $\wedge$:

Let $\hat{a} \stackrel{\text{def}}{=} \alpha(+, C, a)$. If we recurse, the result would be $I : \hat{a} \wedge \alpha(+, C \wedge \hat{a}, b)$. Assume that some formula $f \in P$ appears as a conjunct in both $\hat{a}$ and $\alpha(+, C \wedge \hat{a}, b)$, so that it appears in I. Then $(b \wedge (\hat{a} \wedge C)) \to f$ and $(a \wedge C) \to f$ are valid. Therefore, $(a \wedge b \wedge C \to f)$ is valid, so $f \in S$. Thus, $(\bigwedge S) \subseteq I$.

Assume now that $f \in S$, that is, *checkValid*$((a \wedge b \wedge C) \to f)$ succeeds. Since $(C \wedge a) \subseteq \hat{a}$ (by the definition of $\hat{a}$ and the correctness of α) this means that *checkValid*$((\hat{a} \wedge b \wedge C) \to f)$ also succeeds, assuming that *checkValid* is monotonic. Thus, f is a conjunct in $\alpha(+, C \wedge \hat{a}, b)$. Also, f is implied by $\hat{a}$, so f is implied by I. Thus, $I \subseteq \bigwedge S$. □

A similar argument holds if one or both of the atoms in the conjunction is a mixed atom. Dual arguments apply to $\alpha(-, \varphi \vee \psi)$. What this means is that the contextual over-approximation can benefit mostly from the presence of disjunctions (non-determinism) in the transition relation, while under-approximation can benefit from conjunctions.

We say that transitions τ_1 and τ_2 are *overlapping* if $\mathit{enabled}(\tau_1) \cap \mathit{enabled}(\tau_2) \neq \emptyset$. To exploit disjunctions more, overlapping transitions can be combined, as their disjunction, and abstracted together.

However, we must again be careful if one of the transitions is fair. The combined transition relation cannot have any fairness requirement unless they are both compassionate, in which case the resulting transition can also be compassionate (or uniformly compassionate, if both are uniformly compassionate as well, see Section 3.4.1). Nonetheless, for safety properties, joining transitions can lead to finer consecution constraints on the abstract system.

4.7.2 Propositional Optimization

Any information available about the relationship between basis elements can be used to eliminate redundant tests. As a simple example, if we know that an atom implies b_i, it is unnecessary to test the point $\neg b_i \to b'_j$ for that atom.

More generally, together with the context, which is an assertion over the concrete variables, we can maintain a corresponding *abstract context* $C^{\mathcal{A}}$, a propositional formula over the abstract variables. This abstract context will contain information about the current approximation. Whenever C is updated with $\gamma(f)$, the abstract context $C^{\mathcal{A}}$ is updated with f.

Normally, the two contexts will be related by $C = \gamma(C^{\mathcal{A}})$, but this does not always have to be the case. The initial abstract context can be conjoined with over-approximations of concrete invariants, as well as boolean relationships, previously established, among the basis variables. This information may not be included in the initial context C, e.g., if the validity checker already accounts for them, or if it slows down the validity checking by making C too large; thus, in general $\gamma(C^{\mathcal{A}}) \subseteq C$.

A point is not tested if it is deemed propositionally redundant with respect to $C^{\mathcal{A}}$.

Definition 4.7.2 (Propositional $\mathit{filter}(b, P)$**)** *For a set of points P and an abstract context $C^{\mathcal{A}}$, let $\mathit{filter}(C^{\mathcal{A}}, P)$ be the set of points p in P such that $C^{\mathcal{A}}$ does not propositionally imply p or $\neg p$.*

These implications can be quickly checked if $C^{\mathcal{A}}$ is maintained as an OBDD.

Remark 4.7.3 (Using OBDDs) Note that OBDDs can also be used to represent the abstract transition relation. We can also construct an OBDD expressing the relationships between the basis elements, including any abstract invariants. This OBDD, and its primed version, can be conjoined to the abstract transition relations, giving an abstract system that is ripe for being symbolically model checked. $\square$

We now redefine α_{atom} as:

$$\begin{aligned}
\alpha_{atom}(+, C^{\mathcal{A}}, C, a) &\stackrel{\text{def}}{=} \bigwedge\nolimits^{\mathcal{A}} \left\{p \in \mathit{filter}(C^{\mathcal{A}}, P) \mid \mathit{checkValid}((C \wedge a) \rightarrow \gamma(p))\right\} \\
\alpha_{atom}(-, C^{\mathcal{A}}, C, a) &\stackrel{\text{def}}{=} \bigvee\nolimits^{\mathcal{A}} \left\{p \in \mathit{filter}(C^{\mathcal{A}}, P) \mid \mathit{checkValid}((C \wedge \gamma(p)) \rightarrow a)\right\}
\end{aligned}$$

The revised version of α is then:

$$\begin{aligned}
\alpha(\pi, C^{\mathcal{A}}, C, a) &= \alpha_{atom}(\pi, C^{\mathcal{A}}, C, a), \text{ if } a \text{ is an atom} \\
\alpha(\pi, C^{\mathcal{A}}, C, \neg q) &= \neg^{\mathcal{A}} \alpha(\pi^{-1}, C^{\mathcal{A}}, C, q) \\
\alpha(\pi, C^{\mathcal{A}}, C, q \wedge r) &= \textbf{let } \hat{q} = \alpha(\pi, C^{\mathcal{A}}, C, q) \textbf{ in } \hat{q} \wedge^{\mathcal{A}} \alpha(\pi, C^{\mathcal{A}} \wedge \hat{q}, C \wedge \gamma(\hat{q}), r) \\
\alpha(\pi, C^{\mathcal{A}}, C, q \vee r) &= \textbf{let } \hat{q} = \alpha(\pi, C^{\mathcal{A}}, C, q) \textbf{ in } \hat{q} \vee^{\mathcal{A}} \alpha(\pi, C^{\mathcal{A}} \wedge \neg\hat{q}, C \wedge \neg\gamma(\hat{q}), r) \ .
\end{aligned}$$

If this optimization is used, then recursing into conjunctions can save time, since the propositional context $C^{\mathcal{A}}$ can help restrict validity checking to the conjuncts that are *not* propositionally redundant.

4.7.3 Incremental Computation

The optimized procedure of the previous section is also useful when computing abstractions *incrementally*, where a given abstraction is used as a starting point for constructing a better one. This situation arises, for instance, if we have a hierarchy of validity checkers, each one more powerful, but less efficient, than the previous. Incremental computation is also desirable if, having constructed an abstraction, we want to refine it by adding new elements to the basis (see Chapter 5).

In these situations, we can minimize the number of redundant validity checks in re-computing $\alpha(\rho)$ by adding the previous abstraction $\rho_{\mathcal{A}}$ as a conjunct to the initial abstract context $C^{\mathcal{A}}$. In the case that new basis elements are used, any known propositional constraint relating the new and old basis elements should be conjoined with $C^{\mathcal{A}}$ as well.

4.7.4 Variable Dependencies

We can analyze the dependencies between subformulas to predict when the validity checker is likely to fail. For example, if τ does not modify the free variables of b_i, we can eliminate the points $\{p \rightarrow b_i' \mid p \in P_U\}$ when abstracting τ.

In general, when proving $C \wedge a \rightarrow b$, we can assume that the validity checker would have failed if the context does not connect the variables in a with those of b. This can be implemented by maintaining equivalence classes of variables as part of the context, using, e.g., a union-find structure [Tar75], where initially each variable is in its own class. When an assertion is added to the context, the equivalence classes for the variables appearing in the assertion are merged. A point $p \rightarrow q'$ is only tested if p and q share at least one equivalence class.

Strictly speaking, this does not mean that the validity checker would have failed: the concrete context may be unsatisfiable, for instance.[5] However, reducing the number of tests can only increase the degree of approximation, so soundness is not compromised.

4.8 Invariant Generation

In general, any abstraction can be used to generate invariants of the concrete system, as pointed out, e.g., in [BBM97, GS97].

[5]This should terminate the recursion if detected, of course.

Let *reachable*$(\mathcal{S}^{\mathcal{A}})$ be the boolean formula that characterizes the reachable states of $\mathcal{S}^{\mathcal{A}}$. The assertion $\gamma($*reachable*$(\mathcal{S}^{\mathcal{A}}))$ is obtained by replacing each boolean variable by its corresponding assertion. This formula may be too large to be useful, e.g., when *reachable*$(\mathcal{S}^{\mathcal{A}})$ is a large OBDD. However, any formula that is *implied* by *reachable*$(\mathcal{S}^{\mathcal{A}})$ is also an invariant of the system.

In the OBDD case, for instance, we can then look for short paths from the OBDD root node to the *false* terminal node: if $\pm b_1 \wedge \ldots \wedge \pm b_i$ is an assignment to the OBDD variables that leads to *false*, then $\neg(\gamma(\pm b_1) \wedge \ldots \wedge \gamma(\pm b_i))$ is an invariant of $\mathcal{S}$.[6] Similarly, control invariants for $\mathcal{S}^{\mathcal{A}}$ can be obtained by existentially quantifying out data-related abstract variables in *reachable*$(\mathcal{S}^{\mathcal{A}})$.

This leads to the intriguing possibility of *bootstrapping*, where invariants generated by an abstraction are used in turn to generate a better abstraction, building on the first. (See the discussion of incremental generation of abstractions in Section 4.7.3.)

We can do the reverse too: abstract the invariants of the concrete system, by over-approximating them (expressed as assertions) to obtain invariants for the abstract system. These can then help reason about $\mathcal{S}^{\mathcal{A}}$, e.g., constraining the model checking process.

4.9 Preserving Existential Properties

As described in Section 3.5, a second abstract transition relation must be considered if existential properties are to be preserved.

To model check CTL* properties that combine existential and universal path quantification, the model checker should use both abstract transition relations, considering one or the other depending on the existential quantification being analyzed. In Chapter 6, we will see that the constrained transition relation can also help find abstract counterexamples that correspond to concrete ones.

4.9.1 $\forall\exists$ and $\exists\exists$ Approximations

In the universal-preservation case, considered above, there should be an abstract transition from a_1 to a_2 if *some* concrete state in $\gamma(a_1)$ can reach *some* concrete state in $\gamma(a_2)$. Hence, over-approximation is exactly what is needed. The smaller the abstract transition relation

[6] Recent work by the authors of [BLO98] and Berezin [Ber98] explores the question of generating usable invariants from such OBDD's.

is (that is, the more exact the over-approximation is), the more universal properties will be satisfied. Following [Dams96], we call this the *free* transition relation, and say that $\tau^{\mathcal{A}}$ $\exists\exists$-*approximates* τ:

Definition 4.9.1 ($\exists\exists$-approximation) *An abstract relation $\rho_{\mathcal{A}}$ is an* $\exists\exists$-approximation *of a concrete relation ρ if*

$$\rho \subseteq \gamma(\rho_{\mathcal{A}}) \ .$$

The name comes from the fact that if $\rho(s_1, s_2)$, then there should *exist* a_1 and a_2 such that $s_1 \in \gamma(a_1)$ and $s_2 \in \gamma(a_2)$. This is built into the definition of γ for relations (Definition 3.1.3).

As described in Section 3.5, to preserve existential properties we need a separate *constrained* abstract transition relation $R_{\mathcal{A}}^{\forall\exists}$. We can describe it as an abstract transition $\tau_{\forall\exists}^{\mathcal{A}}$ for each concrete transition τ, where $\tau_{\forall\exists}^{\mathcal{A}}$ goes from a_1 to a_2 only if *for all* concrete states in $\gamma(a_1)$ it is possible to reach a concrete state in $\gamma(a_2)$. We call this the *constrained* transition relation, and say that $\tau_{\forall\exists}^{\mathcal{A}}$ $\forall\exists$-*approximates* τ:

Definition 4.9.2 ($\forall\exists$-approximation) *An abstract relation $\rho_{\mathcal{A}}$ is a* $\forall\exists$-approximation *of a concrete relation ρ if whenever $\rho_{\mathcal{A}}(a_1, a_2)$ holds, then*

$$\gamma(a_1)(\mathcal{V}) \rightarrow \exists \mathcal{V}' \,.\, (\rho(\mathcal{V}, \mathcal{V}') \wedge \gamma(a_2)(\mathcal{V}'))$$

is valid.

Theorem 3.5.3 can be rephrased by saying that each constrained abstract transition $\tau_{\forall\exists}^{\mathcal{A}}$ must be a $\forall\exists$-approximation of the corresponding τ for CTL* preservation to hold. The larger the $\forall\exists$-transition relation is, the more existential properties will be satisfied by the abstract system. In general, if p and q are assertions over the basis B, we can add $p \wedge^{\mathcal{A}} q'$ as a disjunct to $\tau_{\forall\exists}^{\mathcal{A}}$ if we can prove

$$p(\mathcal{V}) \rightarrow \exists \mathcal{V}. \, (\tau(\mathcal{V}, \mathcal{V}') \wedge q(\mathcal{V}')) \ .$$

Remark 4.9.3 The "free" and "constrained" terminology becomes more intuitive by noting that the free ($\exists\exists$) relation can always be made larger without losing soundness, while the constrained one ($\forall\exists$) cannot. □

4.9.2 $\forall\exists$ Considerations

The question, now, is how to obtain abstract $\forall\exists$ transition relations. The case is not entirely dual to the $\exists\exists$-approximation of the preceding sections: while under-approximation is *sufficient*, it is not the case that (for a fixed abstract domain) the best under-approximation of τ gives the best constrained $\tau^{\mathcal{A}}_{\forall\exists}$:

Assume that an abstract relation $\rho_{\mathcal{A}}$ under-approximates a concrete relation ρ, that is, $\gamma(\rho_{\mathcal{A}}) \subseteq \rho$. Thus, if $\rho_{\mathcal{A}}(a_1, a_2)$ holds, then *all* elements of $\gamma(a_1)$ are ρ-related to *all* elements of $\gamma(a_2)$ (so, indeed, $\rho_{\mathcal{A}}$ $\forall\exists$-approximates ρ). But it is enough for each element of $\gamma(a_1)$ to be ρ-related to only *one* element of $\gamma(a_2)$. Thus, intuitively, $\rho_{\mathcal{A}}$ can be made larger while still being a sound $\forall\exists$-approximation of ρ.

Unfortunately, the compositional approximation algorithm of the previous sections is not directly applicable in this case. For example, consider the case of abstracting the relation $\tau : A \wedge B$. Assume that we can prove

$$p_1(\mathcal{V}) \rightarrow \exists \mathcal{V}'.(A \wedge q_1(\mathcal{V}'))$$

and

$$p_2(\mathcal{V}) \rightarrow \exists \mathcal{V}'.(B \wedge q_2(\mathcal{V}'))$$

so that $p_1 \wedge^{\mathcal{A}} q_1'$ (resp. $p_2 \wedge^{\mathcal{A}} q_2'$) is a correct $\forall\exists$-approximation of A (resp. B). However, this does not imply the validity of

$$(p_1 \wedge p_2)(\mathcal{V}) \rightarrow \exists \mathcal{V}'.(A \wedge B \wedge (q_1 \wedge q_2)(\mathcal{V}')) \ ,$$

so we cannot add $(p_1 \wedge^{\mathcal{A}} q_1') \wedge^{\mathcal{A}} (p_2 \wedge^{\mathcal{A}} q_2')$ (as a disjunct) to $\tau^{\mathcal{A}}_{\forall\exists}$. Instead, we can only add $(p_1 \wedge^{\mathcal{A}} p_2) \wedge^{\mathcal{A}} (q_1' \vee^{\mathcal{A}} q_2')$.

The $\forall\exists$-approximation relation does distribute over disjunction, so we can approximate each transition individually: if $\tau_1^{\mathcal{A}}$ $\forall\exists$-approximates τ_1 and $\tau_2^{\mathcal{A}}$ $\forall\exists$-approximates τ_2, then $\tau_1^{\mathcal{A}} \vee^{\mathcal{A}} \tau_2^{\mathcal{A}}$ correctly $\forall\exists$-approximates $\tau_1 \vee \tau_2$.

The design of good $\forall\exists$-approximation algorithms is left as a question for future work. We note that (1) the above discussion addresses $\wedge$ and $\vee$, (2) negations can be pushed down to the atomic level, (3) standard under-approximations, e.g., $\alpha(-, C, f)$ can be disjoined with any $\tau^{\mathcal{A}}_{\forall\exists}$ to get a larger and better $\forall\exists$-approximation, (4) if q is the postcondition of an under-approximation q of the enabling condition of τ, then $p \wedge q'$ can be added to $\tau^{\mathcal{A}}_{\forall\exists}$, and

(5) if the $\forall\exists$ postcondition is limited to a set q, and we can compute the (weak) precondition $pre(\tau, \neg q)$, then the result should *not* be in the pre-image of q for $\tau^{\mathcal{A}}_{\forall\exists}$. We also note that the verification conditions that must be proved are generally harder than for the universal case, due to the alternated quantifiers (which are first-order, in general).

4.10 Related Work

The assertion-based abstract domain is presented in [BBM97], as one of a number of abstract domains that can be used to generate invariants once a fixpoint that over-approximates the reachable states is found (see Section 7.1.2).

In [BLO98], a similar framework for generating abstractions is presented. The framework is essentially the same—Assertion Based Abstraction—but some details differ:

- The method in [BLO98] is geared towards proving invariants, of the form $\Box\,\varphi$ for an assertion φ. The invariant to be proved is assumed when generating the abstraction, yielding a better abstract system: more specifically, the verification conditions $\{\psi\}\,\tau\,\{\chi\}$ are replaced by $\{\psi \wedge \varphi\}\,\tau\,\{\chi\}$.

  We can achieve the same effect by including φ in the initial context; the correctness of the abstraction is then contingent on proving φ over $\mathcal{A}$.

- For efficiency, the basis B is partitioned into sets of abstract variables $S_1, \ldots, S_k$. When generating the abstract system, only relationships between variables in the same set are considered.

- A "substitution method" is used to make more immediate replacements.

These three improvements can be directly combined with our approach. The second one has the goal of reducing the number of test points considered. Heuristics to determine which abstract variables are more likely to be related to each other can be used (c.f. heuristics for finding good variable orderings for OBDD's).

The generation of abstractions proceeds by an "elimination method," where transitions from abstract state s_i to s_j are eliminated from a transition τ if $\{s_i\}\,\tau\,\{\neg s_j\}$ is proved to be valid. We will see this again in Section 5.5.

Graf and Saidi [GS97] present an alternate procedure for generating assertion-based abstractions. The procedure uses validity checking to generate a finite-state abstraction based on a set of formulas $B : \{b_1, \ldots, b_n\}$. The abstract state-space $\Sigma_{\mathcal{A}}$ is the complete lattice of $3^n + 1$ *monomials* over B. A monomial is either *false*, or a set that may contain φ_i or $\neg\varphi_i$, but not both. The ordering $\preceq$ over $\Sigma_{\mathcal{A}}$ is implication. Given an abstract state s, an approximation of its successors is found by deciding which of $\{\varphi_i, \neg\varphi_i\}$ are implied by $post(R, \gamma(s))$. This is done using an automatic strategy (a *tactic*) of the PVS theorem prover [ORR+96]. In the cases where the proof strategy fails (because the implications do not hold, or because the strategy was not powerful enough), the next-state does not include the corresponding φ_i or its negation.

Rather than performing an exhaustive search of the reachable abstract states while constructing $\mathcal{S}^{\mathcal{A}}$, our algorithm transforms $\mathcal{S}^{\mathcal{C}}$ to $\mathcal{S}^{\mathcal{A}}$ directly, leaving the exploration of the abstract state-space to a model checker. Thus, the number of validity checks is proportional to the number of formulas in B and the size of the representation of $\mathcal{S}^{\mathcal{C}}$, rather than the size of the abstract state-space. Furthermore, our procedure is applicable to systems whose abstract state-space is too large to enumerate explicitly, but can still be handled by a symbolic model checker (see Section 2.3).

The price paid by our approach, compared to [GS97], is that a coarser abstraction may be obtained. This is compensated by using a richer abstract state-space: the complete boolean algebra of expressions over $B : \{b_1, \ldots, b_n\}$, rather than only the monomials over this set. Using the complete algebra is prohibitively expensive in the [GS97] approach. On the other hand, the latter has the relative advantage of exploring only the reachable states, if this number is small.

Summary: Validity checking and propositional reasoning can be used in batch mode to generate finite-state abstractions, given an assertion basis provided by the user. The question now is how a suitable basis can be found. Chapter 5 addresses this question, which requires user interaction in general.

Chapter 5

Interactive Abstraction: Rules, Diagrams and DMC

This chapter shows how assertion-based abstraction is implicitly used in the deductive-algorithmic formalism of Generalized Verification Diagrams (GVD's) (Section 5.3), and hence the more specific (and mostly deductive) methods of verification rules and (standard) verification diagrams.

We then show how the related method of Deductive Model Checking (DMC) can be viewed as abstraction refinement, where a suitable abstract system is constructed incrementally and interactively (Section 5.5).

This provides benefits in both directions. On the one hand, the techniques of DMC and GVD's can be used to enhance the abstraction framework to make it (relatively) complete. We will see that they can make finer distinctions regarding fairness and well-founded orders, compared to the transition-mapped abstractions of Chapter 4. This will lead us to propose an extended model checking procedure in Chapter 6.

On the other hand, the abstraction point of view yields an alternative presentation of DMC and GVD's that is simpler to formulate, and leads to alternative implementation schemes. In particular, it allows the reuse and combination of the abstractions that they implicitly generate.

	For assertions $\varphi : \gamma(b_1)$ and p,	
I1.	$\Theta \rightarrow \varphi$	*initiality for* $\mathcal{S}^{\mathcal{A}}$
I2.	$\{\varphi\}\ \tau\ \{\varphi\}$ for each $\tau \in \mathcal{T}$	*consecution for* $\mathcal{S}^{\mathcal{A}}$
I3.	$\varphi \rightarrow p$	*concretization*
	$\mathcal{S} \models \Box\, p$	

Figure 5.1: General invariance rule G-INV revisited

5.1 Example: Invariance Rule

Recall the general invariance rule G-INV, which proves an invariant $\Box\, p$ by finding an inductive assertion φ that strengthens p. As noted in, e.g., [BLO98, KP98], it is easy to see that the success of rule G-INV implies the existence of an abstract system that can prove the property in question. The verification conditions imply that the abstract system with a single state $s^{\mathcal{A}}$ that represents all the concrete states that satisfy φ, in which every abstract transition $\tau^{\mathcal{A}}$ leads from $s^{\mathcal{A}}$ to itself, is a correct abstraction of $\mathcal{S}$.

Figure 5.1 presents this rule, indicating how the verification conditions correspond to abstraction. In our terminology, if we choose the one-bit basis $B : \{b^{\varphi}\}$, then the verification conditions prove the correctness of an assertion-based abstract system

$$\mathcal{S}^{\mathcal{A}} : \left\langle \mathcal{BA}(\{b^{\varphi}\}), \Theta_{\mathcal{A}} : b^{\varphi}, \mathcal{T}_{\mathcal{A}} : \left\{ \tau^{\mathcal{A}} : b^{\varphi} \rightarrow^{\mathcal{A}} b^{\varphi\prime} \right\} \right\rangle \ .$$

Trivially, we can model check $\Box\, b^{\varphi}$ for this system. Verification condition I3 proves that

$$\gamma(\Box\, b^{\varphi}) \text{ implies } \Box\, p,$$

so we can conclude that $\mathcal{S} \models \Box\, p$.

The (relative) completeness of rule G-INV [MP91a] implies the completeness of the abstraction approach—provided the adequate strengthening φ is found. Conversely, assume we have an abstraction $\mathcal{S}^{\mathcal{A}}$ that proves an invariant $\Box\, p$. Then the reachable states of $\mathcal{S}^{\mathcal{A}}$ describe a strengthened assertion φ that can be used to prove $\Box\, p$ using rule G-INV: take $\varphi : \gamma(\textit{reachable}(\mathcal{S}^{\mathcal{A}}))$, where $\textit{reachable}(\mathcal{S}^{\mathcal{A}})$ characterizes the reachable states of $\mathcal{S}^{\mathcal{A}}$. As discussed in Section 4.8, any assertion implied by $\gamma(\textit{reachable}(\mathcal{S}^{\mathcal{A}}))$ is an invariant of $\mathcal{S}$.

For assertions $p, q_0, \ldots, q_m$, $\varphi_0 : \gamma(b_0), \ldots, \varphi_m : \gamma(b_m)$,

W1.	$\{\varphi_i\}\ \tau\ \{\varphi_i \vee \ldots \vee \varphi_0\}$, for each $i = 1, \ldots, m$ and $\tau \in \mathcal{T}$	*consecution for* $\mathcal{S}^{\mathcal{A}}$
W2.	$p \ \rightarrow\ (\varphi_m \vee \ldots \vee \varphi_0)$	*concretization I*
W3.	$\varphi_i \ \rightarrow\ q_i$, for $i = 0, \ldots, m$	*concretization II*

$\mathcal{S} \vDash\ p \Rightarrow (q_m\, \mathcal{W}\ \ldots\ \mathcal{W}\, q_1\, \mathcal{W}\, q_0)$

Figure 5.2: Rule G-WAIT

5.2 Example: Wait-for Rule

Figure 5.2 presents the *general wait-for rule* [MP95b] in the same light. Premise W1 lets us construct a property-preserving abstract system

$$\mathcal{S}^{\mathcal{A}} : \left\langle \mathcal{BA}(\{b^{\varphi_0}, \ldots, b^{\varphi_m}\}), \Theta_{\mathcal{A}} : true^{\mathcal{A}}, \mathcal{T}_{\mathcal{A}} \right\rangle \ ,$$

where $\gamma(b^{\varphi_i}) = \varphi_i$, for which

$$\varphi^{\mathcal{A}} \ :\ \Box\left((b^{\varphi_0} \vee^{\mathcal{A}} \ldots \vee^{\mathcal{A}} b^{\varphi_m}) \ \rightarrow^{\mathcal{A}}\ b^{\varphi_m}\, \mathcal{W}\ \ldots\ \mathcal{W}\, b^{\varphi_0} \right)$$

can be successfully model checked. Each verification condition $\{\varphi_i\}\ \tau\ \{\varphi_i \vee \ldots \vee \varphi_0\}$ adds the conjunct $b^{\varphi_i} \rightarrow^{\mathcal{A}} (b^{\varphi_i}{}' \vee^{\mathcal{A}} \ldots \vee^{\mathcal{A}} b^{\varphi_0}{}')$ to the transition relation $\tau^{\mathcal{A}}$. Since there is no fairness, these can be collected together into a single transition relation R.

Premises W2 and W3 establish that

$$\gamma(\varphi^{\mathcal{A}}) \text{ implies } (p \Rightarrow q_m\, \mathcal{W}\ \ldots\ \mathcal{W}\, q_0) \ .$$

The case of rules for progress properties is slightly different, and we address it in Section 5.4.

5.3 Generalized Verification Diagrams

While different kinds of verification rules and diagrams can be formulated for different classes of temporal properties (see Section 5.1), *Generalized Verification Diagrams* (GVD's) offer a single deductive-algorithmic formalism that is applicable to arbitrary temporal properties [BMS95, BMS96]. We now summarize the description of GVD's from [MBSU98], which offers a simpler presentation and clarifies the connection with abstraction. We will see that in addition to being a proof, a GVD is a (weakly) $\forall$CTL* property-preserving assertion-based abstraction of the system. This connection is formalized in Sections 5.3.2, but we will informally point out the highlights along the way.

A *verification diagram* is a labeled graph, where nodes are labeled by assertions. Our diagrams will not use edge labels. For a node n, the assertion that labels n is $\mu(n)$, and $succ(n)$ is the set of its successor nodes. For a set of nodes $S : \{n_1, \ldots, n_k\}$, we define

$$\mu(S) \stackrel{\text{def}}{=} \mu(n_1) \vee \ldots \vee \mu(n_k)\,,$$

where $\mu(\{\}) = \mathit{false}$.

Definition 5.3.1 (Generalized Verification Diagram) *A* Generalized Verification Diagram *(GVD) Ψ is a verification diagram with a distinguished set of* initial nodes *and an* acceptance condition *$\mathcal{F}$, which is a* set of sets *of nodes of Ψ.*

As we will see, GVD's can be thought of as ω-automata. The set $\mathcal{F}$ is a *Müller acceptance condition* (see [Tho90]), which we use instead of the more concise but less intuitive *Streett acceptance conditions* used in [BMS95].

As defined in [BMS95, MBSU98], a *run* of a diagram is a sequence $\sigma : s_0, s_1, \ldots$ of states of $\mathcal{S}$ such that there is an associated path $\pi : n_0, n_1, \ldots$ through the diagram, where n_0 is an initial node and for each $i \geq 0$, the state s_i satisfies $\mu(n_i)$.

For an infinite path π through a GVD, let $\mathit{inf}(\pi)$ be the set of nodes that appear infinitely often in π. A path π is *accepting* if $\mathit{inf}(\pi) \in \mathcal{F}$. A *computation* of a diagram Ψ is a run of Ψ that has an associated accepting path. Since $\mathit{inf}(\pi)$ must be an SCS for any path π, the set $\mathcal{F}$ only needs to contain *strongly connected subgraphs* (SCS's) of Ψ. The set of all computations of Ψ is $\mathcal{L}(\Psi)$.

5.3.1 The $(\mathcal{S}, \Psi)$ Verification Conditions: Abstraction

Given a GVD Ψ and a system $\mathcal{S}$, verification conditions are proved to ensure that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$. In this case, we say that Ψ is $\mathcal{S}$*-valid*. These verification conditions, as we will see, ensure that $\Psi^{\mathcal{A}}$ is a correct abstraction of $\mathcal{S}$.

- **Initiality:** Every initial state of $\mathcal{S}$ should be covered by some initial node of the diagram, that is,
$$\Theta \rightarrow \mu(I)\,,$$
where I is the set of initial nodes of Ψ. This implies that every run of $\mathcal{S}$ can start at some initial node of Ψ, and corresponds to the initiality condition of Theorem 3.3.3.

- **Consecution:** For every node n and every transition τ, some successor node of n can be reached by τ (if τ can be taken), that is,
$$\mu(n) \wedge \tau \ \rightarrow \ \mu'(succ(n)) \ .$$
This corresponds to Condition 2 of Theorem 3.3.3.

Together, these two conditions imply that every run of $\mathcal{S}$ can remain within Ψ:

Proposition 5.3.2 *If a diagram Ψ satisfies the initiality and consecution requirement for a system $\mathcal{S}$, then the runs of $\mathcal{S}$ are a subset of the runs of Ψ.*

Therefore, once the above verification conditions are proved we can conclude that any safety property of Ψ also holds for $\mathcal{S}$. We will see that this proposition is equivalent to Theorem 3.3.3.

Progress: To preserve progress properties, a second set of verification conditions ensures that every computation of the system can follow an accepting path in the diagram, that is, can always eventually remain in an accepting SCS. Thus, if S is not an accepting SCS, we must show that system computations can always leave S, or cannot stay in S forever.

Definition 5.3.3 *For an SCS S, a $tail(S)$*-computation *is a system computation that has a corresponding path π in the diagram such that $inf(\pi) = S$. An SCS is called* transient *if every $tail(S)$-computation can leave S (so it is also a $tail(S')$-computation, for an SCS $S' \neq S$).*

We want to show that each non-accepting SCS is transient. This can be done in one of the following three ways:

- **Just exit:** An SCS S has a *just exit*, if there is a just transition τ such that the following verification conditions are valid: for *every* node $m \in S$,

  $$\mu(m) \rightarrow enabled(\tau) \quad \text{and} \quad \mu(m) \wedge \tau \rightarrow \mu'(succ(m) - S) \ .$$

  This means that τ is enabled and can leave the SCS at all nodes. We say that τ is the *just exit transition* for S. Note that since transitions can be non-deterministic, system computations may be able to stay within S as well. The just exit simply guarantees that all computations *may* go out. (That is, runs of $\mathcal{S}$ that stay within the SCS are not necessarily unfair.)

- **Compassionate exit:** An SCS S has a *compassionate exit* if there is a compassionate transition τ such that the following verification conditions are valid:

  For *every* node $m \in S$,

  $$\mu(m) \rightarrow \neg enabled(\tau) \quad \text{or} \quad \mu(m) \wedge \tau \rightarrow \mu'(succ(m) - S) \,,$$

  and for *some* node $n \in S$, transition τ is enabled at n:

  $$\mu(n) \rightarrow enabled(\tau) \ .$$

  This means that at every node in S, either τ is disabled or τ can lead out of S, and there is at least one node n where τ can indeed leave S. We say that n is the *exit node* and τ is the *compassionate exit transition* for S.

- **Well-founded SCS:** A *cut-set* of an SCS S is a set of edges E such that every loop in S contains some edge in E (so removing E will disconnect S). An SCS $S : \{n_1, \ldots, n_k\}$ is *well-founded* if the nodes of S can be labeled with ranking functions $\{\delta_1, \ldots, \delta_k\}$ (see Definition 2.4.1) such that the following verification conditions are valid: there is a cut-set E in S such that for all edges $(n_1, n_2) \in E$ and every transition τ,

  $$\mu(n_1) \wedge \tau \wedge \mu'(n_2) \rightarrow \delta_1(\mathcal{V}) \succ \delta_2'(\mathcal{V}) \,,$$

and for all other edges $(n_1, n_2) \notin E$ in S and for all transitions τ,

$$\mu(n_1) \wedge \tau \wedge \mu'(n_2) \;\rightarrow\; \delta_1(\mathcal{V}) \succeq \delta_2'(\mathcal{V}) \; .$$

This means that there is no $tail(S)$-computation: such a computation would have to traverse at least one of the edges in E infinitely often, which contradicts the well-foundedness of the ranking functions.

We say that S has a *fair exit* if it has a just or a compassionate exit. Combined with consecution, the fair exit verification conditions ensure that a $tail(S)$-computation can always follow a path that leaves S. Any run of the system that is *forced* to stay within an SCS with a fair exit must be unfair.[1] If S is well-founded, there can be no $tail(S)$-computations at all. We now claim:

Proposition 5.3.4 *If a GVD Ψ for a system $\mathcal{S}$ satisfies the initiality and consecution requirements, and all non-accepting SCS's have a fair exit or are well-founded, then $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$, that is, Ψ is $\mathcal{S}$-valid.*

To show that a given diagram Ψ is $\mathcal{S}$-valid, the user must prove initiality and consecution, and specify, for each non-accepting SCS, one of the following:

1. a just exit transition τ;
2. a compassionate exit transition τ and its exit node n; or
3. well-founded ranking functions δ_i and a cut-set E that prove that the SCS is well-founded.

5.3.2 Diagrams as Abstract Systems

We now formalize how an $\mathcal{S}$-valid GVD Ψ can be seen as a property-preserving abstraction $\Psi^{\mathcal{A}}$ of $\mathcal{S}$, equipped with well-founded orders that allow more progress properties to be proved for it.

We first note that both the well-founded orders and the considerations on enabling conditions are *local* to the diagram: a fair transition only needs to be shown enabled or

[1] Since transitions are possibly non-deterministic, we cannot claim that every run that stays within one of these SCS must be unfair.

disabled at particular nodes, rather than at all states. In this sense, the use of GVD's corresponds more to an application of the general simulation rule of Theorem 3.6.1 than of Theorem 3.4.1, which assumes a simple transition mapping.

In terms of abstraction, we can understand this locality by adding *control locations* to the abstract system that correspond to the nodes in the diagram. Intuitively, we need control only because we expect correctly approximated enabling conditions. $\mathcal{S}$ will now be a *refinement* of $\Psi^{\mathcal{A}}$ [KMP94]. The control locations of $\mathcal{A}$ are given by an *unobservable* abstract control variable, as defined in Section 3.7, which the refined $\mathcal{S}$ is not required to include. In Chapter 6, we will present an alternative representation of the abstract system defined by a GVD that does not require this control variable.

Given an $\mathcal{S}$-valid diagram Ψ, we now construct an abstraction $\Psi^{\mathcal{A}} : \langle B, \Theta_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}}\rangle$ that satisfies the conditions of the general simulation rule of Theorem 3.6.1:

1. The assertion basis B is the set of all atomic assertions that appear in Ψ. The abstract system $\Psi^{\mathcal{A}}$ contains a new unobservable control variable π, ranging over the finite set of diagram nodes $N : \{n_1, \ldots, n_k\}$.

   To be precise, we use the notation $\mu^{\mathcal{A}}(n)$ to refer to the *abstract* assertion that corresponds to the label of n (see Proposition 3.8.4). For a set of nodes S, now $\mu^{\mathcal{A}}(S) \stackrel{\text{def}}{=} \bigvee^{\mathcal{A}}_{n \in S} \mu^{\mathcal{A}}(n)$.

2. The abstract initial condition is

$$\Theta_{\mathcal{A}} : \bigvee\nolimits^{\mathcal{A}}_{n \in I} \left(\mu^{\mathcal{A}}(n) \ \wedge^{\mathcal{A}} \ \pi = n\right) ,$$

   where I is the set of initial nodes in Ψ. Thus, the diagram initiality condition implies the initiality clause of Theorem 3.6.1.

3. The consecution verification conditions associated with the diagram define abstract transitions that satisfy the consecution clause of Theorem 3.6.1. If the verification condition

$$\mu(n) \wedge \tau \ \rightarrow \ \mu'(succ(n))$$

   is proved for Ψ, then $\mathcal{T}_{\mathcal{A}}$ contains an abstract transition $\tau^{\mathcal{A}}_n$, with no fairness, with transition relation

$$\tau^{\mathcal{A}}_n \ : \ \left(\mu^{\mathcal{A}}(n) \ \wedge^{\mathcal{A}} \ \pi = n\right) \ \rightarrow^{\mathcal{A}} \ \bigvee\nolimits^{\mathcal{A}}_{m \in succ(n)} (\mu^{\mathcal{A}}(m)' \ \wedge^{\mathcal{A}} \ \pi' = m) \ .$$

4. Each SCS S with a fair exit defines a correspondingly fair transition $\tau_S^{\mathcal{A}}$ as follows:

   (a) If S has a just exit transition τ, then a just abstract transition $\tau_S^{\mathcal{A}}$ has transition relation

   $$\tau_S^{\mathcal{A}} : \bigvee\nolimits_{n\in S}^{\mathcal{A}} \left[\left(\mu^{\mathcal{A}}(n) \wedge^{\mathcal{A}} \pi = n\right) \wedge^{\mathcal{A}} \bigvee\nolimits_{m\in succ(n)-S}^{\mathcal{A}} \left(\mu^{\mathcal{A}}(m)' \wedge^{\mathcal{A}} \pi' = m\right) \right] .$$

   This means that $\tau_S^{\mathcal{A}}$ is enabled at all the nodes in the SCS, and when taken will lead to one of the successor nodes outside S. Note that now computations of the abstract system *must* leave the SCS, while it may have been possible for computations of $\mathcal{S}$ to stay within it.

   (b) If S has a compassionate exit transition τ, let E be the nodes where τ, if taken, leads out of the SCS, and n_0 be the node where τ is known to be enabled (which is also an element of E). Then a compassionate $\tau_S^{\mathcal{A}}$ has transition relation

   $$\tau_S^{\mathcal{A}} : \begin{pmatrix} \left[\bigvee\nolimits_{n\in E}^{\mathcal{A}} (\pi = n)\right] \wedge^{\mathcal{A}} \\ \bigvee\nolimits_{n\in E}^{\mathcal{A}} \left[\left(\mu^{\mathcal{A}}(n) \wedge^{\mathcal{A}} \pi = n\right) \rightarrow^{\mathcal{A}} \bigvee\nolimits_{m\in succ(n)-S}^{\mathcal{A}} \left(\mu^{\mathcal{A}}(m)' \wedge^{\mathcal{A}} \pi' = m\right) \right] \\ \bigvee^{\mathcal{A}} \left[\left(\mu^{\mathcal{A}}(n_0) \wedge^{\mathcal{A}} \pi = n_0\right) \wedge^{\mathcal{A}} \bigvee\nolimits_{m\in succ(n_0)-S}^{\mathcal{A}} \left(\mu^{\mathcal{A}}(m)' \wedge^{\mathcal{A}} \pi' = m\right) \right] \end{pmatrix}$$

   This states that $\tau_s^{\mathcal{A}}$ can only be enabled at the nodes in E, must leave the SCS if taken at any node in E, and can always be taken at n_0 to leave the SCS. Note that τ is known to be disabled at all nodes in $S - E$.

   When τ is fair, an unfair abstract version $\tau^{\mathcal{A}}$ of τ accounts for the nodes where τ is only expected to satisfy the consecution requirement.

Remark 5.3.5 (Control variable π) We can collect the transitions with no fairness requirements into a single unfair transition. Similarly, we can collect all the abstract just transitions that correspond to the same just concrete transition τ into a single just abstract transition. If there are no compassionate transitions, we can ignore the unobservable control variable π: the enabling condition of the abstract just transitions will under-approximate that of the original concrete transitions, so every abstract run that is unjust towards $\tau^{\mathcal{A}}$ will be unjust towards τ as well (see Theorem 3.4.1 and its proof).

Notice that the abstract system that has a single just abstract transition for each just concrete one will, in general, satisfy more properties than the one that has a separate just abstract transition for each SCS. Removing the control variable can also increase the number of properties that the abstraction satisfies.

However, π is essential when compassionate transitions are present. If it were removed, we could not ensure that every abstract run that is uncompassionate towards $\tau^{\mathcal{A}}$ is, when concretized, also uncompassionate towards τ, unless we had exact information about the enabled status of τ at all the nodes in the diagram: the problem described in Remark 3.4.2 and Figure 3.4 could occur. The control variable π lets us focus our attention on particular SCS's for which the approximation of the enabling condition is exact, and safely ignore the rest of the abstract state space. $\square$

Proposition 5.3.6 *For an abstract system $\Psi^{\mathcal{A}}$ associated with a GVD Ψ,*

$$\mathcal{L}(\Psi) \subseteq \gamma(\mathcal{L}(\Psi^{\mathcal{A}}))$$

where γ accounts for the unobservable control location (see Definition 3.7.2). Thus, if Ψ is $\mathcal{S}$-valid, then $\Psi^{\mathcal{A}}$ is an LTL-preserving abstraction of $\mathcal{S}$ (c.f. Proposition 3.3.2).

The equality $\mathcal{L}(\Psi) = \gamma(\mathcal{L}(\Psi^{\mathcal{A}})$ does not hold in general, since the diagram may include extra acceptance conditions justified by well-founded orders, which are not reflected in $\Psi^{\mathcal{A}}$. If the $\mathcal{S}$-validity of Ψ does not depend on well-founded orders, then

$$\mathcal{L}(\Psi) = \gamma(\mathcal{L}(\Psi^{\mathcal{A}})) \ .$$

Proof: Consider a computation $\pi : s_0, s_1, \ldots,$ in $\mathcal{L}(\Psi)$, with associated accepting path $n_0, n_1, \ldots$ through Ψ. Define $a_i \stackrel{\text{def}}{=} \alpha(s_i) \wedge \pi = n_i$. Then $\pi_{\mathcal{A}} : a_0, a_1, \ldots$ is clearly a run of $\Psi^{\mathcal{A}}$, and $\pi \in \gamma(\pi_{\mathcal{A}})$.

We now show that $\pi_{\mathcal{A}}$ must be fair: if not, then there is some just (resp. compassionate) transition $\tau_S^{\mathcal{A}}$, corresponding to a non-accepting SCS S in Ψ, that is continuously (resp. infinitely often) enabled but not taken. This would mean that $n_0, n_1, \ldots$ in Ψ remains within the non-accepting S, a contradiction.

If the acceptance conditions of Ψ are justified without using well-founded orders, the reverse is also true: if a run $\pi : s_0, s_1, \ldots$ of Ψ is not accepting (an thus not in $\mathcal{L}(\Psi)$), then it is forced to eventually reside within a non-accepting SCS S of Ψ. Since Ψ is $\mathcal{S}$-valid, the SCS S has a fair exit. Therefore, any corresponding run

$\pi_{\mathcal{A}} : a_0, a_1, \ldots$ of $\Psi^{\mathcal{A}}$ is unfair towards the corresponding fair transition $\tau_S^{\mathcal{A}}$, and thus $\mathcal{L}(\Psi) = \mathcal{L}(\Psi^{\mathcal{A}})$. □

That $\Psi^{\mathcal{A}}$ is a correct abstraction of $\mathcal{S}$ can also be justified using the general simulation rule of Theorem 3.6.1 (Figure 3.5).

If acceptance conditions of a diagram Ψ are justified using well-founded orders, Ψ may prove more progress properties of $\mathcal{S}$ than $\Psi^{\mathcal{A}}$ can. As we will see in Chapter 6, this can be overcome if the model checker for $\Psi^{\mathcal{A}}$ is told about the well-founded orders, which indicate that certain cycles in the abstract state-space cannot be followed indefinitely.

5.3.3 (Ψ, φ) Property Satisfaction: Concretization

Section 5.3.1 describes verification conditions that prove that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$, that is, that the diagram defines a correct abstraction of the system. To prove the $\mathcal{S}$-validity of a property φ, it remains to show that

$$\mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi) \ .$$

This corresponds to checking that the diagram, when seen as an abstraction, satisfies a property $\varphi^{\mathcal{A}}$ such that $\gamma(\varphi^{\mathcal{A}})$ implies φ.

This check can be performed using standard ω-automata model checking, if we can relate the nodes of the diagram with the atomic assertions in φ. In both [BMS95] and [MBSU98], this is done using a *propositional labeling*, where a diagram node n is labeled with a boolean combination b of atomic assertions of φ, provided we prove the validity of

$$\mu(n) \rightarrow b \ .$$

We can think of this in two ways: in the first, the diagram and its propositional labeling define a new finite-state ω-automaton $\Psi^{\mathcal{B}}$, where each node is a state of the automaton, labeled with the given propositions. This automata is an abstraction of $\mathcal{S}$ over a basis that includes all the atoms of φ. The property itself can be seen as an abstract property $\varphi^{\mathcal{B}}$ over its propositions, which can be model checked over $\Psi^{\mathcal{B}}$, and thus $\mathcal{S} \models \varphi$.

In a second point of view, the original node labeling of the diagram defines an abstraction $\Psi^{\mathcal{A}}$ over the atomic propositions that appear in the node labels, independently of φ, proving a property $\chi^{\mathcal{A}}$ that is not necessarily equivalent to $\alpha^t(\varphi)$. The propositional labeling, and the ω-automata check, then ensure that $\gamma(\chi^{\mathcal{A}})$ implies φ.

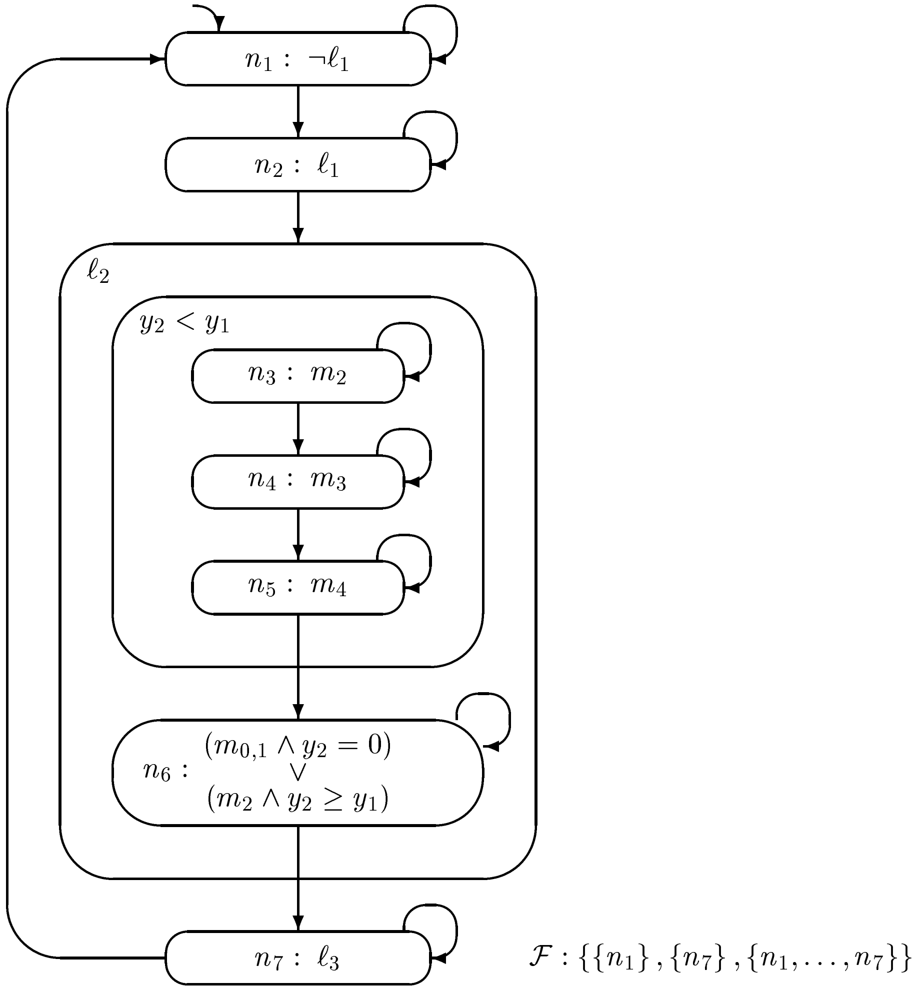

Figure 5.3: GVD for proving BAKERY accessibility $\Box(\ell_1 \rightarrow \Diamond\, \ell_3)$

Since our goal is to reuse abstractions, we prefer the second view. Once a diagram Ψ is proved to be a correct abstraction, that is, $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi) = \gamma(\mathcal{L}(\Psi^{\mathcal{A}}))$, any property that can be model checked for $\Psi^{\mathcal{A}}$ can be transferred over to $\mathcal{S}$.

5.3.4 Example

Example 5.3.7 (BAKERY GVD) *The GVD of Figure 5.3 proves accessibility,*

$$\varphi : \Box(\ell_1 \rightarrow \Diamond\, \ell_3) \ ,$$

for BAKERY *(Figure 2.1). For succinctness, this diagram uses* encapsulation conventions *[MP94] based on those of Statecharts [Har87], where edges entering or leaving a* compound node *are interpreted as entering or leaving all of its subnodes. The assertion that labels a compound node is added as a conjunct to the label of each of its subnodes.*

The acceptance condition $\mathcal{F} : \{\{n_1\}, \{n_7\}, \{n_1, \ldots, n_7\}\}$ *means that diagram computations can remain indefinitely at* n_1 *or, if they leave* n_1 *to reach* ℓ_1 *at* n_2*, must move down the diagram to reach* ℓ_3*. Thus all computations of the diagram satisfy* $\varphi^{\mathcal{A}}$ *(which in this case is identical to* φ *since it only references control variables).*

This diagram has seven SCSs: $\{n_1\} \ldots \{n_6\}$*, and* $\{n_1, \ldots, n_7\}$*. It is easy to show that the SCSs not in* $\mathcal{F}$ *have a just exit. For example,* m_2 *is the just exit transition for* $\{n_3\}$*, since it is always enabled at that node and leads out of it. Similarly,* $\{n_2\}$*,* $\{n_4\}$*,* $\{n_5\}$*, and* $\{n_6\}$ *have just exit transitions* ℓ_1*,* m_3*,* m_4*, and* ℓ_2*.*

Since no well-founded orders are needed to justify the acceptance conditions of the diagram, $\mathcal{L}(\Psi) = \gamma(\mathcal{L}(\Psi^{\mathcal{A}}))$*. Thus,* $\Psi^{\mathcal{A}}$ *is a correct abstraction of* $\mathcal{S}$*, so* $\Psi^{\mathcal{A}} \models \varphi^{\mathcal{A}}$ *implies that* $\mathcal{S} \models \varphi$*. Notice that the assertions on this diagram correspond to the bits* $\{b^{y_1=0}, b^{y_2=0}, b^{y_1 \leq y_2}\}$ *and the control variables of the abstract* BAKERY *of Figure 3.2, that is,* $\Psi^{\mathcal{A}}$ *is an abstraction over the same basis.*

See [MBSU98] for an example that uses just and compassionate exits, and two different well-founded orders.

5.4 Verification Rules: Abstract System Construction

Understanding standard verification diagrams and rules as special cases of GVD's, it is not difficult to see how the considerations in the previous section can be applied to them. Figure 5.4 presents the *chain rule* for proving properties of the form $\square(p \rightarrow \diamondsuit\, q)$ [MP93]. The abstraction that is implicitly generated by the rule is:

$$\mathcal{S}^{\mathcal{A}} : \left\langle \mathcal{V} : \langle b^{\varphi_0}, \ldots, b^{\varphi_m} \rangle, \Theta_{\mathcal{A}} : \mathit{true}^{\mathcal{A}}, \mathcal{T}_{\mathcal{A}} \right\rangle$$

Premises J1, J2 and J3 construct an abstraction for which

$$\varphi^{\mathcal{A}} : \square \left(\left(\bigvee_{j=0}^{m} {}^{\mathcal{A}} \right) b^{\varphi_j} \;\rightarrow^{\mathcal{A}}\; \diamondsuit\, b^{\varphi_0} \right)$$

For assertions $p, q, \varphi_0, \ldots, \varphi_m$,
Just transitions $\tau_1, \ldots, \tau_m$,

J1.	$\{\varphi_i\}\, \tau \left\{ \bigvee_{j \le i} \varphi_j \right\}$ for all $\tau \in \mathcal{T}$		*consecution*
J2.	$\{\varphi_i\}\ \tau_i \left\{ \bigvee_{j < i} \varphi_j \right\}$, for $i = 1, \ldots, m$		*fairness I*
J3.	$\varphi_i \rightarrow \mathit{enabled}(\tau_i)$		*fairness II*
J4.	$p \rightarrow \bigvee_{j=0}^{m} \varphi_j$ and $\varphi_0 \rightarrow q$		*concretization*

$\mathcal{S} \models\ p \Rightarrow \diamondsuit\, q$

Figure 5.4: Rule CHAIN

can be model checked. Each just transition τ_i is abstracted to a just abstract transition

$$\tau_i^{\mathcal{A}} : b^{\varphi_i}\ \wedge^{\mathcal{A}}\ \bigvee\nolimits_{j<i}^{\mathcal{A}} b^{\varphi_j}{}'$$

while all other transitions can be abstracted to

$$\tau^{\mathcal{A}}\ :\ \bigwedge\nolimits_i^{\mathcal{A}} \left(b^{\varphi_i}\ \rightarrow^{\mathcal{A}}\ \bigvee\nolimits_{j \le i}^{\mathcal{A}} b^{\varphi_j}{}' \right)\ .$$

Premise J4 ensures that $\gamma(\varphi^{\mathcal{A}})$ implies $\varphi : \square(p \rightarrow \diamondsuit\, q)$.

The intermediate assertions in the rule correspond to the nodes in a CHAIN verification diagram [MP94], which have the same general form as the GVD shown in Figure 5.3. The use of an unobservable control variable π is not required in this case, since only justice is involved (see Remark 5.3.5). Like diagrams, verification rules that rely on compassion will require such added control.

5.5 Dynamic Abstraction: Deductive Model Checking

Deductive model checking [SUM99] is a method for the interactive model checking of possibly infinite-state systems, generalizing the classic model checking procedure outlined in Section 2.3.1. The procedure itself is presented in detail in [SUM99, Sip98], so here we focus on those aspects relevant to the abstraction point of view espoused in this thesis.

Rather than explicitly building the $(\mathcal{S}, \neg\varphi)$ product graph as described in Section 2.3.1, DMC starts with a general skeleton of the product and progressively refines it. This graph is called the *falsification diagram* for $\mathcal{S}$ and φ. As their name suggests, falsification diagrams are dual to the Generalized Verification Diagrams of Section 5.3. The acceptance condition of a falsification diagram is derived from the tableau atoms for $\neg\varphi$. Instead of showing that every computation of $\mathcal{S}$ *can* stay within an accepting SCS, as in the GVD case, we now show that every computation of $\mathcal{S}$ *must* end in a non-accepting SCS.

Figure 5.5 shows an outline of the Deductive Model Checking (DMC) procedure. The procedure repeatedly applies one of a set of transformations to the falsification diagram, until a counterexample computation is found or it is clear that no such computation can exist. At any given point, the falsification diagram represents all the computations of the system that may possibly violate the temporal property. The procedure can also produce a counterexample, as we will see in Section 5.5.3.

5.5.1 DMC as Abstraction Refinement

By *abstraction refinement*, we mean the process of improving the approximations represented by an abstract system. This includes making $\exists\exists$ transitions smaller and $\forall\exists$ ones larger, while retaining the soundness of the approximation. Refinement also includes obtaining better bounds on enabling conditions and adding new well-founded orders.

Deductive Model Checking can be understood as the process of *refining* an abstraction of $\mathcal{S}$, while simultaneously model checking it. The DMC transformations may be classified into two groups: the first performs model checking tasks, discarding the portions of the product graph that are not relevant to the property being proved. The second performs abstraction refinement, justified by verification conditions.

Figure 5.6 presents this alternative view of DMC. After each refinement step we have three mutually exclusive possibilities:

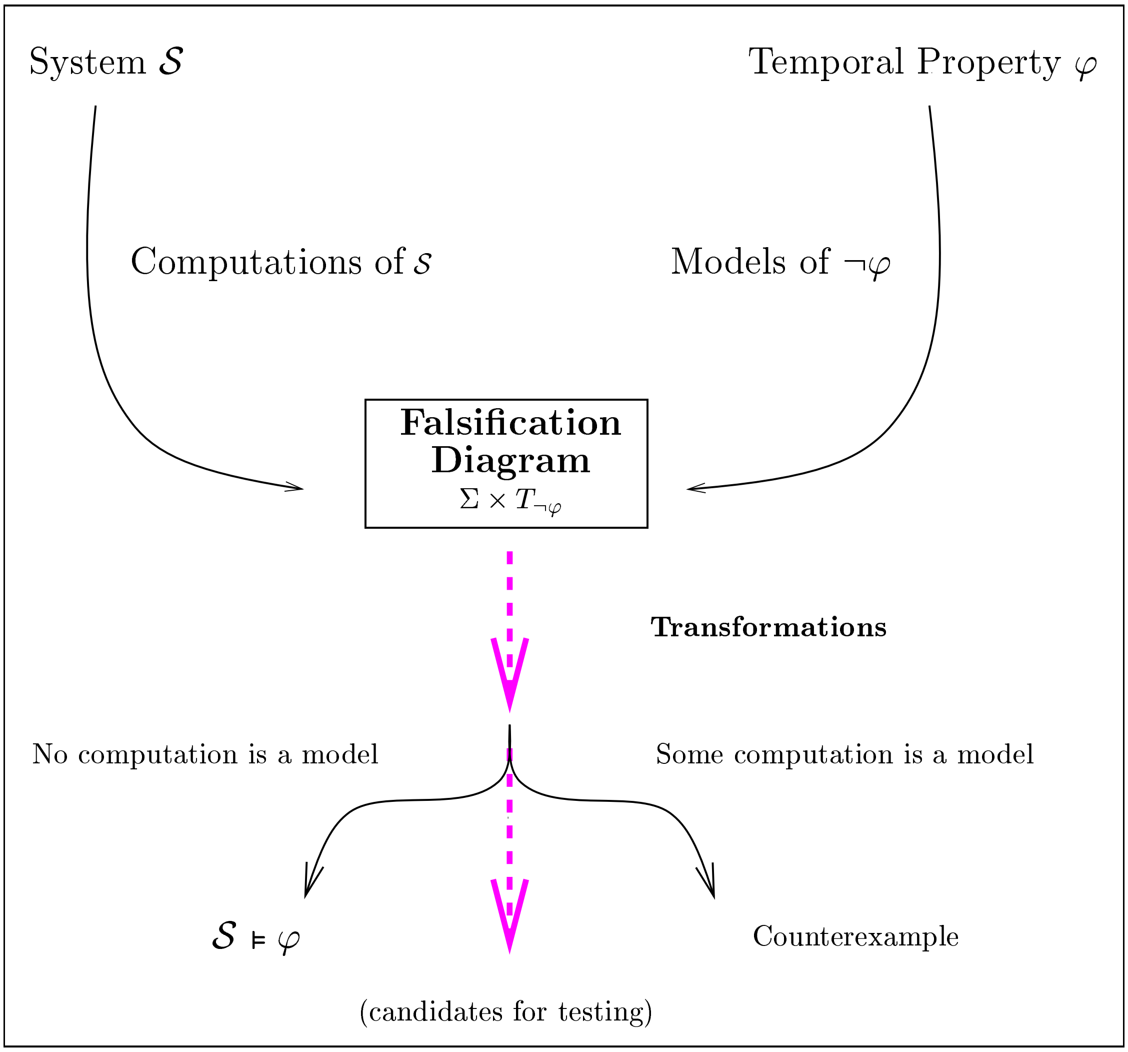

Figure 5.5: Outline of Deductive Model Checking (DMC)

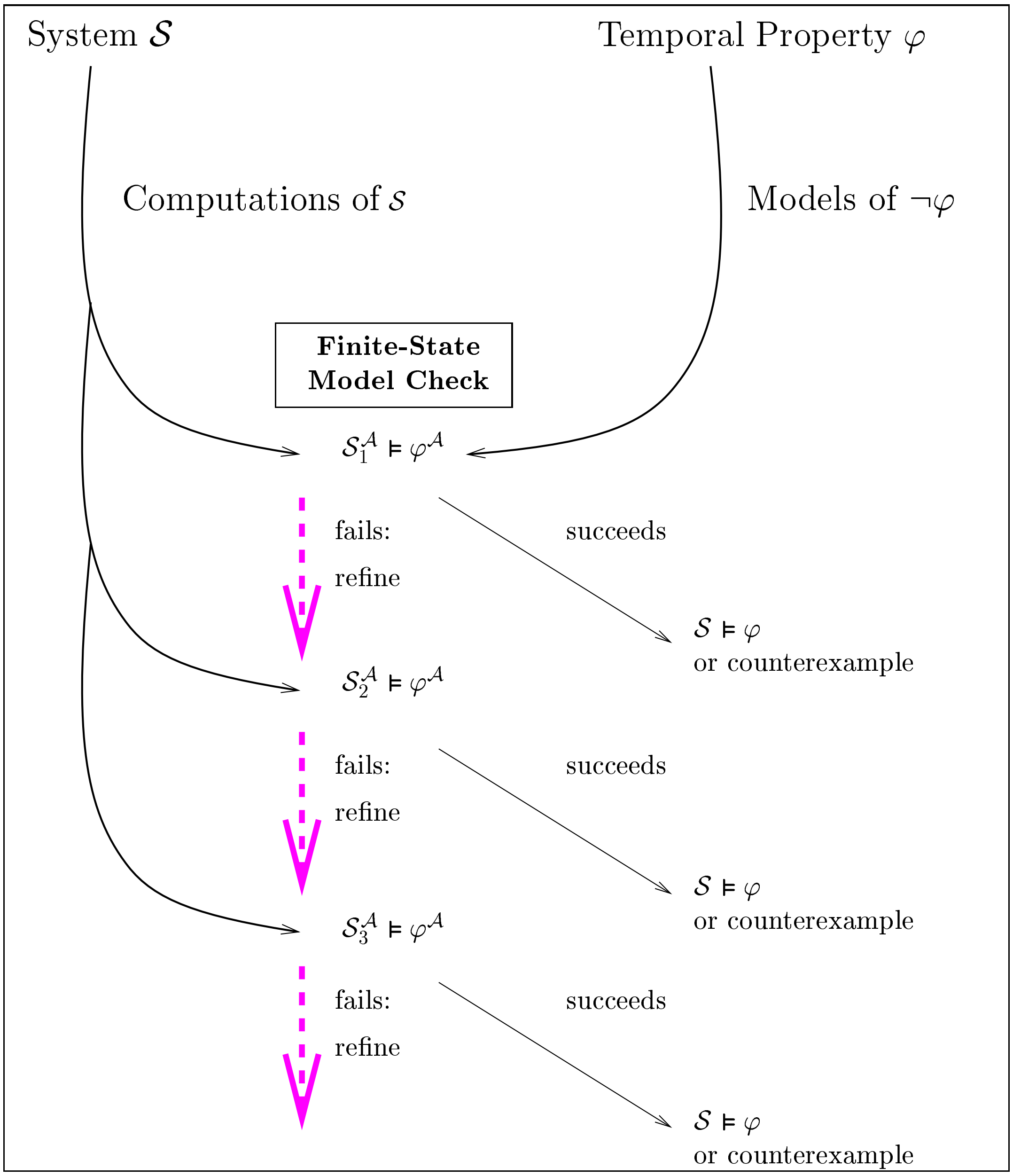

Figure 5.6: Deductive Model Checking viewed as abstraction refinement

1. The abstract system can be successfully model checked, that is, $\mathcal{S}^{\mathcal{A}} \vDash \varphi^{\mathcal{A}}$, where $\gamma(\varphi^{\mathcal{A}})$ in this case is equivalent to φ. Since $\mathcal{S}^{\mathcal{A}}$ is a correct abstraction of $\mathcal{S}$, this means that $\mathcal{S} \vDash \varphi$.

2. The abstract model checker can find, in the constrained $\forall\exists$ abstract relation for $\mathcal{S}^{\mathcal{A}}$, a counterexample to φ, in which case we know that $\mathcal{S} \nvDash \varphi$.

3. The abstract model checker can only determine that $\mathcal{S}^{\mathcal{A}} \nvDash \varphi^{\mathcal{A}}$, finding an abstract counterexample, but cannot determine if a concrete counterexample exists. Thus, it may be the case that $\mathcal{S} \vDash \varphi$, but $\mathcal{S}^{\mathcal{A}}$ is too abstract to prove or disprove it. In this case, the abstraction must be *refined*.

A model checker specially tailored for this task is presented in Chapter 6. A practical consequence is that a procedure similar to DMC can be implemented using a theorem prover and a model checker as black boxes, provided that the model checker gives feedback sufficient to select the next refinement transformation. If DMC succeeds, then a corresponding abstraction exists that can be model checked by the method of Section 6.4. (Since DMC is complete, the converse is also true.)

Similarly, GVD's can also be incrementally constructed with the help of model checking tools. In particular, the user need not specify all the well-founded orders, acceptance conditions and SCS's in advance: a model checker should help identify them when necessary. Furthermore, the SCS's that provide guidance in DMC can be obtained from a model checker, as we will see in Section 6.4.

Remark 5.5.1 We should note that DMC provides an interactive, goal-oriented way of finding a suitable abstraction. Furthermore, it can selectively refine only those transitions and abstract states that are relevant to the property being proven. Thus, it can offer significant savings even for the purely finite-state case. The abstraction is computed "on-the-fly," so that not even the entire *abstract* state-space has to be expanded. We can retain this advantage by applying DMC to the finite-state abstraction—see Section 6.4. □

The initial abstraction: The initial falsification diagram is the product of the $\neg\varphi$ tableau and the abstract system

$$\mathcal{S}_1^{\mathcal{A}} : \left\langle \left\{ b_1 : true^{\mathcal{A}} \right\},\ \theta : b_1,\ \mathcal{T} \right\rangle ,$$

where each concrete transition τ_i is approximated by $\tau_i^{\mathcal{A}} : true^{\mathcal{A}}$. This is the coarsest abstraction of $\mathcal{S}$, with basis $B : \left\{b_1 : true^{\mathcal{A}}\right\}$, and only tautological (generally valid) temporal properties will hold for it. The initial $\exists\exists$-approximations $\tau_{\forall\exists i}^{\mathcal{A}}$ are $false^{\mathcal{A}}$.

In DMC, edges in the product graph are labeled by sets of transitions (see Remark 2.1.4). Initially, every edge is labeled by the set of all transitions.

Model Checking

- **(Model checking transformations)**: The DMC transformations that remove empty edges, unsatisfiable and unreachable nodes, and nodes with no successors, correspond to model checking the current abstraction.

  Note that in the abstraction framework, the only unsatisfiable nodes removed are those that are propositionally unsatisfiable. In DMC, the assertions from the tableau nodes and the states are combined and simplified. This corresponds to using theorem proving to decide relationships between the elements of the basis, ruling out unsatisfiable combinations.

To perform these steps, DMC identifies unfulfilling SCS's, according to the underlying tableau, and *unfair* and *terminating* SCS's, according to what is known about the transitions, as we will see below in Section 5.5.2. The procedure also keeps track of new SCS's as old ones are disconnected. In Chapter 6, a finite-state model checking algorithm will perform the equivalent tasks.

Refinement: More Specific Transitions

We now consider two ways in which an abstraction can be refined. First, more information can be obtained about particular transitions by proving verification conditions that had not been considered previously, given a fixed basis. In the second, new assertions are added to the basis. DMC provides transformations analogous to each of these:

Note 5.5.2 (DMC Nodes) *A DMC node n is of the form $n : \langle A, f\rangle$, where A is a atom from the temporal tableau of $\neg\varphi$, used to define the acceptance conditions (*fulfilling *SCS's are accepting), and f is an assertion. As before, we define the label of the node as the assertion that corresponds to it: $\mu(n) \stackrel{\text{def}}{=} f$.*

DMC uses edge labels to keep track of which transitions *could* be taken at an edge. Transitions are removed from a label when we show that they cannot be taken at that edge:

- **(remove edge label)**: If an edge from (A_1, f_1) to (A_2, f_2) is labeled with a transition τ, and the assertion
$$f_1(\mathcal{V}) \wedge f_2(\mathcal{V}') \wedge \tau(\mathcal{V}, \mathcal{V}')$$
is unsatisfiable, remove τ from the edge label.

  Edges with an empty label, and unreachable nodes, are removed.

This is equivalent to the *elimination method* of [BLO98] (see Section 4.10) and has the effect of conjoining $f_1 \rightarrow^{\mathcal{A}} \neg^{\mathcal{A}} f_2'$ to the transition relation of $\tau^{\mathcal{A}}$. However, in DMC the refinement is local to a given edge: the transformation does not affect other edges where τ appears. We can optimize DMC by sharing the new information learned about $\tau^{\mathcal{A}}$ throughout the falsification diagram. This has the effect of caching the proved verification conditions, reusing them whenever possible. We return to this in Section 6.4.

Refinement: Finer Abstract Domain

The second way to refine an abstract system is to choose a finer abstract domain, by introducing new basis elements. *Node splitting* transformations can be understood as doing precisely this:

- **(binary split)**: Replace a node (A, f) by the two nodes
$$N_1 : (A, f \wedge \chi) \quad \text{and} \quad N_2 : (A, f \wedge \neg\chi)) \ .$$

- **(n-ary split)**: Replace a node (A, f) by the $j + 1$ nodes
$$\begin{array}{rcl} N_1 & : & (A, f \wedge p_1) \\ & \vdots & \\ N_j & : & (A, f \wedge p_j) \\ N_{j+1} & : & (A, f \wedge \neg(p_1 \vee \ldots \vee p_j)) \end{array}$$

Removing edges after a split is equivalent to refining abstract transitions under a new extended basis. As described in Section 4.7.3, computing the abstraction that results from adding new basis elements or test points can be done without repeating the previous work.

5.5.2 DMC, Fairness and Well-founded Orders

In DMC parlance, a transition τ is *fully enabled* at a node n if $\mu(n) \subseteq \mathit{enabled}(\tau)$. We know that a transition cannot be taken at an SCS S if it has been removed from all the edge labels in S. An SCS is then considered unfair if some just (resp. compassionate) transition is fully enabled at all (resp. some) nodes in S and is missing from all the edges in S.

Similarly to GVD's, DMC discards SCS's according to the following criteria:[2]

- They are not accepting, with respect to the acceptance conditions of the original tableau $T_{\neg\varphi}$ (dual to ignoring accepting SCS's in GVD's).
- They are unfair, in the sense that no run can stay in the SCS without being unfair (dual to fair exit in GVD's). This requires proving $\mu(n) \rightarrow \mathit{enabled}(\tau)$ for a τ that does not label outgoing edges from n.
- They have a terminating edge (see below) that cannot be traversed infinitely many times (dual to well-founded SCS's).

Also similarly to GVD's, Deductive Model Checking assigns ranking functions to the nodes of an SCS, to show that a computation cannot forever reside within the SCS, traversing all its edges and visiting all its nodes:

Definition 5.5.3 (Terminating edge) *An edge e_t in an SCS $S : \{n_1, \ldots, n_k\}$ is* terminating *if there are ranking functions $\{\delta_1, \ldots, \delta_k\}$ mapping states into a well-founded domain $\mathcal{D}$ such that:*

1. for every edge $e : (n_1, n_2)$ in S and every $\tau \in e$

$$\mu(n_1) \wedge \tau \wedge \mu'(n_2) \rightarrow \delta_1(\mathcal{V}) \succeq \delta_2(\mathcal{V}')$$

and

2. for every $\tau \in e_t$,

$$\mu(n_1) \wedge \tau \wedge \mu'(n_2) \rightarrow \delta_1(\mathcal{V}) \succ \delta_2(\mathcal{V}') \ .$$

[2]The author apologizes for the profusion of 3-letter acronyms in this and a few other sentences.

We say that an SCS S for which a $tail(S)$-computation cannot exist is *terminating*.[3] The DMC procedure proves the termination of SCS's by identifying terminating edges. Clearly, if e_t is terminating in S, then no $tail(S)$-computation can traverse e_t infinitely many times. Such edges are removed from consideration, yielding smaller SCS's, until all fulfilling SCS's can be shown to be unfair, unreachable or terminating.

In some cases, the ranking function for a node depends on the route taken to reach that node. An *unfolding* transformation makes copies of SCS nodes and allows proving that an SCS cannot support a computation even when suitable ranking functions cannot be found for the original SCS. We will use a simple version of unfolding in Chapter 6, so we now define a version of it adapted to our needs:

Definition 5.5.4 (Unfolding an SCS) *For an SCS S with k nodes $\{n_1, \ldots, n_k\}$, the result of* unfolding *S is a new graph $\mathcal{U}(S)$ obtained as follows:*

1. *Make k copies $S_1, \ldots, S_k$ of S, retaining the same node and edge labels.*
2. *Remove node n_i from copy S_i. Redirect all incoming edges to this node to reach the copy of node n_i in $S_{i\oplus 1}$, where $i \oplus 1 = i + 1$ modulo k.*

With this machinery in place, DMC is relatively complete, as proved in [SUM99, Sip98]. As usual, this assumes that the assertion language is expressive enough and that a complete proof procedure for the required verification conditions is available.

Concretization: DMC does not require an explicit concretization step, since the assertions in $\neg\varphi$ are built into the initial falsification diagram. Thus, the abstract property $\varphi^{\mathcal{A}}$ corresponds exactly to the original concrete φ, that is, $\gamma(\varphi^{\mathcal{A}}) = \varphi$.

5.5.3 Counterexamples in DMC

So far, we have only used the (standard) $\exists\exists$-approximation. Deductive Model Checking can also be used to find counterexamples, by collecting additional information that corresponds to the generation of an abstraction that also preserves existential properties (see Section 3.5). This is done by marking edges as *executable*, which corresponds exactly to the constrained ($\forall\exists$) transition relation of Section 4.9:

[3] This definition is very similar to the *well-founded* SCS's of GVD's (Section 5.3.1) but since the requirements and context differ slightly in each case, we prefer to use different terms.

- **(executable transition in DMC).** Given an edge (n_1, n_2) labeled with transition τ, mark τ as *executable* if the following formula is valid:[4]

$$\mu(n_1) \;\rightarrow\; \exists \mathcal{V}'.\, (\tau(\mathcal{V}, \mathcal{V}') \wedge \mu'(n_2)) \quad .$$

This has the effect of adding $\mu(n_1) \wedge^{\mathcal{A}} \mu'(n_2)$ as a disjunct to $\tau^{\mathcal{A}}_{\forall\exists}$. The set of executable edges in a falsification diagram describes the corresponding $\tau^{\mathcal{A}}_{\forall\exists}$ abstract transitions. Thus, we say that the $\forall\exists$-*subgraph* is the one given by the executable edges.

Dually to the "fully enabled" requirement on fair transitions for DMC and GVD's, the existence of counterexamples requires that fair transitions that are not taken be *fully disabled* at the given nodes, that is, $\mu(n) \subseteq \neg enabled(\tau)$. We can then define:

Definition 5.5.5 (Fully fair) *A transition is* fully taken *at an SCS if it is marked as executable for an edge in the SCS. An SCS S is* fully just *(resp.* fully compassionate*) if every just (resp. compassionate) transition is either fully taken in S or fully disabled at some node (resp. all nodes) in S.*

A counterexample exists if there is a reachable, fulfilling, fully fair SCS S in the $\forall\exists$-subgraph; a $tail(S)$-computation will not violate any fairness requirements. Note that all the assertions in the respective nodes should be satisfiable. This is the case if the initial condition is satisfiable and the $\forall\exists$-approximation is sound. We return to the generation of counterexamples based on $\forall\exists$-approximations in Section 6.4.

5.6 Related Work

The *possibility diagrams* of [KMP98] and the *non-Zenoness diagrams* of [Sip98, BMSU99] (see Section 2.1.4) are verification diagrams that also specify a $\forall\exists$ transition relation, allowing existential properties to be proved. Such diagrams can describe $\forall\exists$ and $\exists\exists$ abstract relations in a common diagrammatic form.

As with the generation of $\forall\exists$-approximations (Section 4.9), an important issue in practice is the strength of the validity checker and theorem proving available, and how well they handle existential quantifiers.

[4] Recall that verification conditions are implicitly universally quantified.

Other top-down methods for model checking infinite-state systems have been proposed, which are related to DMC. These are addressed in Chapter 7.

Summary: We have seen how Generalized Verification Diagrams and the final falsification diagram produced by a Deductive Model Checking proof define an abstract system, using well-founded orders and fairness considerations local to each SCS. In the following chapter we discuss how the underlying abstractions can be represented independently of any particular property to be proved, and automatically model checked. This will also allow combining them into more refined abstractions for which more temporal properties can be model checked.

Chapter 6

Combining Extended Abstractions

The previous chapter shows how deductive rules and deductive-algorithmic methods build abstractions that make fine distinctions concerning fair transitions and well-founded orders. While such abstractions must in general be found interactively, we would like to reuse them as automatically as possible once they have been constructed and their correctness is established.

Consider the scenario where the user of a verification tool such as STeP (Section 2.7) is analyzing a particular system $\mathcal{S}$. This user may prove properties $\{\varphi_1, \ldots, \varphi_n\}$ of $\mathcal{S}$ using one or more applications of abstraction generation, verification rules, GVD's, and DMC. When proving a new property φ_{n+1} of $\mathcal{S}$, it would be convenient to automatically test if φ_{n+1} can be model checked over the *combination* of the abstractions already built.

It is important to note that this is more general than simply checking the validity of

$$(\varphi_1 \wedge \ldots \wedge \varphi_n) \rightarrow \varphi_{n+1} \ ,$$

even if this check were not purely propositional. The combination we propose can uncover interactions between the generated abstractions. Each proof should facilitate the next, regardless of the method used and the type of temporal property proved. Even when a proof fails, the partially constructed abstraction can help build the next one.

Our combination method avoids directly computing the cross product of the abstractions, but instead merges the constraints on $\mathcal{S}$ implied by each one. In particular, this means that the unobservable control variables from the previous chapter, which are really an artifact of the formula being proved, may be ignored.

6.1 The Extended Input

Together with the abstract transition relations, an abstraction will now include additional information in the form of a *fairness table* and a *termination table*, which contain purely propositional (finite-state) information.

Note 6.1.1 (Transition-mapped abstractions) *Throughout this chapter, we will assume that abstractions are* transition-mapped *for fair transitions: each concrete fair transition τ_i corresponds to a single abstract transition $\tau_i^{\mathcal{A}}$.*

6.1.1 The Fairness Table

The abstraction method of Chapter 3 only lets abstracted transitions retain their fairness constraints if their enabling condition is abstracted *precisely*, following Theorem 3.4.1. This is a strong requirement, and in Chapter 5 we saw that partial information about the enabling conditions can still be useful. In general, we have upper and lower bounds, of the form:

$$\begin{array}{ll} \gamma(\mathit{enabled}^{-}(\tau)) \subseteq \mathit{enabled}(\tau) & \text{(under-approximation)} \\ \mathit{enabled}(\tau) \subseteq \gamma(\mathit{enabled}^{+}(\tau)) & \text{(over-approximation)} \end{array}$$

where $\mathit{enabled}^{-}(\tau)$ and $\mathit{enabled}^{+}(\tau)$ are abstract assertions. These correspond, of course, to proving verification conditions of the form $p \rightarrow \mathit{enabled}(\tau)$ and $q \rightarrow \neg\mathit{enabled}(\tau)$ for assertions p and q.

Such bounds can be obtained by applying the approximation procedure of Chapter 4 to concrete enabling conditions, that is, computing $\alpha(+, C, \mathit{enabled}(\tau))$ and $\alpha(-, C, \mathit{enabled}(\tau))$ or, alternatively, computing the enabling condition of an under- or over-approximated abstract transition. We collect this information in a *fairness table:*

Definition 6.1.2 (Fairness table) *A* fairness table *contains the following information for abstract transitions $\tau_1^{\mathcal{A}}, \ldots, \tau_n^{\mathcal{A}}$ that correspond to fair concrete transitions $\tau_1, \ldots, \tau_n$:*

Transition $\tau^{\mathcal{A}}$ (over-approximation of τ)	*fairness of τ (just or compassionate)*	*$\mathit{enabled}^{-}(\tau)$ (lower bound)*
$\tau_1^{\mathcal{A}}$	{`just` | `comp`}	$L_1^{\mathcal{A}}$
$\vdots$	$\vdots$	
$\tau_n^{\mathcal{A}}$	{`just` | `comp`}	$L_n^{\mathcal{A}}$

Transition $\tau^{\mathcal{A}}$	fairness of τ	$enabled^{-}(\tau)$
$\ell_1^{\mathcal{A}}$	`just`	ℓ_1
$\ell_2^{\mathcal{A}}$	`just`	$\ell_2 \wedge (b^{y_2=0} \vee b^{y_1 \le y_2})$
$\ell_3^{\mathcal{A}}$	`just`	ℓ_3
$\ell_4^{\mathcal{A}}$	`just`	ℓ_4
$m_1^{\mathcal{A}}$	`just`	m_1
$m_2^{\mathcal{A}}$	`just`	$m_2 \wedge (b^{y_1=0} \vee \neg b^{y_1 \le y_2})$
$m_3^{\mathcal{A}}$	`just`	m_3
$m_4^{\mathcal{A}}$	`just`	m_4

Table 6.1: Fairness table for abstract BAKERY

where each transition $\tau_i^{\mathcal{A}}$ appears at most once.

Explicit upper bounds in the table would be redundant, since any upper bound on *enabled*(τ) that is found independently of $\tau^{\mathcal{A}}$ should be conjoined with $\tau^{\mathcal{A}}$ to get a more precise over-approximation (if it is not already subsumed by $\tau^{\mathcal{A}}$). Note that a fairness table can be easily extracted from a partial or complete GVD or DMC proof (see Chapter 5).

Example 6.1.3 (Fairness table for BAKERY) *Table 6.1 shows a Fairness table for the* BAKERY *abstract version shown in Figure 3.2. In this case, the concrete enabling conditions are exactly approximated in the abstract system, so the lower bounds are tight.*

BAKERY *transitions ℓ_0 and m_0 are not fair, so $\ell_0^{\mathcal{A}}$ and $m_0^{\mathcal{A}}$ do not appear in the table.*

6.1.2 The Termination Table

We will also want to reuse the ranking functions gathered for a given system. The relevant information can be summarized as follows: if a transition τ is taken at a state satisfying $\gamma(p)$ and reaches a state that satisfies $\gamma(q)$, then a value over particular well-founded domain $\mathcal{D}$ is (a) always decreased, or (b) not increased. This is justified by defining ranking functions δ_1 and δ_2 at p and q, as done by DMC and GVD's in Chapter 5, and proving the appropriate verification conditions. As with the fairness bounds, we gather this information in a table:

Definition 6.1.4 (Termination table) *A* termination table *contains the following information:*

Transition	*Ranking Functions*	*Precondition*	*Postcondition*	*Result*
$\tau_1^{\mathcal{A}}$	$\langle \delta_{1,1},\ \delta_{1,2} \rangle$	$pre_1^{\mathcal{A}}$	$post_1^{\mathcal{A}}$	$\{\preceq \mid \prec\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\tau_n^{\mathcal{A}}$	$\langle \delta_{n,1},\ \delta_{n,2} \rangle$	$pre_n^{\mathcal{A}}$	$post_n^{\mathcal{A}}$	$\{\preceq \mid \prec\}$

where $\delta_{i,1}$ and $\delta_{i,2}$ are ranking functions with the same domain $\mathcal{D}_i$, and $pre_i^{\mathcal{A}}$ and $post_i^{\mathcal{A}}$ are assertions. Each $\tau_i^{\mathcal{A}}$ and pair $\langle \delta_i, \delta_j \rangle$ can appear multiple times.

This table implicitly defines well-founded relations over Σ. The main point to note concerning these tables is that they contain purely propositional, finite-state information[1]. The verification conditions that justify a termination table row i with result $\prec$ (resp. $\preceq$) are:

$$\gamma(pre_i^{\mathcal{A}})(\mathcal{V}) \wedge \tau_i(\mathcal{V}, \mathcal{V}') \wedge \gamma(post_i^{\mathcal{A}})(\mathcal{V}') \rightarrow \delta_{i,1}(\mathcal{V}) \prec \text{ (resp. } \preceq) \, \delta_{i,2}(\mathcal{V}')$$

and

$$\Big(\gamma(pre_i^{\mathcal{A}})(\mathcal{V}) \ \vee \ \gamma(post_i^{\mathcal{A}})(\mathcal{V})\Big) \rightarrow (\delta_{i,1}(\mathcal{V}) \in \mathcal{D}_i \ \wedge \ \delta_{i,2}(\mathcal{V}) \in \mathcal{D}_i) \quad .$$

A termination table is *$\mathcal{S}$-valid* iff these verification conditions are $\mathcal{S}$-valid. Note that the well-foundedness of (the relation induced by) a ranking function depends on possibly local conditions on the system states. For example, if N is a fixed constant and x an integer-valued system variable, the ranking function $\delta_1 : N - x$ is well-founded provided $N \geq x + k_1$, for some constant k_1. The ranking function $\delta_2 : x$ is well-founded provided $x > k_2$. In the table, these conditions are built into $pre_i^{\mathcal{A}}$ and $post_i^{\mathcal{A}}$.

Example 6.1.5 (Termination Table for Bakery) *Our abstract version of the* BAKERY *program has basis*

$$\Big\{ b^{y_1=0} : y_1 = 0,\ b^{y_2=0} : y_2 = 0,\ b^{y_1 \leq y_2} : \ y_1 \leq y_2 \Big\} \quad .$$

If we add

$$b^{N \geq \max} : \ N \geq \max(y_1, y_2)$$

[1] Only identifiers for the ranking functions are needed, as we will see.

Transition	Ranking Functions	Precondition	Postcondition	Result
$\ell_1^{\mathcal{A}}$ $(y_1 := y_2 + 1)$	$\langle \delta_{\max}, \delta_{\max} \rangle$	$b^{y_1 \le y_2}$	$b^{N \ge \max}$	$\prec$
$\ell_1^{\mathcal{A}}$ $(y_1 := y_2 + 1)$	$\langle \delta_{\max}, \delta_{\max} \rangle$	*true*	$b^{N \ge \max}$	$\preceq$
$m_1^{\mathcal{A}}$ $(y_2 := y_1 + 1)$	$\langle \delta_{\max}, \delta_{\max} \rangle$	$\neg(b^{y_1 \le y_2})$	$b^{N \ge \max}$	$\prec$
$m_1^{\mathcal{A}}$ $(y_2 := y_1 + 1)$	$\langle \delta_{\max}, \delta_{\max} \rangle$	*true*	$b^{N \ge \max}$	$\preceq$
$\ell_4^{\mathcal{A}}$ $(y_1 := 0)$	$\langle \delta_{\max}, \delta_{\max} \rangle$	$b^{N \ge \max} \wedge b^{y_1 \le y_2}$	*true*	$\preceq$
$m_4^{\mathcal{A}}$ $(y_2 := 0)$	$\langle \delta_{\max}, \delta_{\max} \rangle$	$b^{N \ge \max} \wedge \neg(b^{y_1 \le y_2})$	*true*	$\preceq$
$\begin{bmatrix} \ell_0, \ell_2, \ell_3, \\ m_0, m_2, m_3 \end{bmatrix}$	$\langle \delta_{\max}, \delta_{\max} \rangle$	$b^{N \ge \max}$	*true*	$\preceq$

Table 6.2: Termination table for abstract BAKERY

we can construct a termination table for BAKERY *featuring ranking function*

$$\delta_{\max} : \ N - \max(y_1, y_2)$$

whose domain is the set of non-negative integers, as shown in Table 6.2.

Recall that in general each row of a termination table includes a pair of ranking functions δ_i and δ_j over the same well-founded domain $\mathcal{D}$. The table defines a set of predicates over $\Sigma_{\mathcal{A}} \times \Sigma_{\mathcal{A}}$, as follows:

Definition 6.1.6 (Predicates $P_{\prec}^{\delta_{i,j,k}}$, $P_{\preceq}^{\delta_{i,j,k}}$, $P_{?}^{\delta_{i,j,k}}$) *A termination table defines an abstract (finite-state) predicate $P_{\prec}^{\delta_{i,j,k}}$ (resp. $P_{\preceq}^{\delta_{i,j,k}}$) for each δ_i, δ_j and transition $\tau_k^{\mathcal{A}}$:*

$$P_{\prec}^{\delta_{i,j,k}}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \ \textit{(resp. } P_{\preceq}^{\delta_{i,j,k}}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}})\textit{)} \ \stackrel{\text{def}}{=} \ \bigvee_r^{\mathcal{A}} \left(pre_r^{\mathcal{A}}(\mathcal{V}_{\mathcal{A}}) \wedge^{\mathcal{A}} post_r^{\mathcal{A}}(\mathcal{V}'_{\mathcal{A}}) \wedge^{\mathcal{A}} \tau_k^{\mathcal{A}}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \right)$$

where r ranges over all the rows in the table where δ_i, δ_j and τ_k appear with result $\prec$ (resp. $\preceq$). If there are no such rows, the predicates are $false^{\mathcal{A}}$. We define

$$P_{?}^{\delta_{i,j,k}}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \stackrel{\text{def}}{=} \neg^{\mathcal{A}} \left(P_{\preceq}^{\delta_{i,j,k}}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \vee^{\mathcal{A}} \ P_{\prec}^{\delta_{i,j,k}}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \right) \ .$$

These abstract predicates satisfy the following:

$$\begin{array}{rcl}\gamma(P_{\prec}^{\delta_{i,j,k}}(\mathcal{V}_{\mathcal{A}},\mathcal{V}'_{\mathcal{A}})) \bigcap \tau_k(\mathcal{V},\mathcal{V}') & \subseteq & \{\langle\mathcal{V},\mathcal{V}'\rangle \mid \delta_j(\mathcal{V}') \prec \delta_i(\mathcal{V})\} \\ \gamma(P_{\preceq}^{\delta_{i,j,k}}(\mathcal{V}_{\mathcal{A}},\mathcal{V}'_{\mathcal{A}})) \bigcap \tau_k(\mathcal{V},\mathcal{V}') & \subseteq & \{\langle\mathcal{V},\mathcal{V}'\rangle \mid \delta_j(\mathcal{V}') \preceq \delta_i(\mathcal{V})\} \\ \gamma(P_{?}^{\delta_{i,j,k}}(B,B')) & \supseteq & \{\langle\mathcal{V},\mathcal{V}'\rangle \mid \tau_k(\mathcal{V},\mathcal{V}') \wedge \delta_j(\mathcal{V}') \succ \delta_i(\mathcal{V})\} \quad .\end{array}$$

Intuitively, $P_{\prec}^{\delta_{i,j,k}}$ (resp. $P_{\preceq}^{\delta_{i,j,k}}$) is true when the value of δ_j after taking τ_k is known to be smaller than (resp. not greater than) the value of δ_i at the current state. The relation $P_{?}^{\delta_{i,j,k}}$ holds when the change is unknown. Note that we can use the under-approximation procedure $\alpha(-,C,f)$ of Chapter 4 to generate these abstract predicates, provided we identify suitable ranking functions and that the validity checker can reason about them. Alternatively, given concrete *pre* and *post* assertions, the under-approximation procedure can be used to obtain the corresponding termination table entries.

These predicates are all that needs to go in the table, but we prefer the $\langle\delta_i,\delta_j\rangle$ notation to make the connection with DMC and GVD's clearer.

Definition 6.1.7 (Ranking function node labeling μ_δ) *Let $\mathcal{G}$ be a graph where each node n is labeled by a propositional formula $\mu(n)$ and edges are labeled by sets of transitions. A* ranking function node labeling *μ_δ maps each node in $\mathcal{G}$ to a ranking function δ_n from the termination table.*

The edge labeling induced by *μ_δ is as follows: an edge (n_1,n_2), with $\mu_\delta(n_1):\delta_i$ and $\mu_\delta(n_2):\delta_j$ labeled with transitions T, is labeled under μ_δ as*

$\prec$, *if $P_{\prec}^{\delta_{i,j,k}}(\mu(n_1),\mu(n_2))$ holds for all $\tau_k \in T$; else*
$\preceq$, *if $P_{\preceq}^{\delta_{i,j,k}}(\mu(n_1),\mu(n_2))$ holds for all other $\tau_k \in T$; else*
$?$ *that is, $P_{?}^{\delta_{i,j,k}}(\mu(n_1),\mu(n_2))$ holds for some $\tau_k \in T$.*

The well-foundedness or termination of an SCS in GVD's (Section 5.3.1) or DMC (Section 5.5.2) can be expressed in terms of these node and edge labels in a straightforward way. Given μ_δ, the induced edge labeling can be easily and automatically computed using a termination table. When $\mathcal{G}$ is the (abstract) Kripke structure, the node label $\mu(s)$ is simply the valuation of the state-variables at s.

6.2 Handling Fairness and Well-Foundedness

The detailed fairness and well-foundedness information described above can be built into the abstraction in three main ways:

- Encode the information as new fairness constraints in the abstract system (this is suggested, e.g., in [DGH95, Dams96]). These constraints are best encoded as acceptance conditions for a finite-state ω-automaton that represents $\mathcal{A}$, so this approach would be best suited when working in an automata-theoretic model checking framework (see Section 2.3.2).

- Represent the constraints as a temporal logic formula $\mathcal{C}^{\mathcal{A}}$, and then model check $\mathcal{C}^{\mathcal{A}} \rightarrow^{\mathcal{A}} \varphi^{\mathcal{A}}$ rather than $\varphi^{\mathcal{A}}$. We sketch a way of doing this in Section 6.3.

- Modify the finite-state model checker to account for the explicitly indicated fairness and well-foundedness constraints. We present this in Section 6.4.

6.3 Temporal Encoding

6.3.1 Encoding Fairness

The temporal constraints are reminiscent to those used in the general simulation and refinement rules of [KMP94] and Theorem 3.6.1. However, we now use only *abstract* formulas, including under-approximations of the concrete enabling conditions. This yields a purely propositional temporal constraint.

Definition 6.3.1 (Temporal constraint $\mathcal{C}^{\mathcal{A}}_{Fair}$) *For a transition-mapped abstraction $\mathcal{A} : \langle \mathcal{V}_{\mathcal{A}}, \Theta_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}} \rangle$ of $\mathcal{S} : \langle \mathcal{V}_{\mathcal{C}}, \Theta_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}} \rangle$:*

$$\mathcal{C}^{\mathcal{A}}_{\tau} \stackrel{\text{def}}{=} \begin{cases} (enabled^{-}(\tau^{\mathcal{A}})) \;\Rightarrow\; \diamondsuit \left(\neg^{\mathcal{A}} enabled^{-}(\tau^{\mathcal{A}}) \vee^{\mathcal{A}} taken(\tau^{\mathcal{A}}) \right) & \text{if } \tau \text{ is just} \\ \square \diamondsuit\, enabled^{-}(\tau^{\mathcal{A}}) \;\Rightarrow\; \diamondsuit\, taken(\tau^{\mathcal{A}}) & \text{if } \tau \text{ is compassionate} \\ true^{\mathcal{A}} & \text{otherwise} \end{cases}$$

Then:

$$\mathcal{C}^{\mathcal{A}}_{Fair} \stackrel{\text{def}}{=} \bigwedge_{\tau \in \mathcal{T}_{\mathcal{C}}} \mathcal{C}^{\mathcal{A}}_{\tau} \;.$$

Note that since abstract formulas are boolean, the *taken* predicate can be expressed simply by replacing each primed proposition b' by $\bigcirc b$. Thus, the rigid quantification or extended syntax needed in the (first-order) concrete case (see Section 3.6) is no longer necessary.

Lemma 6.3.2 (Soundness of $\mathcal{C}^{\mathcal{A}}_{Fair}$)

$$\mathcal{L}(\mathcal{S}^{\mathcal{C}}) \ \subseteq \ \gamma(\mathcal{L}(\mathcal{S}^{\mathcal{A}}) \ \cap \ \mathcal{L}(\mathcal{C}^{\mathcal{A}}_{Fair}))) \ .$$

Proof: Consider a computation $\pi_{\mathcal{A}} : a_0, a_1, \ldots$ of $\mathcal{S}^{\mathcal{A}}$. If $\pi_{\mathcal{A}}$ does not satisfy $\mathcal{C}^{\mathcal{A}}_{\tau}$ for some τ, then all the sequences $\pi : s_0, s_1, \ldots$ in $\gamma(\pi_{\mathcal{A}})$ fail to be computations of $\mathcal{S}$:

Given a fairness table, we know that τ is enabled at all $\gamma(\mathit{enabled}^{-}(\tau^{\mathcal{A}}))$-states. In the case of justice, if the constraint $\mathcal{C}^{\mathcal{A}}_{\tau}$ is violated then there is an $\mathit{enabled}^{-}(\tau^{\mathcal{A}})$ state followed only by states where $\mathit{enabled}^{-}(\tau^{\mathcal{A}}) \wedge^{\mathcal{A}} \neg^{\mathcal{A}} \mathit{taken}(\tau^{\mathcal{A}})$ holds. But then τ would also be enabled at those states in π, since $\mathit{enabled}^{-}(\tau^{\mathcal{A}})$ is a lower bound, but never taken, since $\tau^{\mathcal{A}}$ is an over-approximation of τ (so $\gamma(\neg^{\mathcal{A}} \mathit{taken}(\tau^{\mathcal{A}}))$ implies $\neg \mathit{taken}(\tau)$). Thus π is not just towards τ.

The case of compassion is similar: the constraint $\mathcal{C}^{\mathcal{A}}_{\tau}$ is violated if from some point in $\pi_{\mathcal{A}}$ onwards, $\mathit{enabled}^{-}(\tau^{\mathcal{A}})$ holds infinitely often but $\neg^{\mathcal{A}} \mathit{taken}(\tau^{\mathcal{A}})$ is always true. This means that in the corresponding states of any $\pi \in \gamma(\pi_{\mathcal{A}})$, transition τ is infinitely often enabled but never taken, so π is unfair towards τ. $\square$

6.3.2 Encoding Well-founded Orders

Rows in the termination table where δ_i and δ_j are the same ranking function can be similarly encoded as temporal constraints. For such rows, the following temporal constraint states that if the ordering is known to decrease infinitely many times, then it must have a chance to increase infinitely many times as well:

$$\mathcal{C}^{\mathcal{A}}_{\delta_i} : \square \diamond \left(\bigvee_{\tau_k \in \mathcal{T}} P^{\delta_i, i, k}_{\prec}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \right) \ \rightarrow \square \diamond \left(\bigvee_{\tau_k \in \mathcal{T}} P^{\delta_i, i, k}_{?}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \right) \ .$$

Again, since we are dealing with purely propositional formulas, the primed variables can be expressed using $\bigcirc$. Since the order is well-founded, we have:

Lemma 6.3.3 (Soundness of $\mathcal{C}^{\mathcal{A}}_{\delta_i}$)

$$\mathcal{L}(\mathcal{S}^{\mathcal{C}}) \subseteq \gamma(\mathcal{L}(\mathcal{S}^{\mathcal{A}}) \cap \mathcal{L}(\mathcal{C}^{\mathcal{A}}_{\delta_i}))) \ .$$

Proof: Consider a run $\pi_{\mathcal{A}} : a_0, a_1, \ldots$ of $\mathcal{S}^{\mathcal{A}}$ that does not satisfy $\mathcal{C}^{\mathcal{A}}_{\delta_i}$. Assume that π is a run of $\mathcal{S}$ in $\gamma(\pi_{\mathcal{A}})$. Then $\bigvee_k P^{\delta_{i,i,k}}_{\prec}$ holds infinitely often at $\pi_{\mathcal{A}}$; at the corresponding states in π, the well-founded order δ_i is guaranteed to decrease. However, $\bigvee_k P^{\delta_{i,i,k}}_{?}$, which includes all the steps at which δ_i could increase, holds only finitely many times. This contradicts the well-foundedness of the relation. Thus, $\gamma(\pi_{\mathcal{A}})$ cannot contain any computation of $\mathcal{S}$. □

6.3.3 A Mixed Approach

The temporal encoding can be simplified by augmenting the abstract transition system. For each ranking function δ_i, two new bits $b_{\prec i}$ and $b_{\preceq i}$, initially $\mathit{false}^{\mathcal{A}}$, are added to $\mathcal{V}_{\mathcal{A}}$. Each abstract transition is augmented as follows: for each row in the termination table for transition $\tau^{\mathcal{A}}$, with a single ranking function δ_i and result $\prec$ (resp. $\preceq$), the conjunct

$$b'_{\prec i} \text{ (resp. } b_{\preceq i}) \leftrightarrow^{\mathcal{A}} \left(\mathit{pre}^{\mathcal{A}} \wedge^{\mathcal{A}} \mathit{post}^{\mathcal{A}'}\right)$$

is added to transition $\tau^{\mathcal{A}}$. All other transitions are augmented to set $b_{\prec i}$ and $b_{\preceq i}$ to $\mathit{false}^{\mathcal{A}}$. The constraint $\mathcal{C}^{\mathcal{A}}_{\delta_i}$ is then simply encoded as

$$\Box \Diamond b_{\prec i} \ \rightarrow \ \Box \Diamond \neg \left(b_{\prec i} \vee b_{\preceq i}\right) \ .$$

Example 6.3.4 (Model checking growth beyond any bound) *A single ranking function is used in the* BAKERY *termination table of Example 6.1.5, so the simple temporal encoding is sufficient in this case. Furthermore, since the guards are approximated exactly, we can retain the original fairness constraints, and let a standard model checker handle them. We would like to prove*

$$\varphi : \Diamond \Box \neg((\ell_0 \vee \ell_1) \wedge (m_0 \vee m_1)) \ \rightarrow \ \Diamond(\max(y_1, y_2) > N)$$

for the original BAKERY *program. Adding the new bit*

$$b^{N\geq\max} : \; N \geq \max(y_1, y_2)$$

to the abstraction $\mathcal{S}^{\mathcal{A}}$, it is sufficient to check that $\mathcal{S}^{\mathcal{A}}$ satisfies

$$\varphi^{\mathcal{A}} : \Diamond\,\Box\,\neg((\ell_0 \vee \ell_1) \wedge (m_0 \vee m_1)) \;\rightarrow\; \Diamond\,\neg b^{N\geq\max} \;\; ,$$

which in this case is equivalent to $\alpha^t(\varphi)$. Indeed, we can model check $\mathcal{C}^{\mathcal{A}}_{\delta_{\max}} \rightarrow \varphi^{\mathcal{A}}$ over $\mathcal{S}^{\mathcal{A}}$, or model check

$$(\Box\,\Diamond\, b_{\prec} \rightarrow \Box\,\Diamond\,\neg\,(b_{\prec} \vee b_{\preceq})) \rightarrow \varphi^{\mathcal{A}}$$

over an augmented transition system as described above. This proves that φ is valid over the original BAKERY *program. A computation that satisfies the antecedent of φ can be found by model checking a simple existential abstraction. Thus, we can prove that* BAKERY *is infinite-state by algorithmically model checking finite-state abstractions.*

The result is analogous to the DMC proof for the same property presented in [Sip98, BMSU98]. That proof uses the invariants

$$\begin{array}{lcll} \ell_2 \wedge (m_3 \vee m_4) & \Rightarrow & y_2 < y_1 & (\neg b^{y_1 \leq y_2}) \\ m_2 \wedge (l_3 \vee l_4) & \Rightarrow & y_1 \leq y_2 & (b^{y_1 \leq y_2}) \end{array}$$

These invariants are implied by the safety component of the abstraction, so there is no need to prove them separately.

In general, we can conjoin all the constraints $\mathcal{C}^{\mathcal{A}}_{\delta_i}$ that correspond to termination table rows where $\delta_i = \delta_j$. However, this simple approach does not capture all the usable information in the termination table, since it does not consider the interaction between different ranking functions; contrast this with the node labelings of DMC and GVD's.

Example 6.3.5 *Figure 6.1 shows a simple three-state abstract FTS. Assume that the termination table is:*

Transition	*Ranking Functions*	*Precondition*	*Postcondition*	*Result*
$\tau_1^{\mathcal{A}}$	$\langle\delta_1, \delta_2\rangle$	A	B	$\prec$
$\tau_2^{\mathcal{A}}$	$\langle\delta_2, \delta_3\rangle$	B	C	$\preceq$
$\tau_3^{\mathcal{A}}$	$\langle\delta_3, \delta_1\rangle$	C	A	$\preceq$

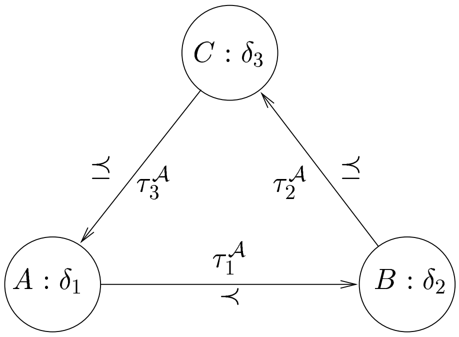

Figure 6.1: Pairwise constraints are not enough for mixed $\langle \delta_i, \delta_j \rangle$ pairs

where δ_1, δ_2 and δ_3 are ranking functions over the same domain. All other results are implicitly "?": for instance, when we go from node A to node B, we do not know if δ_3 increases; when going from B to C, we are not sure about δ_1. Regardless of the initial condition, it is clear that this FTS satisfies

$$\varphi^{\mathcal{A}} : \diamondsuit\,(\square\, A \vee \square\, B \vee \square\, C) \quad .$$

However, there are no simple pairwise constraints $\mathcal{C}^{\mathcal{A}}_{\delta_i}$ over ranking functions $\{\delta_1, \delta_2, \delta_3\}$ that imply $\varphi^{\mathcal{A}}$.

Note that pairwise encoding is sufficient for our BAKERY example (Example 6.3.4), since its termination table (Table 6.1.5) does not have mixed pairs of ranking functions. In the following section we show that, indeed, using mixed pairs is not essential.

6.3.4 Single vs. Mixed Pairs of Ranking Functions

We allow termination table rows to contain different ranking functions for convenience. However, we can show that from the theoretical point of view, this flexibility is not required. First, we note:

Proposition 6.3.6 *Let S be an abstract SCS whose nodes are* disjoint*: if $n_i \neq n_j$, then $\gamma(\mu(n_i)) \cap \gamma(\mu(n_j)) = \emptyset$. Then for any node labeling μ_δ for S, there is a node labeling μ'_δ that uses a single ranking function and induces the same edge labeling for S.*

Proof: If $n_1, \ldots, n_k$ are the nodes in S, and the node labels are disjoint, we can define μ'_δ as:

$$\mu'_\delta(\mathcal{V}) \stackrel{\text{def}}{=} \begin{cases} \mathit{if}\ \gamma(\mu(n_1))(\mathcal{V})\ \mathit{then}\ \mu_\delta(n_1)(\mathcal{V})\ \mathit{else} \\ \mathit{if}\, \gamma(\mu(n_2))(\mathcal{V})\ \mathit{then}\ \mu_\delta(n_2)(\mathcal{V})\ \mathit{else} \\ \ldots\ \mathit{else}\ \gamma(\mu(n_k))(\mathcal{V}) \end{cases}$$

Clearly, the edge labels for S induced by μ and μ' are the same. The new order is well-founded since all ranking functions are assumed to have the same well-founded domain. □

We can ensure that the nodes of an abstract SCS are disjoint by adding a (finite-domain) control variable that takes a different value at each node. This variable plays a role similar to that of the unobservable control variable of Section 5.3, but it should now be present in the concrete system as well.

Example 6.3.7 *For the SCS of Example 6.3.5, we can introduce a new ranking function, δ_{ABC}, defined as follows:*

$$\delta_{ABC}(\mathcal{V}) \stackrel{\text{def}}{=}\ \mathit{if}\ A(\mathcal{V})\ \mathit{then}\ \delta_1(\mathcal{V})\ \mathit{else}\ \mathit{if}\ B(\mathcal{V})\ \mathit{then}\ \delta_2(\mathcal{V})\ \mathit{else}\ \delta_3(\mathcal{V}) \quad .$$

Adapting DMC terminology, we say:

Definition 6.3.8 (T-terminating SCS) *Given a termination table T, an SCS S is T-*terminating *if for any infinite path $\pi : n_0, n_1, \ldots$ through S, there is a mapping from each position i to a ranking function δ_i such that the induced edge labeling contains infinitely many "$\prec$" and no "?". We call this a* suitable mapping *for π.*

A suitable mapping is a node labeling for the *infinite* graph described by π, where each position contains a unique node. Thus, different occurrences of the same node in S can be labeled with a different δ in π. Intuitively, the labeling on a node may depend on the path taken to reach that node, so we need a different copy of that node for each case. Thus, an SCS can be T-terminating but have no terminating edge (see Definition 5.5.3), which explains the need for unfolding in DMC (Definition 5.5.4). However, we have:

Proposition 6.3.9 *To determine if an SCS S is T-terminating for a given termination table T, only finitely many node labelings for a finite number of finite graphs must be considered.*

Proof: (Outline) An infinite path π has a suitable mapping if and only if there is a finite cycle (where we identify some of the different node occurrences that have the same labels) that can be labeled to include at least one "$\prec$." Only the smallest such cycles need to be considered, of which there is a finite number. □

A more efficient procedure, suggested by N. Bjørner, constructs a graph $\mathcal{G}$ from S, whose nodes are the pairs $\langle n, e \rangle$ for a node n and edge e in S, and where $\langle n_1, e_1 \rangle$ and $\langle n_2, e_2 \rangle$ are connected by an edge in $\mathcal{G}$ if e_1 is an edge from n_1 to n_2 in S. It is sufficient to ensure that each cycle in this graph corresponds to a path π with a suitable mapping. This, in turn, can be done by taking the product of $\mathcal{G}$ and a graph that represents T, whose nodes are the ranking functions δ_i and whose edges are the rows in T. It may also be possible to adapt the method for finding ranking function labelings presented in Section 6.4 for this task.

In Section 6.4 we will see that for any finite-state abstraction, there are only finitely many SCS's that must be shown to be terminating. If finitely many node labelings for finite graphs are needed for each, then by Proposition 6.3.6 it is theoretically sufficient to use a termination table with a single ranking function δ_i in each row. In this case, the conjunction of the pairwise temporal constraints $\mathcal{C}^{\mathcal{A}}_{\delta_i}$ will be enough as well. This observation shows that the termination table constraints do not go beyond those expressible by regular ω-automata.

The conditional ranking functions above requires anticipating what the SCS's of interest are. However, we would like to use an existing termination table to model check new properties, as we will do in Section 6.4, so an efficient implementation of a T-termination test for mixed $\langle \delta_i, \delta_j \rangle$ pairs is preferred.

6.3.5 The Combined Fairness and Termination Constraint

To summarize this section, instead of model checking $\varphi^{\mathcal{A}}$, we can model check

$$\left(\bigwedge_{\mathcal{D}} \mathcal{C}^{\mathcal{A}}_{\delta_{i,j}} \right) \wedge \mathcal{C}^{\mathcal{A}}_{Fair} \rightarrow \varphi^{\mathcal{A}} \ .$$

To fully capture the information in the termination table, this requires that each row use a single ranking function. Another potential drawback of this approach is that the formula to be model checked can be quite large, leading to inefficiency if its tableau is explicitly constructed.[2] An alternative is to express the temporal constraints in the μ-calculus and

[2] In general, the complexity of LTL model checking is exponential in the size of the formula.

then use symbolic model checking [BCJM96], as done for LTL under fairness in the STeP symbolic model checker [Bjø98b].

The final alternative we consider is to incorporate the constraints directly into the model checking process, which we describe in the following section.

6.4 Extended Model Checking

Based on the insights provided by DMC and GVD's, we now propose an explicit-state model checking algorithm for finite-state abstractions that takes advantage of the extra information concerning the enabling conditions of fair transitions and well-founded orders described in Section 6.1. We call this an *extended abstract model checking* (E$_\mathrm{X}$MC) procedure.[3]

This procedure is simply a static version of Deductive Model Checking (see Section 5.5), where we assume that a fixed amount of knowledge about the concrete system is given *a priori* and expressed in terms of a finite-state abstraction.

6.4.1 The E$_\mathrm{X}$MC Input

The inputs to the algorithm are an abstract LTL property $\varphi^{\mathcal{A}}$ to be model checked, and an *extended abstract system* $\mathcal{S}^{\mathcal{A}}$, which includes the following components:

- A standard ($\exists\exists$) finite-state abstract transition system with no fairness requirements, which defines the runs of $\mathcal{S}^{\mathcal{A}}$.
- A fairness table for $\mathcal{S}^{\mathcal{A}}$.
- A termination table for $\mathcal{S}^{\mathcal{A}}$.
- Optional: a set of constrained ($\forall\exists$) abstract transition relations (see Sections 3.5 and 4.9).

We will only speak of the runs of $\mathcal{S}^{\mathcal{A}}$, as defined by the over-approximated abstract transition relations. Informally, the computations of $\mathcal{S}^{\mathcal{A}}$ are those runs that do not contradict the fairness and termination tables. We will write $\mathcal{S}^{\mathcal{A}} \vDash \varphi^{\mathcal{A}}$ if the E$_\mathrm{X}$MC procedure can establish that that no run $\pi_{\mathcal{A}}$ of $\mathcal{S}^{\mathcal{A}}$ can satisfy $\neg\varphi^{\mathcal{A}}$ and include, in $\gamma(\pi_{\mathcal{A}})$, a computation

[3] The "E$_\mathrm{X}$MC" acronym for this procedure is not to be confused with the classic EMC model checking program of Edmund M. Clarke et.al. [BCDM86].

of $\mathcal{S}$ that satisfies the constraints expressed by the fairness and termination tables. This will imply that $\mathcal{S} \models \gamma(\varphi^{\mathcal{A}})$.

6.4.2 The EXMC Procedure

For convenience, in this section we assume that the Kripke structure (and the product graph) has one edge per transition, rather than a single edge labeled with multiple transitions (see Remark 2.1.4). The Extended Model Checking algorithm is:

1. Compute the product graph $\mathcal{G}^{\mathcal{A}} : \mathcal{S}^{\mathcal{A}} \times T_{\neg\varphi^{\mathcal{A}}}$ (see Definition 2.3.2). Recall that a node $\left\langle s^{\mathcal{A}}, a \right\rangle$ is initial in $\mathcal{G}^{\mathcal{A}}$ iff $s^{\mathcal{A}}$ is initial in $\mathcal{S}^{\mathcal{A}}$ and a is initial in $T_{\neg\varphi^{\mathcal{A}}}$.

   For a node $n : \left\langle s^{\mathcal{A}}, a \right\rangle$, let $\mu(n) \stackrel{\text{def}}{=} s^{\mathcal{A}} \wedge^{\mathcal{A}} \mathit{assertions}^{\mathcal{A}}(a)$, where $\mathit{assertions}^{\mathcal{A}}(a)$ is the set of assertions in the tableau atom a, treated as propositional formulas.

   Remove nodes n where $\mu(n)$ is propositionally unsatisfiable. Let each transition $\tau^{\mathcal{A}}$ label an edge from n_1 to n_2 iff $\mu(n_1) \wedge^{\mathcal{A}} \tau^{\mathcal{A}} \wedge^{\mathcal{A}} \mu'(n_2)$ is satisfiable. Again, this is a propositional test.

2. We say that a (propositional) predicate p holds at a node n if $\mu(n) \rightarrow p$ is (propositionally) valid.

   An SCS S is *unjust* towards a just transition τ if the predicate $\mathit{enabled}^{-}(\tau)$ holds at all nodes in S and $\tau^{\mathcal{A}}$ does not label any edge in S.

   S is *uncompassionate* towards a compassionate transition τ if the predicate $\mathit{enabled}^{-}(\tau)$ holds at some node n in S and $\tau^{\mathcal{A}}$ does not label any edge in S.

   S is *unfair* if it is unjust or uncompassionate towards some τ.

3. We say that an SCS S is *adequate* if and only if it is returned by the following test:

   (a) If S is unfulfilling or unjust, **fail**. Otherwise:

   (b) If S is uncompassionate towards τ, recursively invoke the procedure on all *maximal* SCS's (MSCS's) of the subgraph of S obtained by removing the nodes where $\mathit{enabled}^{-}(\tau)$ holds. **fail** if all these calls fail.

   Otherwise (S is fulfilling, just and compassionate):

   (c) Choose a ranking function node labeling μ_δ for S and compute the induced edge labeling (see Definition 6.1.7 above, where each edge is now labeled by a single transition). We say that μ_δ is *suitable* for S if no edge is labeled "?" and some edge is labeled "$\prec$."

If a suitable μ_δ is found, recursively invoke the adequacy procedure on all the MSCS's obtained by removing the edges labeled "$\prec$" from S. **succeed** if any of these tests succeeds; **fail** if all these tests fail.

If there is no suitable μ_δ for S, recursively invoke the procedure on the result of *unfolding* S (see Definition 5.5.4). If this fails, **succeed**, returning the current SCS.

4. Invoke the above adequacy procedure on each reachable MSCS in $\mathcal{G}^{\mathcal{A}}$. If no adequate SCS is found, report that $\mathcal{S}^{\mathcal{A}} \models \varphi^{\mathcal{A}}$. Otherwise, return an adequate reachable SCS and a path that leads to it from an initial state as an abstract counterexample.

Note again that the EXMC procedure only performs propositional checks, and that all structures are finite.

The use of unfolding in section 3 (c) of the algorithm can be replaced by a T-termination test, which should return a cycle in the SCS for which no suitable mapping exists, as a counterexample, or else **fail**, indicating that the SCS is T-terminating and thus not adequate. The EXMC procedure should never apply unfold twice in succession: if the main ring in an unfolded SCS cannot be broken, the above adequacy test should fail, returning the original SCS.

If an SCS is unfulfilling (resp. unjust), then all of its subgraphs are unfulfilling (resp. unjust). Thus, there is no need to check sub-SCS's, except for the case of compassion. Note also that there is no need for backtracking when choosing a labeling: once a labeling is found for S under which an edge can be removed, other labelings for S do not have to be considered.

Figure 6.2 modifies the example of Figure 6.1 to illustrate the interactions that can occur between fairness and well-founded orders. The SCS $\{A, B, C\}$ does not directly violate any well-founded order. However, assume that the compassionate transition $\tau_1^{\mathcal{A}}$ is fully enabled at A. Then $\tau_1^{\mathcal{A}}$ should be taken infinitely many times if a computation remains within this SCS, decreasing the well-founded order infinitely many times, a contradiction. The EXMC procedure detects that such an SCS is inadequate by removing $\tau_1^{\mathcal{A}}$ and then detecting that the resulting SCS is uncompassionate towards $\tau_1^{\mathcal{A}}$.

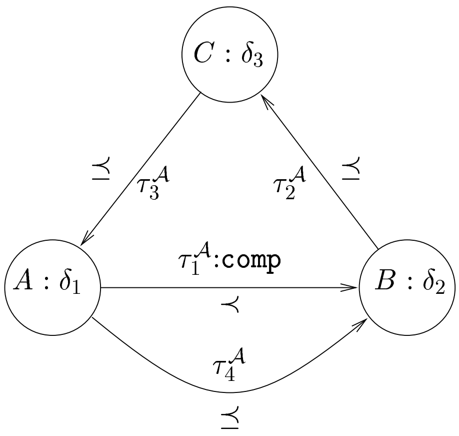

Figure 6.2: Interaction between fairness and ranking functions

Finding Ranking Function Labelings

Part (c) of the adequacy test is non-deterministic: we try all possible labelings μ_δ of S. The search for a suitable one can be carried out as follows:

For efficiency, each edge will be marked with the pairs $\langle \delta_i, \delta_j \rangle$ that *fail* for that edge.
Let all edges in S be unlabeled and $\mu_\delta := \{\}$.
procedure *extend*(μ_δ):

If μ_δ labels all the nodes in S, **succeed** with μ_δ;
else choose an unlabeled node n and a node label δ_i
    that is not know to fail for any of the edges at n.
Let $\mu_\delta := \mu_\delta \cup \{n \leftarrow \delta_i\}$.
Compute the new edge labels induced by μ_δ.
If any edge label is "?," **fail** and mark that edge
    to indicate the failure of the corresponding label pair.
Call *extend*(μ_δ). If it fails, choose the next δ_i for n.
If all δ_j's have been chosen, **fail**

The first two node labels, δ_i and δ_j, should be chosen such that at least one edge is labeled "$\prec$." New labels that induce "$\prec$" labels should be preferred. If D is the number of rows in the termination table, the procedure considers at most D^2 pairs of labels for each edge in the SCS, so the number of tests is polynomial in the number of edges in the SCS and the number of rows in the fairness table.

The basic E$_X$MC algorithm can be improved in may ways. For instance, the finite-state version of DMC itself can be used to expand the product graph in a top-down manner, hoping to avoid the expansion of the entire state-space (see Section 5.5). In this case, $\mu(n)$ will characterize a set of abstract states, rather than a single one, but the definitions and essentials of the algorithm remain unchanged. The tableau $T_{\neg\varphi^{\mathcal{A}}}$ can be generated "on-the-fly" as well.

Theorem 6.4.1 (Soundness of E$_X$MC) *If $\mathcal{S}^{\mathcal{A}}$ is a correct abstraction of $\mathcal{S}$ (including $\mathcal{S}$-valid fairness and termination tables) and the E$_X$MC procedure reports that $\mathcal{S}^{\mathcal{A}} \models \varphi^{\mathcal{A}}$, then $\mathcal{S} \models \gamma(\varphi^{\mathcal{A}})$.*

> **Proof:** (Outline) Any computation of $\mathcal{S}$ must be included in $\gamma(\pi_{\mathcal{A}})$ for some run $\pi_{\mathcal{A}}$ of $\mathcal{S}^{\mathcal{A}}$. The product graph $\mathcal{G}$ of $\mathcal{S}^{\mathcal{A}}$ and $T_{\neg\varphi^{\mathcal{A}}}$ describes all the runs of $\mathcal{S}^{\mathcal{A}}$ that satisfy $\neg\varphi^{\mathcal{A}}$.
>
> We say that a run $\pi_{\mathcal{A}}$ of $\mathcal{S}^{\mathcal{A}}$ is *adequate* if $\gamma(\pi_{\mathcal{A}})$ contains a computation of $\mathcal{S}$. An adequate run of $\mathcal{S}^{\mathcal{A}}$ that satisfies $\neg\varphi^{\mathcal{A}}$ must eventually remain within an adequate SCS of $\mathcal{G}$: otherwise, all runs in $\gamma(\varphi)$ must violate a fairness requirement of $\mathcal{S}$ or contradict a well-foundedness constraint.
>
> There are no adequate SCS's if the adequacy test fails on the MSCS's of $\mathcal{G}$. Thus, if E$_X$MC reports that $\mathcal{S}^{\mathcal{A}} \models \varphi^{\mathcal{A}}$, there are no adequate runs of $\mathcal{S}^{\mathcal{A}}$, which means that no computation of $\mathcal{S}$ satisfies $\neg\varphi$, that is, $\mathcal{S} \models \varphi$. □

Theorem 6.4.2 (Relative Completeness of E$_X$MC) *For any state-quantified LTL property φ, if $\mathcal{S} \models \varphi$ then there is a finite-state abstract system $\mathcal{S}^{\mathcal{A}}$, with finite fairness and termination tables, such that $\mathcal{S}^{\mathcal{A}} \models \alpha^t(\varphi)$ can be checked by E$_X$MC. This assumes, as usual, that the concrete assertion language is sufficiently expressive and that complete proof methods are available for establishing the verification conditions that justify the correctness of $\mathcal{S}^{\mathcal{A}}$.*

> **Proof:** (Outline) Deductive Model Checking is relatively complete [SUM99, Sip98]: if φ is $\mathcal{S}$-valid, then there is a DMC proof of it (assuming that the assertion language is expressive enough and that we can prove all valid verification conditions).

The final falsification diagram $\mathcal{G}$ produced by a DMC proof, including all the nodes that it generates, together with the verification conditions that show that $\mathcal{G}$ cannot support a $\neg\varphi$-computation, describes an extended abstract system for which the property can be successfully model checked:

1. As the assertion basis B, take the set of all atomic formulas that appear in $\mathcal{G}$.
2. As the abstract initial condition, take the disjunction

$$\Theta_{\mathcal{A}} : \bigvee\nolimits^{\mathcal{A}}_{n\in I} \left(\neg \mathit{assertions}^{\mathcal{A}}(n) \vee^{\mathcal{A}} \mu^{\mathcal{A}}(n)\right)$$

   over the set I of initial nodes in $\mathcal{G}$. Notice that we can over-approximate $\Theta_{\mathcal{A}}$ to $\mathit{true}^{\mathcal{A}}$ for the states that are not relevant to the property being model checked.

3. The abstract transition relations are given by the edges that are missing from τ, as described in Section 5.5. Transitions are only removed from an edge if the corresponding verification condition

$$(\mu(n_1) \wedge \tau \wedge \mu'(n_2))$$

   is unsatisfiable, that is, no pair of states that satisfies τ is represented by this edge. This adds the conjunct $\mu^{\mathcal{A}}(n_1) \rightarrow^{\mathcal{A}} \neg^{\mathcal{A}}\mu^{\mathcal{A}}(n_2)'$ to $\tau^{\mathcal{A}}$.

4. The fairness table entry for $\tau^{\mathcal{A}}$ has as $\mathit{enabled}^{-}(\tau)$ the disjunction $\bigvee^{\mathcal{A}}_{n}\mu^{\mathcal{A}}(n)$ over all the nodes n where τ was proved to be enabled in order to show that an SCS is unfair.
5. The termination table contains an entry

$$\tau^{\mathcal{A}}, \quad \langle\delta_1, \delta_2\rangle\,, \quad \mathit{pre}_i : \mu^{\mathcal{A}}(n_1), \quad \mathit{post}_i : \mu^{\mathcal{A}}(n_2), \quad \preceq \text{ (or } \prec)$$

   for each verification condition used to justify a terminating edge (see Definition 5.5.3).

We now must show that EXMC will also prove that $\mathcal{S}^{\mathcal{A}} \models \varphi^{\mathcal{A}}$. Notice that in this case, $\gamma(\varphi^{\mathcal{A}}) = \varphi$, since all of the atoms of φ are in the basis.[4]

[4]Since DMC simplifies formulas as it goes along, it could be the case that the basis contains more refined atoms instead, but $\gamma(\varphi^{\mathcal{A}})$ will still be equivalent to φ.

If the DMC proof is successful, each fulfilling SCS has been shown to be unreachable, unfair or terminating. We can assume without loss of generality that all the transformations that removed SCS's, including the unfolding transformation, were saved for last in the DMC proof.

It should be clear that if DMC discards an unfair or unfulfilling SCS, then E_XMC will do so as well, by the construction of the abstract system above. We are left with the terminating ones. The DMC completeness proof tells us that if an SCS with n nodes cannot support a concrete computation, then there is a well-founded order that disconnects $\mathcal{U}(S)$ into a set of smaller SCS's. Thus, assuming that the termination table has the required information, unfolding is sufficient for detecting T-terminating SCS's. □

The above proof assumes that the given termination table T includes all the necessary well-founded orders. Furthermore, it assumes that we have exactly the ones that the DMC proof uses. However, E_XMC may succeed even when other ranking functions are given, by using a general T-termination test.

Remark 6.4.3 Generalized Verification Diagrams (Section 5.3) are also relatively complete for state-quantified LTL properties [BMS95]. Given a GVD that proves $\mathcal{S} \vDash \varphi$, we can construct an extended abstract system $\mathcal{S}^{\mathcal{A}}$, as follows:

1. The assertion basis B is the set of all atomic assertions that appear in Ψ.

2. The abstract initial condition and over-approximated transition relations are given by the initiality and consecution conditions of the diagram.

3. If $\mu(m) \rightarrow enabled(\tau)$ is proved to show that an SCS has a fair exit, then $\mu^{\mathcal{A}}(m)$ is added as a disjunct to $enabled^{-}(\tau)$ in the fairness table for $\mathcal{S}^{\mathcal{A}}$.

4. If the verification condition

$$\mu(n_1) \wedge \tau \wedge \mu'(n_2) \;\rightarrow\; \delta_1(\mathcal{V}) \succ (\text{resp. } \succeq)\, \delta_2'(\mathcal{V})$$

is proved to show that an SCS is well-founded, a corresponding row

$$\tau^{\mathcal{A}}, \quad \langle \delta_1, \delta_2 \rangle\,, \quad pre_i^{\mathcal{A}} : \mu^{\mathcal{A}}(n_1), \quad post_i^{\mathcal{A}} : \mu^{\mathcal{A}}(n_2)$$

is added to the termination table for $\mathcal{S}^{\mathcal{A}}$.

For this abstract system $\mathcal{S}^{\mathcal{A}}$, ExMC can show that $\mathcal{S}^{\mathcal{A}} \vDash \varphi^{\mathcal{A}}$, where $\gamma(\varphi^{\mathcal{A}})$ implies φ.

Note that we do not need the unobservable control variable π used in Section 5.3.2, so this construction is simpler. □

The model checker finds, among the space of all possible deductive rules that could be applied, one that proves the property given the available verification conditions, if such exists.

Disclaimer: With these techniques, chances are that we will not be able to show termination, say, of the famous (open) "$3n+1$" *Collatz problem*, equivalent to proving

$$\varphi : \Box(n > 0 \rightarrow \Diamond\, n = 1)$$

for the following one-transition FTS, where n is an integer-valued system variable:[5]

$$\text{Transition } \tau_1 \text{ (just)} : \text{if } even(n) \text{ then } n' = n/2 \text{ else } n' = 3n+1 \ .$$

But chances also are that the correctness of actual systems does not depend on the (presumed) termination of such functions. Hopefully, the system designers are not assuming the termination of the Collatz problem while developing the design; and if they did, the design is probably wrong anyway, and bugs rather than proofs should be found (unless, of course, the designers can also solve the open problem itself).

6.4.3 Counterexample Generation

If φ is not $\mathcal{S}$-valid, the above procedure can only identify abstract counterexamples that are potential concrete ones. If we are given constrained ($\forall\exists$) transition relations $\tau^{\mathcal{A}}_{\forall\exists}$ (see Sections 3.5 and 4.9) we can also search for a concrete counterexample. This procedure uses upper bounds on the enabling conditions of all concrete fair transitions (which can default to $true^{\mathcal{A}}$).

Again, we present the algorithm for a product structure where each node n is labeled with an abstract assertion $\mu(n)$ rather than a single abstract state.

[5] Paul Erdös is said to have said, *"Mathematics is not yet ready for such problems."*

1. Label each edge in the $(\mathcal{S}^{\mathcal{A}}, T_{\neg\varphi})$ product graph with the $\forall\exists$-transitions that can be taken along it. That is, an edge (s_1, s_2) is labeled with τ if

$$\mu(n_1) \;\rightarrow\; \exists \mathcal{V}'_{\mathcal{A}}. \left(\tau^{\mathcal{A}}_{\forall\exists}(\mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}}) \wedge \mu'(n_2)\right)$$

   is propositionally valid. Note that the existential quantification is now finite. We call this the $\forall\exists$-*subgraph*, following Definition 3.5.1.[6]

2. In this $\forall\exists$-subgraph, find a fulfilling SCS reachable from an initial node.

3. Check that the SCS is "fully fair," as follows: we say that $\tau^{\mathcal{A}}_{\forall\exists}$ is *fully disabled* at a node n if $\mu(n) \subseteq \neg enabled^{+}\tau$. An SCS is fully fair if every just (resp. compassionate) transition is either taken in S (that is, it labels an edge in the $\forall\exists$-subgraph) or is fully disabled at some (resp. all) nodes in S. This is analogous to the DMC Definition 5.5.5.

4. Well-founded orders can also be used to identify counterexamples. The classic example is a program loop that forms an SCS S, where the condition for exiting the loop is at node n: all edges except one, $e_1 : \langle n, n_{\text{in}} \rangle$, can be labeled as executable. An edge $e_2 : \langle n, n_{\text{out}} \rangle$ will lead out of the SCS. We may be able to establish that if $\mu(n)$ holds, then either e_1 or e_2 can always be taken, leading to n_{in} or n_{out}. If we can show that S is terminating, then we can be sure that eventually e_2 will be taken, and n_{out} can be reached.

We say that a reachable, fulfilling and fully fair SCS is *fully adequate*.

Proposition 6.4.4 *A fully adequate SCS in the $(\mathcal{S}^{\mathcal{A}}, \neg\varphi^{\mathcal{A}})$-product graph contains a concrete counterexample to the $\mathcal{S}$-validity of φ.*

> **Proof:** (Outline) By induction, we construct the *concrete* counterexample computation $s_0, s_1, \ldots$ starting from an initial node and following the executable edges in the $\forall\exists$-subgraph, to reach the fully adequate SCS and then infinitely often traverse all its edges and visit all its nodes. Since the SCS is fulfilling and fair, this is a computation of $\mathcal{S}$ that satisfies $\neg\varphi$. □

[6]In practice, this subgraph can be generated by symbolic forward propagation of the $\forall\exists$ transitions from the initial condition.

Note that we assume that every just or fair concrete transition is represented by an abstract $\tau_{\forall\exists}^{\mathcal{A}}$ transition; otherwise, we could not guarantee that an SCS is fully fair.

The constrained $\forall\exists$ transition relation can, in general, be used to model check CTL* over the abstraction. Weak preservation then guarantees that model checked properties can be transferred to $\mathcal{S}$. The formulation and efficient implementation of such a model checker is left for future work.

6.4.4 Abstraction Generation Revisited

Consider again the abstraction generation procedure presented in Chapter 4. In Section 4.3, we propose a conservative generation of abstract systems, based on Lemma 3.4.6, where the fairness of an abstract transition is retained only if the enabling condition is exactly approximated. This approach lets us apply a standard finite-state model checker to the abstract system and transfer model checked properties to the concrete one.

A more aggressive approach also computes lower bounds on the enabling conditions of the other fair transitions, building a fairness table. If ranking functions are identified, a termination table is generated as well. We may then use an Extended Model Checker to model check the abstract system given the additional information, and, as before, transfer the resulting properties over to the concrete system.

The essential idea remains the same: pre-compute as much as possible, safely approximating the concrete system, and leave the combinatorics to a model checker, which can mix and match abstract transitions, enabling conditions, well-founded orders and SCS's.

6.5 Combining the Input

The information from two separate abstractions A_1 and A_2 can be easily combined into a new abstraction A, as follows:

- The set of system variables of A is the union of the system variables of A_1 and A_2.
- If $\tau_1^{\mathcal{A}}$ and $\tau_2^{\mathcal{A}}$ are the standard ($\exists\exists$) transitions for τ in each system, the transition for $\tau^{\mathcal{A}}$ in A is $\tau_1^{\mathcal{A}} \wedge^{\mathcal{A}} \tau_2^{\mathcal{A}}$.

  If $\tau_{\forall\exists 1}^{\mathcal{A}}$ and $\tau_{\forall\exists 2}^{\mathcal{A}}$ are the $\forall\exists$-transitions for τ, then $\tau_{\forall\exists}^{\mathcal{A}}$ in A is $\tau_{\forall\exists 1}^{\mathcal{A}} \vee^{\mathcal{A}} \tau_{\forall\exists 2}^{\mathcal{A}}$.
- The fairness tables are merged simply by disjoining the lower bounds (their union is also a lower bound).

- The two termination tables are merged by taking their union.[7]

Note that the new abstraction has, in general, smaller $\exists\exists$ transitions and larger $\forall\exists$ ones, and thus will satisfy more CTL* properties.

As usual, an OBDD representation can facilitate the process, specially if we have an OBDD that relates the assertions ("bits") that only appear in A_1 with those that only appear in A_2 (containing, e.g., invariants, or previously proved relationships between the respective assertions). If the combined system has a large number of system variables, finite-to finite-state abstraction can be performed—we must then be willing to throw away bits as well as collect them.[8]

Example 6.5.1 (Second termination table for BAKERY**)** *If we add*

$$\left\{ \begin{array}{ll} b^{N \geq y_1}: & N \geq y_1, \\ b^{N \geq y_2}: & N \geq y_2 \end{array} \right\}$$

we can construct a new termination table for Bakery featuring ranking functions

$$\delta_1 : N - y_1, \ \ \delta_2 : N - y_2$$

with the non-negative integers as their domain, as described in Table 6.5.1.

This table lets us prove

$$\varphi : \Diamond \Box \neg((\ell_0 \vee \ell_1) \wedge (m_0 \vee m_1)) \ \rightarrow \ \Diamond (y_1 > N)$$

To relate this table and the one in Example 6.1.5 (Figure 6.2), we can add the axiom

$$b^{N \geq \max} \leftrightarrow \left(b^{N \geq y_1} \vee b^{N \geq y_2} \right) \ .$$

[7] Techniques can be devised for eliminating redundancies from the resulting table, but we leave them for future work.

[8] A large number of bits can be particularly problematic for symbolic model checking. It should be interesting to see how the recently proposed satisfiability-based symbolic model checking [BCCZ98] performs on these combined abstractions.

Transition	*Ranking Functions*	*Precondition*	*Postcondition*	*Result*
$\ell_1^{\mathcal{A}}$ $(y_1 := y_2 + 1)$	$\langle \delta_1, \delta_1 \rangle$	$b^{N \geq y_1} \wedge b^{y_1 \leq y_2}$	$b^{N \geq y_1}$	$\prec$
$m_1^{\mathcal{A}}$ $(y_2 := y_1 + 1)$	$\langle \delta_1, \delta_1 \rangle$	$b^{N \geq y_1}$	*true*	$\preceq$
$\ell_1^{\mathcal{A}}$ $(y_1 := y_2 + 1$	$\langle \delta_2, \delta_2 \rangle)$	$b^{N \geq y_2}$	*true*	$\preceq$
$m_1^{\mathcal{A}}$ $(y_2 := y_1 + 1$	$\langle \delta_2, \delta_2 \rangle)$	$b^{N \geq y_2} \wedge \neg(b^{y_1 \leq y_2})$	$b^{N \geq y_2}$	$\prec$
$\begin{bmatrix} \ell_0, \ell_2, \ell_3, \\ m_0, m_2, m_3 \end{bmatrix}$	$\begin{bmatrix} \langle \delta_1, \delta_1 \rangle \\ \langle \delta_2, \delta_2 \rangle \end{bmatrix}$	$\begin{bmatrix} b^{N \geq y_1} \\ b^{N \geq y_2} \end{bmatrix}$	*true*	$\preceq$

Table 6.3: Second termination table for abstract BAKERY

6.6 Discussion

The common framework of abstraction we have presented lets us optimize the workings of verification tools such as STeP. The following components generate abstractions:

- Assertion-based abstraction generator (Chapter 4).
- Standard verification diagrams and rules (Section 5.1).
- GVD's and DMC (Sections 5.3 and 5.5).

For a given system, the verification conditions proved using one or more of the above methods can be collected into a common abstraction, even when the method fails. Since these first-order formulas can be expensive to prove, they should be reused as much as possible. A theorem proved *once* by the theorem-prover can be used many times by the model checker. Thus, our slogans are:

- *No wasted verification conditions,* and
- *Leave the combinatorics to the model checking tools.*

The framework we propose can also be used in an iterative refinement framework, following the process used in Figure 5.6. If E$_{\rm X}$MC fails and cannot produce a concrete counterexample, one of the following actions can be carried out by the user:

- Add a new element to the abstraction basis.

- Prove extra verification conditions for the current abstraction, improving the $\exists\exists$ or $\forall\exists$ abstraction. This includes improving the lower bounds $enabled^{-}(\tau)$.

- Provide new ranking functions δ_i and δ_j, with *pre* and *post* to go with them, which are added to the termination table.

Note that the search for counterexamples is dual to the standard abstraction refinement process: starting with a transition system that is most abstract but has no transitions, we add edges whenever they are known to exist.

6.7 Related Work

6.7.1 Finite-State Model Checking

There are many variants and optimizations of finite-state model checking algorithms (see Section 2.3) that could be adapted to EXMC. (For infinite-state model checking, see Section 7.3.) "On-the-fly" model checking algorithms, such as the one for CTL* presented in [BCG95] and the ones for LTL in SPIN [Hol91] and STeP construct only the portion of the state-space required by the given formula, rather than labeling the entire state-space.

Partial order reductions [GW91, Pel94] are used to avoid the state explosion that arises from the interleaving of independent transitions. The "partial model checking" of [And95] performs consecutive transformations of the system and its μ-calculus specification. Another approach to combat the state explosion is exploiting *symmetry* in the state-space [CFJ93, ES93].

6.7.2 Combining Abstractions

Hierarchical Verification Diagrams [BMS96] combine verification diagrams by taking their product, as automata. This approach has two main drawbacks compared to the one we propose: first, the combination does not capture all the interaction between the two abstractions. Second, we have seen that the control points of a verification diagrams are an artifact of the formula being proved; but when the product of two diagrams is computed, the number of new control configurations can grow exponentially, which may not have been necessary. In our approach, it is up to the model checker to explore the portion of the abstract state-space that is relevant to the *new* temporal property, independently of the particular properties that may have been proved before: the abstractions used to prove them are more useful.

On the other hand, the diagrams in [BMS96] are used for hierarchical verification and existential quantification over flexible variables, subjects that we do not address.

Reusing Abstractions: The authors of [BLO98] have recently proposed concretizing the reachable state-space of a failed ABA when proving invariants (private communication), using this as a lemma in a new attempted proof.

Combining Abstract Interpretations: A refinement-based method for combining abstract interpretations is presented in [DS97], starting with an over-approximated *control flow graph* of the program being verified. A *filter*, which constrains the set of computations of the current abstraction, is represented in temporal logic as an automaton or as a set of transitions. Universal properties can then be model checked relative to the set of filters available. This is implemented in the FLAVERS static analysis tool. Filters are applied to finite-state software abstractions in [DP98].

The relationship between model checking, program analysis and abstract interpretation is described in [Sch98, SS98]).

6.7.3 Fairness Diagrams

The *Fairness Diagrams* of [dAM96] present a diagram-based methodology for universal temporal properties that features refinement, well-founded orders and fairness considerations. A proof begins with an initial fairness diagram that directly corresponds to the system. This diagram is then transformed into one that corresponds directly to the specification, or which can be shown to satisfy it by purely algorithmic methods. A simulation rule that accounts for fairness justifies the transformations. An abstract system can be refined and transformed according to the simulation rule, independently of any temporal property, model checked, and refined again.

Doing justice to their name (no pun intended), fairness is represented quite generally in these diagrams, which can also be seen as abstractions. Fairness constraints on the system are combined with ranking functions to justify diagram constraints and generate new ones. Like DMC and GVD's, the methodology provides a (relatively) complete proof system.

The relationship between the formalisms of Fairness Diagrams, GVD's and DMC is discussed in [dAMSU97]. This paper also presents an alternative to the unfolding rule, based on Fairness Diagrams, which requires an additional set of assertions labeling the nodes of the SCS.

Compared to Fairness Diagrams, the approach proposed in this chapter attempts to isolate the information known about the system in a form that can be model checked, avoiding the direct construction of a diagram and the introduction of (unnecessary) abstract control variables.

In the following chapter, we discuss a broader range of work related to the main themes of this thesis.

Chapter 7

Related Work

There is a very large amount of research on abstraction, present and past, and we cannot possibly hope to include all of it here. Thus, we focus only on work that addresses themes similar to those in this thesis, and give representative samples for the rest, giving up (as one should do for most practical theorem proving applications) any claim of completeness.

Most methods that combine deductive and algorithmic verification through abstraction can be classified according to the following criteria:

- whether the abstraction is constructed *a priori* or or dynamically refined (*static* vs. *dynamic*);

- whether the abstraction generation is tailored specifically for the property of interest (*bottom-up* vs. *top-down*);

- whether the process is automatic or interactive.

Bottom-up methods operate independently of any particular formula to be verified: the resulting abstractions are meant to preserve a wide class of properties. Top-down methods use the formula to be verified to guide the construction of an abstraction that proves it. Focusing on a particular formula allows for coarser, less refined abstract systems, for which fewer properties hold, but which can be easier to construct and model check. As an example, the abstraction generation of Chapter 4 is mostly bottom-up, while the diagram-based methods discussed in Chapter 5 are top-down.

7.1 Static Abstractions

We now survey two classes of *static* methods: the first generates abstractions automatically using abstract interpretation (Sections 7.1.1), and includes the important special case of invariant generation (Section 7.1.2). The second checks the correctness of given abstractions (Section 7.1.3). The first approach is more automatic, while the second is more flexible.

7.1.1 Finding $\mathcal{A}$ Using Abstract Interpretation

Abstract interpretation [CC77] provides a general framework and a methodology for automatically producing abstract systems given a choice of the abstract domain $\Sigma_{\mathcal{A}}$. Most applications of classic abstract interpretation can be considered to be *bottom-up*, since they abstract the system independently of any particular property to be proved.

Abstract interpretation is usually meant to be *compositional* in the following sense:[1] each operation in the concrete program P is approximated as a corresponding abstract operation, producing an abstract program $\mathrm{P}^{\mathcal{A}}$. By construction, $\mathrm{P}^{\mathcal{A}}$ will be guaranteed to be a correct abstraction of $\mathcal{S}$. Furthermore, $\mathrm{P}^{\mathcal{A}}$ can be efficiently generated, since the abstraction process is local to each statement in P. As usual, properties that can be proved for $\mathrm{P}^{\mathcal{A}}$ will then carry over to P.

In our terminology, this approach avoids producing the abstract Kripke structure explicitly: the abstract program $\mathrm{P}^{\mathcal{A}}$ is symbolically executed to generate it instead. Chapter 4 can be understood as a simple attempt to perform generic abstract interpretation using theorem proving, compositionally in the boolean structure of the transition relations.

The classic abstract interpretation framework is intended (implicitly) for proving safety properties, and did not consider temporal logic or model checking. This is no longer the case: [LGS$^+$95] presents weakly- and strongly-preserving abstractions for the μ-calculus, showing how abstract interpretation can also be compositional over parallel composition. [CGL94] presents abstract interpretation for finite-state systems that is weakly-preserving for $\forall$CTL*, and notes that the resulting abstractions preserve all of CTL* when they are strongly preserving. [CIY95] defines a measure of optimality of abstractions, by which the largest possible set of properties is preserved. [DGG94] present abstract interpretation frameworks that preserve various fragments of CTL*. This is extended to the μ-calculus in [DGG97]. All of these frameworks weakly preserve CTL* by using two different abstract

[1] This use of the word is generally different from what is meant by "compositional verification."

transition relations for existential and universal path quantification. Chapter 3 presents a special case of such temporal abstract interpretation, extended to include fairness, which most related work does not consider.

The relationship between model checking and abstract interpretation continues to be the subject of much research, see e.g., the recent workshops [Bos97, Bos98] on the subject. For instance, [LV97] shows how verification methods in logic programming can be explained in terms of abstract interpretation.

Classic (and less classic) static program analysis for software systems based on abstract interpretation has been recently re-formulated in terms of model checking. [Sch98, SS98] describes how many program analysis techniques can be understood as the model checking of particular classes of abstractions. This work also addresses the combination and refinement of abstractions, as discussed in Section 6.7.2. [CC99] combines model checking and the approximated fixpoint analysis of abstract interpretation.

7.1.2 Invariant Generation

Invariant generation is an important special case of static analysis. In Section 4.8, we saw that the concretization of the reachable state-space of the abstract system is an invariant of the concrete one. Furthermore, any *over-approximation* of this state-space is also an invariant.

Abstract interpretation is used in [BBM97] to automatically generate invariants for general infinite-state systems. The abstract program $\mathrm{P}^{\mathcal{A}}$ is symbolically executed to find conservative approximations to the reachable state-space. Fixpoints of the abstract program express invariances that can then be concretized to invariants of the original system. *Widening*, a classic abstract interpretation technique, is used to speed up or ensure convergence of the abstract fixpoint operations, by performing safe over-approximations. The abstract domains that can be used to perform such computations include the assertion-based abstract domain itself, as well as set constraints, linear arithmetic, and polyhedra. These invariant generation methods have been implemented as part of STeP (see Section 2.7).

Using the same framework, [BBM97] also presents a top-down procedure for proving LTL safety properties. To prove φ, an *assertion graph* for φ is built, representing the computations that satisfy φ. The nodes in this graph then can then be strengthened to produce, in effect, a safety verification diagram that proves φ. The strengthening can be performed as a fixpoint computation over the graph. In the case of invariances, this is equivalent to finding an inductive assertion that implies the invariance to be proved (see Section 5.1).

Invariant generation using the assertion-based abstract domain is the focus of [GS97] (see Section 4.10).

7.1.3 Checking Preservation Using Theorem-Proving

System abstractions can always be constructed manually, and then proved correct. Typically, the correctness of the abstraction relation (the equivalent of Theorem 3.3.3) is established using theorem proving, while the abstract system itself is model checked. [Hun93] constructs an abstraction based on data independence (see Section 7.3.1) and modular verification. Deductive modular decomposition is used in [KL93] to reduce the correctness of a large system to that of smaller components that can be model checked. Abstraction is used in [RSS95] and [DF95] to obtain subgoals that can be model checked; the correctness of the abstraction is proved deductively.

Rules for proving simulation between systems are presented in [KMP94], as described in Section 3.6. Verification rules and (Generalized) Verification Diagrams are also an instance of static abstraction, as discussed in Chapter 5.

7.1.4 Generating $\mathcal{S}^{\mathcal{A}}$ Using Assertions and Theorem Proving

The procedure of Chapter 4 constructs a finite-state abstraction using decision procedures, and can be regarded as bottom-up and static. As described in Section 4.10, the approaches of [GS97] and [BLO98] are closely related, since they generate assertion-based abstractions as well. However, they are geared towards generating invariants and proving safety properties, so they do not consider fairness or the $\forall\exists$ transition relation.

A similar method for generating abstract systems is presented in [LS98], intended for the practical analysis and debugging of complex software systems. A set of control predicates defines the abstraction, which is generated using an approximated weakest precondition (*wpc*) operation as well as abstract interpretation of program operations. The resulting finite-state abstractions are model checked by SPIN [Hol91]. This approach uncovered bugs in the software for a NASA spacecraft controller.

The generation of assertion-based abstractions is given a top-down flavor by including in the basis the atomic subformulas of the temporal property φ being verified. Furthermore, for invariances, better abstractions can be obtained if the invariant to be proved is *assumed* when generating the abstraction [BLO98] (again, see Section 4.10).

7.1.5 Abstraction for Finite-state Systems

To finish this section, we note that there is much work on applying abstraction to purely *finite-state* systems as well, particularly large hardware systems. See, for instance, [Lon93, MHF98]. Abstracting from the finite-state case to an *infinite-state* abstract domain has proved useful here, namely, by using uninterpreted function symbols, by symbolically executing the system using a decidable logic, e.g., [BD94, JSD98, BBCZ98].

A method for automatic datapath abstraction is presented in [HB95], generalizing the notion of data-independence (see Section 7.3.1).

Automatic methods for eliminating variables from finite-state software specifications are presented in [BH97], depending on the property being verified. This reduces the number of variables in the system that is model checked.

7.2 Dynamic Abstractions: Finding $\mathcal{A}$ Through Refinement

We now review methods for constructing an abstract system $\mathcal{A}$ that are based on the general idea of *refining* an initial conservative approximation.

To prove φ for $\mathcal{S}$, an initial weakly-preserving finite-state abstraction $\mathcal{A}$ can be suggested. If model checking shows that $\mathcal{A} \models \varphi$, the verification is complete. However, if this model checking fails, $\mathcal{A}$ can be used as the starting point for producing a finer abstraction, $\mathcal{A}'$, which is also weakly-preserving for $\mathcal{S}$ but satisfies more properties. This process may be repeated until the property is proved (or, perhaps, disproved), as described in Figure 5.6.

Like the more static techniques of Section 7.1, these methods may be classified as bottom-up or top-down, depending on whether they consider a particular formula to be verified.

7.2.1 Bottom-up: Partition Refinement

Many bottom-up methods are reminiscent of simulation- and bisimulation-equivalence procedures, which are well-known in the case of finite-state systems (independently of abstraction).

A method for finding the coarsest bisimulation for a possibly infinite-state system (thus preserving CTL*) is presented in [BFH+92]. If two elements of an equivalence class are not bisimilar, the class is split according to *pre* and *post* operations on the transition relation. The procedure keeps track of accessible states as much as possible, to avoid spending time refining inaccessible portions of the state-space. This method is combined with abstract interpretation in [Fer93].

Refinement-based generation of a weakly-preserving $\forall$CTL* abstraction, i.e., simulation equivalence, is presented in [DGG93]. The details of the algorithm are restricted to finite-state systems. In [DGD+94], this method is combined with abstract interpretation, using OBDDs, for finite-state systems.

7.2.2 Top-Down Refinement

As described in Section 5.5, Deductive Model Checking can be classified as a dynamic, top-down refinement method. The refinement operations of DMC, based on pre- and post-conditions, are similar to those in the bottom-up [BFH+92] (see above). However, DMC uses the temporal tableau for $\neg\varphi$ as a starting point, includes fairness constraints, and tracks potential counterexamples.

As discussed in Section 6.7.3, *Fairness Diagrams* [dAM96] present an alternate dynamic refinement method that combines the top-down and bottom-up approaches. Fairness Diagrams are extended to hybrid systems in [dAKM97], and related to GVD's and DMC in [dAMSU97]. See [Sip98] for more on Deductive Model Checking, GVD's, and their application to the verification of real-time and hybrid systems.

Another top-down approach is presented in [DGH95], as a "truly symbolic" model checking procedure, analyzing the separation between data and control. A tableau-based procedure for $\forall$CTL generates the required verification conditions in a top-down, local manner, similarly to DMC.

7.2.3 Failure-based Refinement

Refining abstractions based on the failure of model checking for the abstract system has been investigated in more detail in the case of finite-state systems, where it can be performed automatically.

[Kur94] presents error trace analysis for finite-state systems: if model checking the abstraction produces an abstract counterexample $\pi_{\mathcal{A}}$, the concrete system is executed along corresponding concrete paths. If a concrete counterexample is found, the property is disproved; otherwise, the failure of the counterexample can be used to further restrict the abstraction, producing a new one for which $\pi_{\mathcal{A}}$ is not a computation. [BSV93] present a BDD-based procedure to carry out refinement: a conservative abstraction is progressively refined, at each step ruling out a set of false counterexamples.

An automatic BDD-based method for μ-calculus symbolic model checking (which includes CTL*), for finite-state systems, is presented in [Par97, PH97]. Subformulas are over- and under-approximated, depending on their polarity, to control the BDD size. The overall procedure preserves positive results, and a dual procedure can be made to preserve negative ones. If model checking fails, the set-difference between the model checking result and the set of initial states can be used as a starting point to refine the abstraction. The computations for individual formulas can be made more precise, until the desired property can be established.

A similar approximated BDD-based symbolic model checking procedure is presented in [KDG95], this time based on the abstract interpretation framework of [Dams96], but does not perform refinement. Such abstractions do not change the state-space of the system, but instead approximate the transition relation to produce smaller BDDs.

7.3 Model Checking for Infinite-State Systems

Many ideas behind the algorithmic model checking of decidable classes of infinite-state systems, including real-time and hybrid systems, are closely related to abstraction and refinement. Evidently, these methods can be considered top-down, since they model check particular temporal properties.

For infinite-state systems, state-space exploration must be adapted if it is to remain an algorithmic procedure. For many classes of infinite-state systems for which model checking is decidable the exploration of a finite quotient of the state-space is sufficient, e.g., for certain classes of real-time and hybrid systems [ACH+95]. For others, the right choice of abstract assertion language ensures convergence of fixpoint operations [HH95, KMM+97, BGP97]. Reductions for hybrid systems that explicitly use abstraction are presented in [OSY94]. The underlying principles are explored for the case of safety properties in [EN98].

A number of "local model checking" procedures for infinite-state systems e.g., [BS92], can be said to fall somewhere in between deductive rules and model checking, similarly to

GVD's. [Lev98] uses an abstract constraint language to model check first-order μ-calculus formulas for possibly infinite-state systems. The local μ-calculus model checking algorithm for real-time systems of [SS95] can be regarded as a variant of the DMC procedure, specialized to real-time systems: it refines a finite representation of an infinite product graph, splitting nodes to satisfy clock constraints that depend on the property and system being checked.

In all of these cases, some form of abstraction is used, to make the infinite-state space manageable. Often, a finite approximation or quotient of the infinite state-space is explored instead. Since a useful abstraction does not usually satisfy all the properties of the original system, abstractions must often be refined before the property of interest can be proved.

[DW95, Won95] uses abstract interpretation to establish general safety properties of real-time systems. A general algorithm is presented that performs forwards and backwards over- and under-approximations, This procedure is then specialized to the case of real-time systems.

A method for "model checking" finite abstractions of formulas over infinite domains is presented in [Jac95]. A class of exact abstractions allows concrete theorems to be proved or disproved at the abstract level, while approximations may produce false counterexamples. The Nitpick system [JD96, Jac96], an example of a "lightweight" verification system (see Section 8.2), uses similar ideas to detect errors in software systems. Constraints are used to reduce a potentially infinite number of configurations to a finite number that can be automatically checked. Temporal logic is not explicitly used in this work—implicitly, it is invariants that are checked.

7.3.1 Data Independence

The often-used notion of *data independence* for reactive systems is defined in [Wol86]. Briefly, a system is data-independent if the particular values of the (possibly infinite-state) data that it manipulates are not relevant to the behavior of the system. Thus, for instance, a protocol that transmits messages may be proved correct regardless of the type of messages it transmits. Similarly, a hardware system that moves data between different registers but does not change the data or branch depending on its value is data-independent as well.

Data-independent systems are described as "essentially finite-state." The same can be said for systems that have finite-state assertion-based abstractions that do not use well-founded orders, a class of systems that includes the data-independent ones: only finitely

Method	Refinement?	uses φ?	automatic?
Static Analysis and Invariant Generation (Abstract Interpretation)	static	bottom-up	automatic
Assertion Graphs (Intermediate Assertions)	static	top-down	automatic
(Generalized) Verification Diagrams	static	top-down	interactive
Deductive Model Checking	dynamic	top-down	interactive
Fairness Diagrams	dynamic	[both]	interactive
Partition Refinement	dynamic	[both]	[both]
Infinite-state Model Checking	dynamic	top-down	[both]

Table 7.1: Classification of some abstraction methods

many distinctions concerning the data are made, which can be captured by a finite number of assertions. (This distinction is relative to the particular property being verified—programs can be said to be data-independent with respect to some properties, but not others.)

In contrast, note that the use of well-founded orders makes it possible to reason about unbounded sequences of operations on data of unbounded size.

Issues of data, control, and data-independence are discussed in, for instance, [DGH95, HB95]. Wolper points out how propositional logics of programs [FL79, KT89] isolate program properties from first-order complexity. Our abstract systems can also be analyzed using such logics.

7.4 Combined Methods

Table 7.1 summarizes the classification of methods in this chapter. For general infinite-state systems, the automatic methods are incomplete, and the complete methods are interactive.

There are many ways in which theorem proving, model checking, invariant generation and abstraction can be combined. The STeP system (see Section 2.7) integrates all of these components. General-purpose theorem provers such as HOL and the Boyer-Moore prover have been equipped with OBDD's. The PVS system provides an OBDD-based procedure for the evaluation of μ-calculus formulas that can be used to perform symbolic model checking [RSS95, HS96, ORR$^+$96].

[DF95] applies abstraction and error trace analysis to infinite-state systems. The abstract system is generated automatically given a *data abstraction* that maps concrete variables and functions to abstract ones. The abstraction is based on a simulation relation, so it is weakly preserving for $\forall$CTL*. If an abstract counterexample is found that does not correspond to a concrete one, an *assumption* that excludes this counterexample is generated. This is a temporal formula that is expected to hold of the concrete system $\mathcal{S}$. The model checker, which can take such assumptions into account, is then again used to check the original property. The process is iterated until a concrete counterexample is found, or model checking succeeds under a given set of assumptions. At this point, the assumptions can be proved over the concrete system, using deductive verification (theorem proving). If they hold, the proof is complete. The assumptions are provided by the user, and are often simple invariants which are easy to prove, deductively, over $\mathcal{S}$.

[RS99] presents a framework that combines abstraction, abstraction refinement and theorem proving in a way similar to that espoused in this thesis, specialized to the case of invariants, where the different components are treated as "black boxes." After an assertion-based abstraction is generated (using the method of [GS97]), abstract counterexamples are analyzed to refine the abstraction or produce a concrete counterexample, in an iterative process similar to that of Figure 5.6. Conjectures generated during the refinement process are given to the theorem prover, and the process repeated.

Chapter 8

Conclusions

We have shown how a number of deductive-algorithmic verification methods share the common feature of justifying, deductively, the correctness of an abstraction that can then be model checked, and presented ways in which these abstractions can be incrementally constructed and combined. In our framework, theorem proving is used to reason about relations over the (possibly unbounded) data values in the system. The results are expressed as a finite-state abstraction, so that algorithmic model checking can perform the required state-space exploration and combinatorial reasoning, which remains hidden to the user.

By "compiling" all the information produced by deductive methods into an abstraction, we reuse the results of theorem proving as much as possible, maximizing the benefits of algorithmic tools and minimizing the need for user interaction. The $\mathrm{E_XMC}$ procedure of Chapter 6 is intended to maximize the number of temporal properties that can be automatically model checked once such abstractions are built, taking advantage of the extra precision offered by ranking functions and bounds on fair transitions.

Our framework leads naturally to an *abstraction refinement* methodology, where the search for a suitable abstract system is iterative, guided by the user (see Figure 5.6). The tools we propose can provide quick user feedback during such a process.

Dynamic methods such as Deductive Model Checking (Section 5.5) and Fairness Diagrams (see Section 6.7.3) can be viewed as abstraction refinement, and interleave the model checking and the theorem proving. However, they expose the combinatorial complexity to the user, making them less directly applicable to larger systems and complex properties. On the other hand, they allow more precision and finer distinctions to be made. The same can be said for the static method of Generalized Verification Diagrams, which can force the user to explicitly consider a large number of points in the abstract state-space and a large number of SCS's. The approach we propose can help these methods scale up, leaving the combinatorial reasoning to automatic tools. Note, however, that graphical formalisms such as GVD's and DMC can still be used for the essential (and unavoidable) user interaction.

8.1 Future Work

All of the algorithms we present are, of course, open to improvement, and the practical implications of our framework remain to be explored. It is very desirable to make further improvements to the approximation procedures of Chapter 4.

Efficient implementations for the Extended Model Checking of Chapter 6 should also be developed, perhaps adapting existing finite-state model checking tools. The complexity of this finite-state procedure should be related to the complexity of standard model checking problems as well. Other extensions include adapting the $\mathrm{E_XMC}$ procedure to all of CTL*.

8.1.1 Finite and Infinite Domains

This thesis focuses on abstraction from the infinite-state to the finite-state. Clearly, the same methods can also be used to abstract from finite-state to finite-state (fewer bits). Furthermore, the general techniques we present can be adapted to assertion languages other than purely propositional logic, and the use of more powerful assertion languages at the abstract level should be investigated.

This will support, for instance, finite- to infinite-state abstractions, which can be particularly useful in the case of complex hardware, as mentioned in Section 7.1.5. It will also allow infinite- to infinite-state abstractions, where the abstract domain features a more tractable assertion language amenable to model checking and other automated tools (c.f. invariant generation, see Section 7.1.2).

8.1.2 Parameterized Systems

The techniques we present do not address the case of *parameterized systems*, which compose an unbounded, parameterized number of processes. Deductive methods are especially well-suited for verifying such systems, in an interactive setting. The relationship between the methods we propose and those used for parameterized system verification, e.g., [CGB89, KM89, WL89, MP90, LHR97, Nam98] should be explored.

8.1.3 Modular Specification and Compositional Verification

Like abstraction, modular specification and compositional verification (widely understood) can play a very important role in fighting the state-explosion problem (for a small sample of work in the area, see [dRLP98]). The interaction between compositional verification and abstraction for the finite-state is studied in [Lon93]; these methods should be extended and adapted to the infinite-state case.

Modularity and abstraction for infinite-state systems are addressed in [MCF$^+$98, KP98]. Encouraging preliminary results relevant to this thesis are presented in [MCF$^+$98]. A detailed STeP specification of the *steam boiler* hybrid system case study [ABL96] was debugged by generating finite-state abstractions that were then model checked. The abstraction procedure from Chapter 4 eliminated the real-time and hybrid portions of the system to yield finite-state (weakly-preserving) abstractions. Since the system was modularly specified, individual modules could be abstracted as well.

8.2 Mediumweight Formal Methods

Two frequently cited obstacles to the application of formal methods to real-world system design is their cost and the time it takes to apply them. Often, the product deadline is much closer than the time it would take to specify and verify it.

In [JW96], Jackson and Wing advocate *lightweight formal methods*, which are characterized by *partiality* in language, modeling, analysis and composition. In order to be more automatic and usable, such methods should be willing to use less expressive but more tractable specification languages, be able to analyze partial specifications of large systems, and give a higher priority to the finding of bugs rather than the construction of proofs.

The pure, unadulterated application of theorem proving can be *heavyweight* and, when not in the hands of an experienced user, expensive and slow. On the other hand, invariant generation and static analysis tools are lightweight, efficiently proving a limited class of properties. Combinations of abstraction, model checking, refinement and modularity can result in *mediumweight* methods, which can prove more complex properties more quickly with a moderate amount of user interaction.

Even though design errors should be quickly found, the process of finding them should facilitate the proof of system properties (with the caveats of Section 1.2). Proven properties can focus the attention on other conjectured properties and aspects of the system where bugs may be found. Thus, while initial negative results can help debug the system, positive results will establish simple properties that will be useful in more complex, global falsification, verification, debugging and proofs.

We hope that this thesis is a contribution towards this goal. In particular, the methods we present can be used for a faster iteration of the basic abstract-model check-refine loop, so the right abstraction, or a counterexample, can quickly be found.

Bibliography

[ABL96] Jean-Raymond Abrial, Egon Boerger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicholas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AH96] Rajeev Alur and Thomas A. Henzinger, editors. *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1996.

[And95] Henrik Reif Andersen. Partial model checking. In *Proc. 10th IEEE Symp. Logic in Comp. Sci.*, pages 398–406, 1995.

[BBC+95] Nikolaj S. Bjørner, Anca Browne, Eddie S. Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[BBC+96] Nikolaj S. Bjørner, Anca Browne, Eddie S. Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [AH96], pages 415–418.

[BBCZ98] Sergey Berezin, Armin Biere, Edmund M. Clarke, and Yunshan Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Gopalakrishnan and Windley [GW98], pages 369–386.

[BBLS92] Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. Property preserving simulations. In *Proc. 4^{th} Intl. Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 260–273. Springer-Verlag, 1992.

[BBM97] Nikolaj S. Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997. Preliminary version appeared in 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589–623, Springer-Verlag, 1995.

[BCCZ98] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. Technical report, Carnegie-Mellon University, 1998. To appear.

[BCDM86] Michael C. Browne, Edmund M. Clarke, David L. Dill, and Bud Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, 35(12):1035–1044, December 1986.

[BCG88] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, July 1988.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL*. In *Proc. 10th IEEE Symp. Logic in Comp. Sci.*, pages 388–397, 1995.

[BCJM96] Sergey Berezin, Edmund M. Clarke, Somesh Jha, and Will Marrero. Model checking algorithms for the μ-calculus. Technical Report CMU TR CMU-CS-96-180, School of Computer Science, Carnegie-Mellon University, September 1996.

[BCM$^+$92] Jerry R. Burch, Edmund M. Clarke, Ken L. McMillan, David L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In Dill [Dil94], pages 68–80.

[BDL96] Clark Barrett, David L. Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *1st Intl. Conf. on Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, November 1996.

[Ber98] Sergey Berezin. Personal communication. `http://www.cs.cmu.edu/~berez/`, 1998.

[BFH$^+$92] Ahmed Bouajjani, J-C. Fernandez, Nicholas Halbwachs, P. Raymond, and Christophe Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BGP97] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In Grumberg [Gru97], pages 400–411.

[BH97] Ramesh Bharadwaj and Constance Heitmeyer. Model checking complete requirements specifications using abstraction. Technical Report NRL/MR/5540–97-7999, Naval Research Laboratory, Washington, DC, November 1997.

[Bie72] Ambrose Bierce. *The Collected Works of Ambrose Bierce*. Citadel Press, 1972.

[Bjø98a] Nikolaj S. Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Computer Science Department, Stanford University, November 1998.

[Bjø98b] Nikolaj S. Bjørner. Symbolic temporal tableaux. Draft manuscript, Computer Science Department, Stanford University, 1998. Implemented in STeP.

[BLM97] Nikolaj S. Bjørner, Uri Lerner, and Zohar Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Intl. Conf. on Temporal Logic*. Kluwer, 1997. To appear.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, Boston, MA, 1988.

[BMS95] Anca Browne, Zohar Manna, and Henny B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 484–498. Springer-Verlag, 1995.

[BMS96] Anca Browne, Zohar Manna, and Henny B. Sipma. Hierarchical verification using verification diagrams. In 2^{nd} *Asian Computing Science Conf.*, volume 1179 of *Lecture Notes in Computer Science*, pages 276–286. Springer-Verlag, December 1996.

[BMSU97] Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Deductive verification of real-time systems using STeP. In *4th Intl. AMAST Workshop on Real-Time Systems*, volume 1231 of *Lecture Notes in Computer Science*, pages 22–43. Springer-Verlag, May 1997.

[BMSU98] Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. Technical report, Computer Science Department, Stanford University, March 1998.

[BMSU99] Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Deductive verification of real-time systems using STeP. Technical report, Computer Science Department, Stanford University, January 1999. To appear in *Theoretical Computer*

Science. Preliminary version appeared in *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[Bos97] Annalisa Bossi, editor. *International Workshop on Verification, Model Checking and Abstract Interpretation*, October 1997.

[Bos98] Annalisa Bossi, editor. *2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, September 1998.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proc. of the* 27^{th} *ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry91] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[BS92] Julian C. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96(1):157–174, April 1992.

[BSU97] Nikolaj S. Bjørner, Mark E. Stickel, and Tomás E. Uribe. A practical integration of first-order reasoning and decision procedures. In *Proc. of the* 14^{th} *Intl. Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Computer Science*, pages 101–115. Springer-Verlag, July 1997.

[BSV93] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In Courcoubetis [Cou93], pages 29–40.

[CC77] Patrick Cousot and Rhadia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 238–252. ACM Press, 1977.

[CC99] Patrick Cousot and Rhadia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1), 1999. To appear. Special issue on Automated Software Analysis.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CFJ93] Edmund M. Clarke, T. Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [Cou93], pages 450–462.

[CFZ95] Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. Hybrid decision diagrams. Overcoming the limitations of MTBDDs and BMDs. In *IEEE/ACM Intl. Conference on Computer-Aided Design*, pages 159–163, November 1995.

[CG87] Edmund M. Clarke and Orna Grumberg. Research on automatic verification of finite-state concurrent systems. In J.F. Traub, B.J. Grosz, B.W. Lampson, and N.J. Nilsson, editors, *Annual Review of Computer Science*, volume II, pages 269–290. Annual Reviews, 1987.

[CGB89] Edmund M. Clarke, Orna Grumberg, and Michael C. Browne. Reasoning about networks with many finite state processes. *Information and Computation*, 81:13–31, 1989.

[CGH94] Edmund M. Clarke, Orna Grumberg, and K. Hamaguchi. Another look at LTL model checking. In Dill [Dil94], pages 415–427.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CIY95] Rance Cleaveland, Purush Iyer, and Daniel Yankelevich. Optimality in abstractions of model checking. In *Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 51–63. Springer-Verlag, September 1995.

[Cou93] Costas Courcoubetis, editor. *Proc.* 5^{th} *Intl. Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[CU98] Michael A. Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [HV98], pages 293–304.

[dAKM97] Luca de Alfaro, Arjun Kapur, and Zohar Manna. Hybrid diagrams: A deductive-algorithmic approach to hybrid system verification. In *14th Symposium on Theoretical Aspects of Computer Science*, volume 1200 of *Lecture Notes in Computer Science*, pages 153–164, February 1997.

[Dams96] Dennis R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, July 1996.

[dAM96] Luca de Alfaro and Zohar Manna. Temporal verification by diagram transformations. In Alur and Henzinger [AH96], pages 287–299.

[dAMSU97] Luca de Alfaro, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Visual verification of reactive systems. In *Third Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 334–350. Springer-Verlag, April 1997.

[DF95] Jürgen Dingel and Thomas Filkorn. Model checking of infinite-state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In Wolper [Wol95], pages 54–69.

[DGD+94] Dennis R. Dams, Rob Gerth, G. Döhmen, R. Herrmann, Peter Kelb, and H. Pargmann. Model checking using adaptive state and data abstraction. In Dill [Dil94], pages 455–467.

[DGG93] Dennis R. Dams, Orna Grumberg, and Rob Gerth. Generation of reduced models for checking fragments of CTL. In Courcoubetis [Cou93], pages 479–490.

[DGG94] Dennis R. Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$ECTL*, CTL*. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*, pages 573–592, June 1994.

[DGG97] Dennis R. Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[DGH95] Werner Damm, Orna Grumberg, and Hardi Hungar. What if model checking must be truly symbolic. In *First Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, May 1995.

[Dil94] David L. Dill, editor. *Proc.* 6^{th} *Intl. Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Dil96] David L. Dill. The Murφ verification system. In Alur and Henzinger [AH96], pages 390–393.

[dNV95] Rocco de Nicola and Frits W. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, March 1995.

[DP90] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[DP98] Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In *Proc. of the 6th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, November 1998.

[dRLP98] Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference. COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[DS97] Matthew B. Dwyer and David A. Schmidt. Limiting state explosion with filter-based refinement. In Bossi [Bos97].

[DW95] David L. Dill and Howard Wong-Toi. Verification of real-time systems by successive over and under approximation. In Wolper [Wol95], pages 409–422.

[EH86] E. Allen Emerson and Joseph Y. Halpern. 'Sometimes' and 'not never' revisited: On branching time versus linear time. *Journal of the ACM*, 33:151–178, 1986.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers (North-Holland), 1990.

[EN98] E. Allen Emerson and Kedar S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. 13th IEEE Symp. Logic in Comp. Sci.*, pages 70–80. IEEE Press, 1998.

[ES93] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In Courcoubetis [Cou93], pages 463–478.

[Fer93] J.C. Fernandez. Abstract interpretation and verification of reactive systems. In Springer-Verlag, editor, *Proc. 3rd Intl. Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 60–71, 1993.

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comp. Sys. Sci.*, 18:194–211, 1979.

[FMS98] Bernd Finkbeiner, Zohar Manna, and Henny B. Sipma. Deductive verification of modular systems. In de Roever et al. [dRLP98], pages 239–275.

[GM93] Michael Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[Gru97] Orna Grumberg, editor. *Proc. 9^{th} Intl. Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science.* Springer-Verlag, June 1997.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In Grumberg [Gru97], pages 72–83.

[GW91] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *Proc. 6th IEEE Symp. Logic in Comp. Sci.*, pages 406–415. IEEE Computer Society Press, 1991.

[GW98] Ganesh Gopalakrishnan and Phillip Windley, editors. *2nd Intl. Conf. on Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, November 1998.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HB95] Ramin Hojati and Robert K. Brayton. Automatic datapath abstraction in hardware systems. In Wolper [Wol95], pages 98–113.

[Hen96] Thomas A. Henzinger. Some myths about formal verification. *ACM Computing Surveys*, 28(A):4, December 1996. Position statement for the working groups on formal methods and concurrency, ACM workshop on Strategic Directions in Computing, June 1996.

[HH95] Thomas A. Henzinger and Pei-Hsin Ho. HyTech: The Cornell hybrid technology tool. In *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 265–293. Springer-Verlag, 1995.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Engelwood Cliffs, NJ, 1991.

[Hol98] Gerard J. Holzmann. On checking model checkers. In Hu and Vardi [HV98], pages 61–70.

[HS96] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe*, pages 662–681, March 1996.

[Hun93] Hardi Hungar. Combining model checking and theorem proving to verify parallel processes. In Courcoubetis [Cou93], pages 154–165.

[HV91] Joseph Y. Halpern and Moshe Y. Vardi. Model checking vs. theorem proving: A manifesto. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Essays in Honor of John McCarthy*, pages 151–176. Academic Press, 1991.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Proc. 10^{th} Intl. Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1998.

[Jac95] Daniel Jackson. Abstract model checking of infinite specifications. In *Formal Methods Europe*, October 1995.

[Jac96] Daniel Jackson. Nitpick: A checkable specification language. In *Workshop on Formal Methods in Software Practice*, January 1996.

[JD96] Daniel Jackson and Craig A. Damon. Nitpick reference manual. Technical report, Carnegie-Mellon University, 1996.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, pages 111–119, January 1987.

[JSD98] Robert B. Jones, Jens U. Skakkebæk, and David L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In Gopalakrishnan and Windley [GW98], pages 2–17.

[JW96] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[KDG95] Peter Kelb, Dennis Dams, and Rob Gerth. Practical symbolic model checking of the full μ-calculus using compositional abstractions. Technical Report 95/31, Eindhoven University of Technology, The Netherlands, October 1995.

[KL93] Robert P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In Courcoubetis [Cou93], pages 166–179.

[KM89] Robert P. Kurshan and Ken L. McMillan. A structural induction theorem for processes. In *Proceedings of the Eight Annual Symposium on Principles of Distributed Computing.* ACM Press, 1989.

[KMM+97] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich assertional languages. In Grumberg [Gru97], pages 424–435.

[KMMP93] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A decision algorithm for full propositional temporal logic. In Courcoubetis [Cou93], pages 97–109.

[KMP94] Yonit Kesten, Zohar Manna, and Amir Pnueli. Temporal verification of simulation and refinement. In Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rosenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 273–346. Springer-Verlag, 1994.

[KMP98] Yonit Kesten, Zohar Manna, and Amir Pnueli. Verification of clocked and hybrid systems. In Grzegorz Rozenberg and Frits W. Vaandrager, editors, *Embedded Systems*, pages 4–73. Springer, Heidelberg, 1998.

[Knu77] Donald E. Knuth. Notes on the van Emde Boas construction of priority queues: An instructive use of recursion. Informally distributed course notes, March 1977.

[Koz83] Dexter C. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[KP98] Yonit Kesten and Amir Pnueli. Modularization and abstraction: The keys to practical formal verification. In *Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 54–71, August 1998.

[KT89] Dexter C. Kozen and Jerzy Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, 1989.

[Kup95] Orna Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, Technion—Israel Institute of Technology, Haifa, Israel, June 1995.

[Kur86] Robert P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121-861010-33, Bell Labs, 1986.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[Lam95] Leslie Lamport. Proving possibility properties. Technical Report 137, Digital Equipment Corporation, Systems Research Center, July 1995.

[Lev98] Francesca Levi. Abstract model checking by constraints abstraction. In Bossi [Bos98].

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[LHR97] D. Lesens, Nicholas Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In 24^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 346–357, 1997.

[Lon93] David E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July 1993.

[LS98] Michael Lowry and Mahadevan Subramaniam. Abstraction for analytic verification of concurrent software systems. In *Symposium on Abstraction, Reformulation, and Approximation*, May 1998.

[LV97] Giorgio Levi and Paolo Volpe. A reconstruction of verification techniques by abstract interpretation. In Bossi [Bos97].

[MBB+98] Zohar Manna, Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Mark Pichora, Henny B. Sipma, and Tomás E. Uribe. An update on STeP: Deductive-algorithmic verification of reactive systems. In *Tool Support for System Specification, Development and Verification*, pages 87–91. Christian-Albrechts-Universitat, Kiel, June 1998. Full version to appear in LNCS.

[MBSU98] Zohar Manna, Anca Browne, Henny B. Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. In Armando Haeberer, editor, *AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, December 1998.

[MCF+98] Zohar Manna, Michael A. Colón, Bernd Finkbeiner, Henny B. Sipma, and Tomás E. Uribe. Abstraction and modular verification of infinite-state reactive systems. In Manfred Broy, editor, *Requirements Targeting Software and Systems Engineering (RTSE)*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To appear.

[McG95] Hugh McGuire. *Two Methods for Checking Formulas of Temporal Logic*. PhD thesis, Computer Science Department, Stanford University, Stanford, CA, 1995.

[McM93] Ken L. McMillan. *Symbolic Model Checking*. Kluwer Academic Pub., 1993.

[MD93] Ralph Melton and David L. Dill. *Murphi Annotated Reference Manual*. Stanford University, November 1993.

[MHF98] Sela Mador-Haim and Limor Fix. Input elimination and abstraction in model checking. In Gopalakrishnan and Windley [GW98], pages 304–320.

[MP90] Zohar Manna and Amir Pnueli. An exercise in the verification of multi-process programs. In W.H.J. Feijen, A.J.M van Gasteren, David Gries, and Jayadev Misra, editors, *Beauty is Our Business*, pages 289–301. Springer-Verlag, 1990.

[MP91a] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

[MP91b] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[MP93] Zohar Manna and Amir Pnueli. A temporal proof methodology for reactive systems. In *NATO ASI Series, Series F: Computer and System Sciences*, volume 118 of *Lecture Notes in Computer Science*, pages 287–323. Springer-Verlag, 1993.

[MP94] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In M. Hagiya and John C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765. Springer-Verlag, 1994.

[MP95a] Zohar Manna and Amir Pnueli. Clocked transition systems. In *Proc. of the Intl. Logic and Software Engineering Workshop*, August 1995. Beijing, China.

[MP95b] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[MP96] Zohar Manna and Amir Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University, April 1996.

[MS98] Zohar Manna and Henny B. Sipma. Deductive verification of hybrid systems using STeP. In T.A. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 305–318. Springer-Verlag, 1998.

[Nam98] Kedar S. Namjoshi. *Ameliorating the State Explosion Problem.* PhD thesis, Department of Computer Science, University of Texas at Austin, 1998.

[ORR+96] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [AH96], pages 411–414.

[OSR93] Sam Owre, Natarajan Shankar, and John M. Rushby. User guide for the PVS specification and verification system (draft). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1993.

[OSY94] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In Dill [Dil94], pages 81–94.

[Par97] Abelardo Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems.* PhD thesis, University of Colorado, Boulder, Colorado, 1997.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In Dill [Dil94], pages 377–390.

[PH97] Abelardo Pardo and Gary Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In Grumberg [Gru97], pages 12–23.

[PK98] Amir Pnueli and Yonit Kesten. Verification by finitary abstraction. In *Fourth Intl. SPIN Workshop*, November 1998. Invited talk.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pages 46–57. IEEE Computer Society Press, 1977.

[Pra80] Vaughan R. Pratt. A near-optimal method for reasoning about action. *Journal of Computer and System Sciences*, 20(2):231–254, 1980.

[PS96] Amir Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In Alur and Henzinger [AH96], pages 184–195.

[QS82] J.P. Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and Ugo Montanari, editors, *Intl. Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

[RS99] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *5th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science. Springer-Verlag, March 1999. To appear.

[RSS95] Sreeranga Rajan, Natarajan Shankar, and Mandayam K. Srivas. An integration of model checking with automated proof checking. In Wolper [Wol95], pages 84–97.

[Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-07, Computer Science Laboratory, SRI International, November 1993. Companion to Federal Aviation Administration (FAA) Digital Systems Validation Handbook.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In 25^{th} *ACM Symp. Princ. of Prog. Lang.*, 1998.

[SdRG89] F. A. Stomp, Willem-Paul de Roever, and Rob T. Gerth. The μ-calculus as an assertion language for fairness arguments. *Inf. and Comp.*, 82:278–322, 1989.

[Sip98] Henny B. Sipma. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems.* PhD thesis, Computer Science Department, Stanford University, December 1998.

[SS95] Oleg V. Sokolsky and Scott A. Smolka. Local model checking for real-time systems. In Wolper [Wol95], pages 211–224.

[SS98] David A. Schmidt and Bernhard Steffen. Program analysis *as* model checking of abstract interpretations. In Giorgio Levi, editor, *Proc. 5th Static Analysis Symposium*, Lecture Notes in Computer Science, September 1998.

[SUM99] Henny B. Sipma, Tomás E. Uribe, and Zohar Manna. Deductive model checking. To appear in *Formal Methods in System Design*, 1999. Preliminary version appeared in *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219, 1996.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers (North-Holland), 1990.

[vG90] Robert J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Vrije Universiteit te Amsterdam, The Netherlands, 1990.

[VW86] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comp. Sys. Sci.*, 32:183–221, 1986.

[WL89] Pierre Wolper and V. Lovingfosse. Verifying properties of large sets of processes with network invariants. In Joseph Sifakis, editor, *CAV'89: Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1989.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. Princ. of Prog. Lang.*, pages 184–193, 1986.

[Wol95] Pierre Wolper, editor. *Proc. 7^{th} Intl. Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1995.

[Won95] Howard Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Computer Science Department, Stanford University, March 1995. Tech. Report CS-TR-95-1546.

Index of Symbols

Index