Approximation and Refinement Techniques for Hard Model-Checking Problems

by

Mihaela Bobaru (née Gheorghiu)

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Approximation and Refinement Techniques for Hard Model-Checking Problems

Mihaela Bobaru (née Gheorghiu)

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2009

Formal verification by model checking verifies whether a system satisfies some given correctness properties, and is intractable in general. We focus on several problems originating from the usage of model checking and from the inherent complexity of model checking itself. We propose approximation and iterative refinement techniques and demonstrate that they help in making these problems tractable on practical cases. Vacuity detection is one of the problems, relating to the trivial satisfaction of properties. A similar problem is query solving, useful in model exploration, when properties of a system are not fully known and are to be discovered rather than checked. Both of these problems have solution spaces structured as lattices and can be solved by model checking using those lattices. The lattices, in the most general formulation of these problems, are too complex to be implemented efficiently. We introduce a general approximation framework for model checking with lattices and instantiate this framework for the two problems, leading to algorithms and implementations that can obtain efficiently partial answers to the problems. We also introduce refinement techniques that consider incrementally larger lattices and compute even the partial answers gradually, to further abate the size explosion of the problems. Another problem we consider is the state-space explosion of model checking. The size of system models is exponential in the number of state variables and that renders model checking intractable. We consider systems composed of several components running concurrently. For such systems, compositional verification checks components individually to avoid composing an entire system. Model checking an individual component uses

assumptions about the other components. Smaller assumptions lead to smaller verification problems. We introduce iterative refinement techniques that improve the assumptions generated by previous automated approaches. One technique incrementally refines the interfaces between components in order to obtain smaller assumptions that are sufficient to prove a given property. The smaller assumptions are approximations of the assumption that would be obtained without our interface refinement. Another technique computes assumptions as abstractions of components, as an alternative to current approaches that learn assumptions from counterexamples. Our abstraction refinement has the potential to compute smaller nondeterministic assumptions, in contrast to the deterministic assumptions learned by current approaches. We confirm experimentally the benefits of our new approximation and refinement techniques.

Acknowledgements

I am very indebted to all the people who have helped me along the way to completing my Ph.D. To my supervisor, Marsha Chechik, for introducing me to model checking research, and for very keen and supportive supervision. To Arie Gurfinkel, for helping me with the intricacies of model checking tools and of multi-valued model checking. The work in Chapter 4 has resulted from my collaboration with Marsha and Arie. To Corina Pasareanu and Dimitra Giannakopoulou, for their priceless support with the work that is in Chapter 5, and for making my stay at NASA Ames such an enjoyable experience. To the members of my Ph.D. committee: Steve Easterbrook, Azadeh Farzan, and Eric Hehner, for useful feedback on my work. A special 'Thanks!' goes to the external reviewer, Kedar Namjoshi, who gracefully agreed to the task on a rather short notice, and gave me useful comments and suggestions. To Shiva Nejati, for taking me up on an interesting project of hers at some point, and for sharing with me into the many joys and sorrows of Ph.D. work. To Jocelyn Simmonds, who served as a sounding board when I was facing programming problems, and especially for being there, as a good roommate and friend, for the last part of my Toronto stay.

Personally, I cannot thank my husband enough, for sacrificing, yet again, a few good years of marriage, to give me the freedom to pursue my dreams. The RO-lunch mates: Relu, Daniela, Raluca, Cristiana, Daniel, Matei, Adrian, were also a source of encouragement and fun many times when I needed that. My sister and her family saved the day when they moved to Toronto half-way through my Ph.D., and brought along a great deal of the warmth and security that I had left behind many years ago in Romania. There are many other people with whom I shared nice moments that meant a great deal to me: Wojciech, Suzette, Yiqiao, Mehrdad, Golnaz, and the entire Formal Methods and Software Engineering groups at the Univ. of Toronto. John and Maria Brzozowski, and Daniel Berry, from Waterloo, continued to give me invaluable support.

Selfishly, I have to admit that the hardest times along the path, I had to face alone, and I realize that I have only been able to overpass them by faith in myself and in God, and the support of a few simple things: a bike, a tennis racket, music, and a few bottles of wine.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer systems are pervasive in our world today. They control critical areas of our lives: they monitor nuclear reactors, fly airplanes, carry out radiation treatments, diagnose diseases, perform financial transactions, etc. The reliability of the computer systems we build needs to keep up with the fast pace at which these systems evolve. As systems become more and more complex, however, ensuring their correct behavior is more and more challenging. Systems are still deployed without a full guarantee for their reliability. This is due to that fact that the dominant methods for system debugging are testing and simulation, which are expensive and incomplete.

As an alternative, formal verification can be used to establish the behavioral correctness of hardware and software systems statically, before they execute. For infinite-state systems, however, such as programs with unbounded inputs, verification is undecidable in general. For the finite-state case, such as systems that are control-based rather than data-based, automated verification is possible by *model checking* [39, 85, 35]. This technique takes as input a behavioral model of the system to be verified, and a correctness property. The model usually consists of the states of the system, and the transitions that the system would make between those states during execution. The property is usually given as a formulas in some temporal logic [38, 74, 75]. The verification proceeds by exhaustive exploration of all the paths in the

model according to the property. This approach has been successful in practice: Intel and IBM report on verifying large hardware designs [46, 13]; the SLAM project at Microsoft verified important properties of Windows device drivers [7, 6, 8]. Model checking, however, is still subject to a number of challenges, of which we address some important ones in our thesis.

1.1 Our Thesis

The model checking process is simple in principle: given a model of a system, and a property of its behavior, the process checks whether the model satisfied the property. There are, however, a number of non-trivial questions that have to be answered in the process: How can we make sure that the model faithfully represents the physical system behavior? How can we ensure that the property expresses the desired system behavior? The model and the property are formalized mathematically to correspond to the real, in-formal system and its requirements. Abstraction is inherent in the formalization. How can we ensure that these formal, mathematical abstractions, represent the physical or the in-formal conceptual reality? To increase user confidence in the model checking process, the process needs to support model and property debugging. Two main problems in this direction are *vacuity detection* and *query solving*.

Vacuity relates to the trivial satisfaction of the properties in a model [11]. Industrial researchers noticed that temporal logic formulas sometimes pass verification for unexpected reasons, and some parts of a formula may be *vacuous*, *i.e.*, irrelevant to its satisfaction. They noted that in about 20% of the cases, vacuity indicates a problem in the model or in the property and is useful for debugging purposes. Finding the largest vacuous subformulas of a formula requires an exponential number of calls to a model checker.

Query solving is a similar problem, originating in *model exploration*: for many legacy systems that were not formally verified when designed, properties are not available. Thus, the analysts need to explore the models in order to *infer* which properties those models satisfy. Usually some templates of the properties being sought are available. For model checking, these

templates are called *temporal logic queries*, and they consist of temporal formulas with missing subformulas. The problem is to find *solutions* to the queries, that is, the missing subformulas that can fill in the templates in order to make them into full formulas satisfied by the model [25]. This problem is again double-exponential in the cost of model checking [63].

Finally, apart from the modeling problems, the user is very likely to be faced with the inherent computational complexity of the model checking process itself. Since the number of possible model states is exponential in the number of system variables, model checking is computationally intractable, which is the well-known state-space explosion problem [34]. *Assumption generation* stems from this problem. The behavior of a system consisting of several components running concurrently is the product of the component behaviors. Computing the product model leads to state-space explosion and renders model checking intractable. To avoid this problem, verification can be applied component-wise in an *assume-guarantee* style [78, 65, 83]. This involves the computation of intermediate *assumptions* under which the components are verified individually.

Our work provides a unified treatment of these problems, by:

- Defining suitable *approximations* for each problem, by defining partial solutions that are satisfactory for specific practical purposes and can be obtained cheaper than the exact solutions.

- Describing how such approximate solutions can be obtained automatically. Moreover, defining *refinement* strategies by which approximations are incrementally computed, to further decrease the complexity of the problems.

- Demonstrating experimentally that these approximation and refinement techniques perform well on interesting practical cases.

The main thesis put forth by our work is that: for important and computationally hard model checking problems, that a typical user is likely to be challenged by, it is sufficient to

compute approximate solutions that can be efficiently obtained, eventually by refinement, and make these problems, otherwise intractable, tractable on many interesting, practical cases.

We outline the related work and our contributions in the rest of this section. More details are in Chapter 3.

1.1.1 Vacuity detection

Most research efforts for vacuity detection concentrate on extensions of vacuity to various temporal logics and variations of the definition of vacuity [70, 5, 59, 58, 20]. Very few approaches address the efficiency of the vacuity detection algorithms [84, 92].

We provide a new vacuity detection algorithm with an efficient implementation on top of the widely-used model checker NuSMV [32]. Our approach follows that of [59], where all the possible answers to vacuity detection are ordered in a lattice, and detection proceeds by model checking using that lattice. We do not use, however, the lattice of [59], since it is too complex to be implemented efficiently. That lattice gives information about *mutual vacuity*: all the *sets* of subformulas that are *simultaneously* vacuous in a formula. Model checking using the mutual vacuity lattice is equivalent to a number of calls to a regular model checker that is double-exponential in the number of system variables.

We approximate the mutual vacuity lattice to a simpler lattice that gives information only about *individual* subformulas that are vacuous *independently* of each other, that is, we keep from the original lattice elements only the singletons (sets of single vacuous subformulas). This leads to our algorithm, called VAQUOT, that we describe and evaluate on a number of models and formulas. Our results show that VAQUOT is more efficient than the naive vacuity detection defined by [70].

In addition, we show that by iterative refinement we can recover some of the missing vacuity information lost due to our approximation. By running VAQUOT repeatedly on progressively smaller subformulas of a formula, we are guaranteed to find the largest vacuous subformulas, which would be indicated by the mutual vacuity detection using the lattice of [59].

Thus, by approximation and refinement we solve efficiently a problem otherwise intractable.

1.1.2 Query solving

Previous research in this area proposes restricting the logic for the queries in order to make the problem tractable [25], or solves general queries [17, 60] without improving the complexity of the problem. Other approaches extend the queries to various logics [90] and systems [96], still keeping the problem intractable.

We propose a novel approach in which we make the problem more tractable not by restricting the logic, but by restricting the solutions to queries. In general, solutions to queries correspond to *sets* of states of a model. We restrict them to represent *individual* states, and thus define state-query solving, which is exponentially easier than general query solving. We do so by following the approach of [60], where query solving is performed by model checking over the lattice of sets of solutions proposed in [17]. Similarly to our approximation for vacuity, we approximate the lattice of [17, 60] by keeping, from each set of solutions, those solutions representing single states. This again leads to an efficient algorithm, called TLQ, which we describe and evaluate on an application from genetics. This application, as well as others that we identify, specifically require solutions to queries to be individual states, which motivated us in defining our approximation.

The number of formula atoms involved in a query determines the size of the lattice we work with, and the efficiency of TLQ. Thus, to further optimize TLQ, we propose an iterative refinement scheme in which queries are first asked about fewer atoms of the formula, and then gradually about more atoms, using solutions obtained earlier, to restrict later queries.

Due to their similarity, we treat the problems of vacuity detection and query solving in a unified way. We define a general approximation framework for model checking with lattices of sets, and obtain our approximations for the two problems as two different instances of the general framework.

1.1.3 Assumption generation

The simplest assume-guarantee rule states that: if we find an assumption such that a first component satisfies a property under that assumption, and a second property satisfies the assumption, then the model composed from the two components satisfies the property as well. With such rules, we can avoid checking a composed model, but the challenge is in generating a correct assumption that is also easy to verify.

Current approaches [2, 91, 41, 79, 93] to assumption generation use the learning framework pioneered by [42] or similar approaches [56] that guess and check assumptions and modify them using information from the counterexamples obtained when verifying the components. We extend this framework by an iterative refinement technique that also infers the alphabet of the interfaces between components during learning. We show experimentally that our refinement technique improves the performance of previous learning-based algorithms by orders of magnitude.

We also propose an alternative to the learning of assumptions that works by abstraction-refinement: it computes assumptions as abstraction of the components and iteratively refines them also using counterexamples, but according to another, well-known counterexample guided abstraction refinement (CEGAR) framework [33]. Our new algorithm, called AGAR, incorporates interface refinement as well. We argue and confirm experimentally the benefits of AGAR over the learning-based approaches.

1.2 Thesis Structure

The contributions of our thesis are rather technical; for this reason, we postpone a more detailed discussion of the problems we consider, the related work, and our contributions, to Chapter 3, after we introduce some preliminary notions of model checking in Chapter 2. We present a general approximation framework for model checking with lattices, with instances for vacuity detection and query solving, with algorithms, implementation, refinement, and evaluation, in

Chapter 4. In Chapter 5, we present our new refinement techniques for assumption generation, and their evaluation. Each of the latter two chapters contains an introduction, outlining the motivation behind the work and the structure of the chapter, additional background specific to the problem in that chapter, further comparison with related work, and conclusions and future work. We conclude the thesis in Chapter 6, with a summary of the overall contributions and directions for future work.

Chapter 2

Preliminaries

In this chapter we review basic notions of model checking. Illustrations of the concepts we introduce here appear as we use these notions in later chapters.

2.1 Models

System behavior is usually modeled by some type of a graph where nodes represent system states and edges represent transitions that the system makes between those states during execution. There are two main types of models. One is state-based, where the state of a system is described by the values of system variables at some point in time, and the system proceeds by transitions between states.

Definition 1 (Kripke structure [39]) *A* Kripke structure *is a tuple* $M = \langle S, s_0, P, I, R \rangle$*, where* S *is a finite set of states,* s_0 *is the initial state,* P *is a finite set of atomic propositions,* $I : S \rightarrow 2^P$ *is a function labeling each state with the set of atomic propositions true in that state, and* $R \subseteq S \times S$ *is a left-total transition relation.*

Another type of model is action-based. Here transitions are labeled with action names. The behavior of the system is then seen as the sequences of actions that the system may perform,

and the choices of actions it has at each step. The states are just nodes carrying no information with themselves, except for having incoming and outgoing actions.

Definition 2 (Labeled transition system (LTS) [77]) *Let $\mathcal{A}$ be a set of universal actions, and let τ denote a special,* internal *action. A* labeled transition system *is a tuple $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where Q is the set of states; $\alpha M \subseteq \mathcal{A}$ is the set of observable actions called the* alphabet *of M; $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is the transition relation, and q_0 is the initial state.*

The sequence of state transitions or actions that a system makes in one execution forms a path.

Definition 3 (Paths) *A* path *π from a state s in a Kripke structure or an LTS is an infinite sequence of states $\pi_0, \pi_1, \ldots$ in which $\pi_0 = s$ and every two consecutive states are related by the transition relation: in a Kripke structure, $\forall i \geq 0, (\pi_i, \pi_{i+1}) \in R$. In an LTS, a path also contains the actions for the transitions: $\pi = \pi_0 \sigma_0 \pi_1 \sigma_1 \ldots, \forall i \geq 0, (\pi_i, \sigma_i, \pi_{i+1}) \in \Delta \cup \{\tau\}$.*

When systems are made from multiple components, we formalize what it means for those components to execute together, by their parallel composition.

Definition 4 (Synchronous composition of Kripke structures [35]) *The* synchronous *composition M'' of Kripke structures M and M' has $S'' = \{(s, s') \mid I(s) \cap P' = I'(s') \cap P\}$, $s_0'' = \{(s_0, s_0')\} \cap S''\}$, $P'' = P \cup P'$, $I''((s, s')) = I(s) \cup I(s')$, and $R''((s, s'), (t, t'))$ iff $R(s, t)$ and $R'(s', t')$.*

Asynchronous composition of Kripke structures is as synchronous composition, except that $R''((s, s'), (t, t'))$ iff $R(s, t)$ and $s' = t'$ *or* $R'(s', t')$ and $s = t$.

We say that an LTS M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if and only if $(q_0, a, q_0') \in \delta$, and $Q = Q', \alpha M = \alpha M'$, and $\delta = \delta'$.

Definition 5 (Parallel composition [77]) *Given LTSs $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$, their parallel composition $M_1 \parallel M_2$ is the LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ where*

$Q = Q^1 \times Q^2, q_0 = (q_0^1, q_0^2), \alpha M = \alpha M_1 \cup \alpha M_2$, *and* δ *is as follows (the symmetric version also applies):*

$$\frac{M_1 \xrightarrow{a} M_1', a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2} \qquad \frac{M_1 \xrightarrow{a} M_1', M_2 \xrightarrow{a} M_2', a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'}$$

To compare models and express that fact that one model can appear to behave as the other, the notion of simulation is commonly used. We give its definition for LTSs; that for Kripke structures is analogous.

Definition 6 (Simulation of LTSs [77]) *A relation $\rho \subseteq S_1 \times S_2$ between LTSs M_1 and M_2 is a* simulation *if for all $(s_1, s_2) \in \rho$, for each $(s_1, \sigma, t_1) \in \delta_1$ there exists $(s_2, \sigma, t_2) \in \delta_2$ such that $(t_1, t_2) \in \rho$. Simulation defines a preorder denoted $\preceq$: if ρ is a simulation between M_1 and M_2, then $M_1 \preceq M_2$.*

2.2 Temporal Logics

Properties of the temporal behavior of systems are commonly formalized in various temporal logics that we present next. Our presentation follows [35], where these logics are interpreted over Kripke structures; the definitions for LTSs are analogous.

2.2.1 CTL*

Let P be a set of atomic propositions. CTL* consists of two types of formulas: *state formulas* that are evaluated in the states, and *path formulas* that are evaluated along the paths, of a Kripke structure.

CTL* is the set of *state* formulas φ generated by the following grammar:

$$\varphi \triangleq p \mid \neg\varphi \mid \varphi \wedge \psi \mid E\ \xi,$$

where $p \in P$, ψ is a state formula, and ξ is a *path formula* defined by:

$$\xi \triangleq \varphi \mid \neg\xi \mid \xi \wedge \chi \mid X\ \xi \mid \xi\ U\ \chi,$$

where φ is a state formula, and χ is a path formula.

The CTL* semantics is defined with respect to the paths of a Kripke structure having the same set P of atomic propositions as the formulas of the logic. We use π^i to denote the suffix starting at π_i in a path π. For a state formula φ, $M, s \models \varphi$ denotes that φ is satisfied in state s of the Kripke structure M. For a path formula ψ, $M, \pi \models \psi$ denotes that ψ holds along path π in M. Let φ_1, φ_2 be state formulas, and ψ_1, ψ_2 be path formulas. Then the satisfaction relation $\models$ is defined inductively as:

- $M, s \models p$ iff $p \in I(s)$;

- $M, s \models \neg\varphi_1$ iff $M, s \not\models \varphi_1$;

- $M, s \models \varphi_1 \wedge \varphi_2$ iff $M, s \models \varphi_1$ and $M, s \models \varphi_2$;

- $M, s \models E\ \psi_1$ iff there exists a path π from s such that $M, \pi \models \psi_1$;

- $M, \pi \models \varphi_1$ iff the first state s along π is such that $M, s \models \varphi_1$;

- $M, \pi \models \neg\psi_1$ iff $M, \pi \not\models \psi_1$;

- $M, \pi \models \psi_1 \wedge \psi_2$ iff $M, \pi \models \psi_1$ and $M, \pi \models \psi_2$;

- $M, \pi \models X\ \psi_1$ iff there exists $k \geq 0$ such that $M, \pi^k \models \psi_1$;

- $M, \pi \models \psi\ U\ \psi_2$ iff there exists a $k \geq 0$ such that $M, \pi^k \models \psi_2$ and for all $o \geq j < k$, $M, \pi^j \models \psi_1$.

Some derived operators are commonly used: $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $A\ \varphi = \neg E\ \neg\varphi$, $F\ \psi = \mathsf{true}\ U\ \psi$, $G\ \psi = \neg F\ \neg\psi$, $\varphi\ R\ \psi = \neg(\neg\varphi\ U\ \neg\psi)$.

ACTL* is the subset of CTL* where only the universal path quantifier A is allowed and negations are allowed only on the atomic propositions. ECTL* is the subset of CTL* where only the existential path quantifier E is allowed and negations are allowed only on the atomic propositions.

CTL is the subset of CTL* where each path operator X, U, F, G, R is immediately preceded by a path quantifier E or A. ACTL is the fragment of CTL in ACTL*, and ECTL is the fragment of CTL in ECTL*.

LTL is the subset of CTL* with formulas of the type $A\ \varphi$ where A is usually implicit, φ is

a path formula, and the only state subformulas allowed are the atomic propositions.

2.2.2 Satisfaction

A Kripke structure M satisfies a temporal logic (state) formula φ, denoted $M \models \varphi$, if $M, s_0 \models \varphi$.

2.2.3 μ-calculus

For the interpretation of μ-calculus, Kripke structures are slightly changed, so that instead of a single transition relation they have a set of transitions relations T, such that for each $a \in T$, $a \subseteq S \times S$. The a's can be thought of as labels on transitions. Let VAR $= \{Y, Z, \ldots\}$ be a set of variables, such that each variable can be assigned a subset of S.

μ-calculus is the set of formulas φ defined by the grammar:

$$\varphi \triangleq p \mid \neg\varphi \mid \varphi \wedge \psi \mid \langle a \rangle \varphi \mid \mu Y.\, \varphi,$$

where $p \in P$, ψ is a formula, $a \in T$, and Y is a variable appearing under an even number of negations in φ in the scope of μ. Some derived formulas are defined via DeMorgan's laws: $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $[a]\varphi = \neg\langle a \rangle \neg\varphi$, $\nu Y.\, \varphi = \neg(\mu Y.\, \neg\varphi)$.

The semantics is defined with respect to an environment e that assigns subsets of states to the variables, *i.e.*, e : VAR $\to 2^S$. A formula φ is interpreted as the set of states where φ holds, denoted $[\![\varphi]\!]_M e$, and defined recursively as follows:

$$\begin{aligned}
[\![p]\!]_M e &\triangleq \{s \in S \mid p \in I(s)\} \\
[\![Z]\!]_M e &\triangleq e(Z) \\
[\![\neg\varphi]\!]_M e &\triangleq S \setminus [\![\varphi]\!]_M e \\
[\![\varphi \wedge \psi]\!]_M e &\triangleq [\![\varphi]\!]_M e \cup [\![\psi]\!]_M e \\
[\![\langle a \rangle \varphi]\!]_M e &\triangleq \{s \mid \exists t.\, (s,t) \in a \wedge t \in [\![\varphi]\!]_M e\} \\
[\![\mu Y.\, \varphi]\!]_M e &\triangleq \textstyle\bigcup_i \tau_i(\mathsf{false}),
\end{aligned}$$

where $\tau : 2^S \rightarrow 2^S$ is a predicate transformer defined by $\tau(W) = [\![\varphi]\!]_M e[X \leftarrow W]$ and $e[Y \leftarrow W]$ is the environment that is like e except Y is assigned W.

With this semantics, for a Kripke structure M and a μ-calculus formula φ, we have $M, s \models \varphi$ iff $s \in [\![\varphi]\!]_M$.

2.2.4 CTL fixpoint semantics

Of particular interest for the model checking algorithms is the μ-calculus (fixpoint) semantics for CTL. The value of a CTL formula φ at state s is denoted $[\![\varphi]\!](s)$ and defined recursively as follows:

$$\begin{aligned}
[\![\ell]\!](s) &\triangleq \ell, \text{for } \ell \in \{\mathsf{true}, \mathsf{false}\} \\
[\![p]\!](s) &\triangleq p \in I(s), \text{for } p \in P \\
[\![\neg\varphi]\!](s) &\triangleq \neg[\![\varphi]\!](s) \\
[\![\varphi \vee \psi]\!](s) &\triangleq [\![\varphi]\!](s) \vee [\![\psi]\!](s) \\
[\![EX\ \varphi]\!](s) &\triangleq \bigvee_{s' \in R(s)} [\![\varphi]\!](s') \\
[\![EG\ \varphi]\!](s) &\triangleq [\![\nu Z.\varphi\ \wedge\ EXZ]\!](s) \\
[\![E[\varphi\ U\ \psi]]\!](s) &\triangleq [\![\mu Z.\psi \vee (\varphi \wedge EXZ)]\!](s)
\end{aligned}$$

Other common CTL operators are derived from these:

$$\begin{aligned}
[\![\varphi \wedge \psi]\!](s) &\triangleq \neg([\![\neg\varphi]\!](s) \vee [\![\neg\psi]\!](s)) \\
[\![AX\ \varphi]\!](s) &\triangleq \neg([\![EX\neg\varphi]\!](s)) \\
[\![AF\ \varphi]\!](s) &\triangleq \neg([\![EG\neg\varphi]\!](s)) \\
[\![EF\varphi]\!](s) &\triangleq [\![E[\mathsf{true}\ U\ \varphi]]\!](s) \\
[\![AG\varphi]\!](s) &\triangleq \neg([\![EF\neg\varphi]\!](s))
\end{aligned}$$

With this semantics, for a model M and a CTL formula φ, we have $M, s \models \varphi$ iff $[\![\varphi]\!](s) = \mathsf{true}$.

2.2.5 Safety vs. liveness

A common distinction among properties is whether they specify *safety* or *liveness* of a system. Intuitively, safety properties specify that "something bad does not happen", while liveness properties specify that "something good eventually happens". Technically, safety properties have counterexample paths of finite length, reaching a state where the "bad" condition holds. Liveness properties have infinite counterexample paths, reaching loops where the "good" thing never happens.

2.3 Model Checking Algorithms

The μ-calculus semantics given above leads to straightforward algorithms for the evaluation of CTL formulas by manipulating sets of states. A propositional subformula represents the set of states where that subformula holds. The basic step in the computation of temporal operators is the computation of EX which amounts to computing the predecessors of a set of states, also called *pre-image computation*. All other temporal operators are computed by iterative pre-image computations. For a set of states W, the pre-image operator is defined as:

$$\mathit{Pre}(W) = \{s \in S \mid \exists t \in S.\, (s,t) \in R \wedge t \in W\}$$

A class of algorithms use representations of Kripke structures or LTSs as explicit graphs and for this reason are called *explicit-state*. A possible representation of sets of states for an explicit Kripke structure is by labeling the states with the subformulas they satisfy. Computing *Pre* of a set of states in this case consists of labeling the predecessors of those states in the graph. Another approach, called *symbolic*, represents sets of states with their characteristic Boolean formulas, and set operations as Boolean operations. The transition relation R is encoded symbolically by labeling atomic propositions in a current state as unprimed $x, y, \ldots$ and labeling them in the next state with primed versions $x', y', \ldots$. Then the symbolic computation of *Pre* for a set of states characterized by Boolean formula ϕ (over $x, y, \ldots$) is the computation

of the following Boolean formula: $\exists x', y', \ldots . R(x, y, \ldots, x', y', \ldots) \wedge \phi'$, where ϕ' is ϕ with all occurrences of variables $x, y, \ldots$ replaced by their primed versions $x', y', \ldots$. Existential quantification is computed as a disjunction over all possible values, which are finitely many, as atomic propositions are Boolean.

Symbolic model checkers such as NuSMV [32] use *decision diagrams* to represent and manipulate the Boolean formulas arising in symbolic computations. Reduced Ordered Binary Decision Diagrams(ROBDDs) [18] are binary decision trees that encode the truth tables of Boolean formulas. At each level, the decision is made on the binary value true or false of a different variable in the formula; one subtree corresponds to value true and the other subtree to value false. The leaves contain Boolean constants true, false. A BDD imposes an order on the variables in which they are considered for the decisions. The value of the formula for one truth assignment to all its variables is obtained by reading the value in the leaf reached by the path following the values in that assignment, in the variable order fixed by the decision tree. A BDD is also reduced in the sense that, if different decisions lead to the same subtree, the variable on which that decision is made is eliminated. Boolean operations are performed by graph transformations and merging of these BDDs. The size of the diagrams and consequently the running time of the BDD operations depends on the number and the ordering of the variables. The state-space explosion problem manifests itself in BDD explosion in symbolic model checkers, although non-monotonically: large state spaces can be represented with small BDDs and viceversa.

Either implementations, explicit or symbolic, have a complexity of $O(|S| \times |\varphi|)$ for CTL model checking and $O((|M| \times |\varphi|)^k)$ for μ-calculus, where $|M| = |S| + |R|$, $|\varphi|$ is the number of subformulas of φ, and k is the maximum nesting depth of the fixpoint formulas. Two fixpoint formulas are considered nested if a fixpoint variable appears free in the scope of the other fixpoint [19].

A common algorithm for the evaluation of LTL formulas is automata-based and is commonly used in explicit-state algorithms for LTSs, but some symbolic implementations use this

idea as well. A Buchi automaton is like a finite automaton, except that it accepts infinite strings: a string is accepted if upon reading that string the automaton passes infinitely often through one if its "final" states (more appropriately called "accepting states"). The negation of any LTL formula can be translated to a Buchi automaton. A Kripke structure trivially translates to a Buchi automaton whose every state is accepting. Then model checking consists of checking whether the automata product (intersection) of the structure automaton and the automaton for the negation of the formula has any reachable loop containing accepting states. This procedure is linear in the size of the structure, but exponential in the size of the formula, and LTL model checking is PSPACE-complete [95].

For mostly theoretical purposes, automata-based algorithms have been defined for the model checking of CTL, CTL*, and μ-calculus. A CTL, CTL*, or μ-calculus formula can be translated to a nondeterministic tree automaton or to an alternating tree automaton [71]. These are automata that run on the computation tree of the Kripke structure. The computation tree of a Kripke structure is the infinite tree of all computation paths of the structure, obtained by "unfolding" the structure. The automata-based algorithms for CTL, CTL* and μ-calculus work by the same principle as that for LTL: they check the product of a structure automaton with a formula automaton. For CTL and μ-calculus, the automata-based algorithms have the same complexity as the corresponding explicit-state or symbolic algorithms. For CTL*, the automata-based algorithm has the same complexity as for LTL: linear in the size of the structure and exponential in the size of the formula, and CTL* model checking is also PSPACE-complete [71].

Chapter 3

The Problems: Definitions, Related Work, Our Contributions

In this chapter we give a detailed account of the problems we address in the thesis, related work, and the contributions and limitations of our work.

3.1 Vacuity Detection

3.1.1 The problem

A common problem noticed early on in the application of formal verification to hardware is that properties sometimes hold for the wrong reasons. For instance, properties about how a hardware system reacts to stimuli from its environment are often formalized as implications $a \rightarrow b$, where a represents the environment stimulus, and b, the system behavior in response to the stimulus. It may happen that the implication holds in a model simply because the environment has been modeled incorrectly and a never happens. This problem is referred to as *antecedent failure* in [10]. More evidence of the occurrence of this kind of problem in the verification practice has led IBM researchers to formulate trivial satisfaction of properties more generally as *vacuity* [11]. The formulas considered in [11] are no longer limited to implications, and

the problem manifests itself in some subformulas not being important for the satisfaction of a formula in a given model. Those subformulas are called *vacuous*, and the formula is deemed *vacuous* in those subformulas. In the case of antecedent failure, the consequent of the implication $a \rightarrow b$ is vacuous, since the implication holds regardless of b, and only because a is always false in the given model. The formal definition of vacuity given initially in [11] is as follows:

Definition 7 (Vacuity [11]) *Let φ be a temporal formula and M be a model such that $M \models \varphi$. We say that φ is* vacuous *in M if there exists a subformula ψ of φ that* does not affect *the satisfaction of φ in M: ψ can be replaced by any formula ξ, and the new φ, written $\varphi[\psi \leftarrow \xi]$, still holds in M.*

Definition 7 is not effective as it does not lead directly to an algorithm for vacuity detection: there are infinitely many formulas that can be replaced for ψ! This definition, however, gives a framework for several approaches in the research work related to vacuity detection that we summarize next. These approaches differ from each other with respect to the temporal logics whose formulas are checked for vacuity, the semantics of vacuity, in particular of "does not affect", and the corresponding detection algorithms.

3.1.2 Related work

Based on industrial experience, Beer *et al.* [11] define a subset of ACTL, called w-ACTL, for which they are able to detect vacuity efficiently. w-ACTL is so that for any formula φ a single "witness" $w(\varphi)$ is used to detect the vacuity of φ in some of its subformulas, deemed "important". The definition of w-ACTL intuitively captures the interaction between a system and its environment beyond simple implications. Each binary operator connects a propositional formula to a temporal one. The assumption, stemming from observations of the typical uses of CTL, is that the propositional formula represents the stimuli from the environment and the temporal one specifies the response of the system.

The semantics of vacuity for w-ACTL is that a formula φ is vacuous in a model M if

the "witness" formula $w(\varphi)$ holds in M [11, 12]. The witness is obtained by replacing the smallest state subformula with false. The replacement proceeds top-down on the parse tree of the formula, at each binary operator being applied recursively on the non-propositional operand. This is a generalization of antecedent failure where the consequent is vacuous and it represents the response of the system to the environment stimuli; antecedent failure can be detected by replacing b with false in $a \rightarrow b$ and verifying that the implication still holds. For w-ACTL, the temporal subformulas represent the system response and are therefore considered "important", *i.e.*, likely to be vacuous, hence replacement is applied to these subformulas. Model checking of $w(\varphi)$ is linear in the sizes of the formula and of the model. Thus, this algorithm is of the same complexity as CTL model checking.

Kupferman and Vardi[69, 70] define vacuity for CTL*, restricted to (anti)monotonic formulas. (Anti)Monotonicity is ensured by a sufficient syntactic condition: a formula is (anti)monotonic all the occurrences of its subformulas are in the scope of an even number of negations, or all of them are in the scope of an odd number of negations. Such formulas are called of *pure polarity*, otherwise they are of *mixed polarity*. In this context, a formula φ is vacuous in a model M if it contains a state subformula ψ such that $\varphi[\psi \leftarrow \mathsf{true}]$ and $\varphi[\psi \leftarrow \mathsf{false}]$ both hold in M [69, 70]. This definition can be applied to detect vacuity of false formulas (unsatisfied in the model) as well, as done in [59] for CTL: a false formula φ is vacuous in M if for some subformula ψ, both $\varphi[\psi \leftarrow \mathsf{true}]$ and $\varphi[\psi \leftarrow \mathsf{false}]$ are false in M. Samer [90] modifies the definition slightly, by parameterizing it with a finite set of replacements that the user can specify as the likely causes of vacuity. A formula is vacuous if none of those replacements affect the value of the formula in the model.

Vacuity detection by the definition of Kupferman and Vardi is performed by creating two "witnesses" for each state subformula ψ: $\varphi[\psi \leftarrow \mathsf{true}]$ and $\varphi[\psi \leftarrow \mathsf{false}]$. The total number of witnesses is thus linear in the size of the formula. The witnesses are then model checked. The method applies to any CTL* formulas as long as different occurrences of the same subformula are treated as different subformulas. The overall complexity of this algorithm for CTL is linear

in the size of the model and quadratic in the size of the formula, and for LTL and CTL* it is $|\varphi|$ times the cost of model checking. Purandare and Somenzi [84] present an optimized version of this algorithm for CTL where all witnesses are checked in a single parsing of the formula and intermediate results are cached for reuse; while this does not change the complexity of the algorithm, it shows up to 40% time improvement in a number of test cases reported in [84]. Gurfinkel and Chechik [59] define a framework that reduces vacuity detection, as defined by Kupferman and Vardi, to multi-valued model checking over a lattice of possible vacuity answers. They also introduce the notion of *mutual vacuity* for subformulas that are syntactically non-overlapping and simultaneously vacuous in a formula. Since this notion is important for our work, we give its definition next. It is not the definition introduced in [59], but an equivalent one.

Definition 8 (Mutual vacuity) *Let φ be a temporal formula and M be a model such that $M \models \varphi$. Let ψ, ξ be two non-overlapping subformulas of φ. We say that ψ, γ are* mutually vacuous *in φ (in M) if φ is vacuous in ψ, and for any replacement ξ of ψ in φ, $\varphi[\psi \leftarrow \xi]$ is vacuous in γ, where vacuity is as defined in Definition 7.*

Beer *et al.* [12] extend the vacuity detection of [69, 70] to any logics whose operators are monotonic or antimonotonic in each operand; these are called *logics with polarity*.

Armoni *et al.* [5] remove the restriction to pure polarity and hence to (anti)monotonic formulas by introducing vacuity detection for LTL formulas with mixed polarity. Gurfinkel and Chechik [58] extend this approach to CTL*, whereas Bustan *et al.* [20] extend it to LTL formulas containing certain forms of regular expressions coming from practical experience. Simmonds *et al.* [92] also address the vacuity of LTL formulas. In these approaches, syntactic replacements of subformulas as described so far are no longer correct to detect vacuity. A formula with mixed polarity is no longer monotonic or antimonotonic, hence replacement of only the *extreme* semantic values true or false is no longer sufficient. A subformula corresponds semantically to a set or a sequence of states on a computation path, depending on whether the

subformula is a state or a path formula, respectively. Thus, the replacements need to take into account all possible sets or sequences of states along computation paths.

In [5, 58], a formula φ is vacuous in a subformula ψ if the quantified formula $\forall x.\, \varphi[\psi \leftarrow x]$ holds. For LTL extended with certain regular expressions, [20] uses the same definition, where quantification also applies to subformulas that are regular expressions. A similar definition is used in [92], where a variable p is vacuous if $\varphi[p \leftarrow x]$ holds, with x a fresh variable. The complexity of vacuity detection is linear in the complexity of model checking for LTL and ACTL* and ECTL* [5]. For general CTL* formulas it is 2EXPTIME-complete (EXPTIME-complete for CTL) [58]. For LTL with regular expressions it is in EXPSPACE and is NEXPTIME-hard [20].

For μ-calculus, Namjoshi [80] defines a proof-based semantics for vacuity: a proof is obtained as a by-product of model checking the formula; then, the formula is considered vacuous if some of its subformulas are not essential to any possible proof. The detection algorithm works by building a maximal proof and has the same complexity as model checking. Dong *et al.* [43] define a different semantics for vacuity of μ-calculus: a formula is vacuous if there exists a subformula of it that can be strengthened; a subformula in μ-calculus represents a set of states, and it is strengthened when mapped to a smaller set of states. This latter definition can also be extended to false formulas where subformulas are weakened. The strengthening or weakening results from replacements of subformulas with false or true, respectively, as in [69, 12], and with the same complexity, or by simplifying the transition label sets of the modal operators "$\langle\ \ \rangle$" or "$[\ \ ]$". Thus, the notion of vacuity is extended to capture unnecessary labels of modal operators. The detection algorithm works by considering maximal sets of labels to be replaced, and is $n \times |\varphi|$ more expensive than model checking.

3.1.3 Our contribution

We improve the vacuity detection of Kupferman and Vardi by proposing a new algorithm along the lines of [59]. We choose the definition of vacuity given by Kupferman and Vardi because

it does not restrict the temporal logic, as was done by Beer *et al.* [11], and allows us to work with full CTL. At the same time, it is less computationally expensive than semantic notions of vacuity as defined by Armoni *et al.* [5] and other similar approaches. It also does not rely on any special underlying engine providing proofs (as in [80]), which most model checkers do not provide.

We follow the approach of [59] where all possible mutual vacuity answers are ordered in a lattice, and the detection proceeds by model checking with this lattice. We, however, do not use the lattice of [59]. We define a new *vacuity lattice* that restricts or *approximates* the mutual vacuity answers to subformulas that are syntactically non-overlapping, and independently (not mutually) vacuous in a formula for a given model, by the definition of Kupferman and Vardi. The difference between mutual and individual vacuity is that mutually vacuous formulas can be *simultaneously* replaced by constants, whereas individually vacuous formulas are *independently* replaced by constants, without affecting the value of a formula.

Thus, our approximation loses the information about *mutuality* of the vacuity, but in exchange for a gain in complexity. Formally, using the mutual vacuity lattice of [59], an answer to vacuity detection is a set of *sets* of mutually vacuous formulas, whereas ours is a set of singleton vacuous subformulas. Detecting mutual vacuity by model checking using the mutual vacuity lattice of [59] is equivalent to making simultaneous replacements by constants for all possible subsets of atomic propositions, and checking the resulting "witnesses", similar to the approach of Kupferman and Vardi. This requires a number of runs of a classical model checker that is *exponential* in the number of atomic propositions. Our algorithm, called VAQUOT, is equivalent to Kupferman and Vardi's algorithm, and thus equivalent to a *linear* number of calls to a classical model checker. Intuitively, VAQUOT essentially checks all the witnesses of Kupferman and Vardi in parallel, and reports whether a formula is vacuous, and which of its subformulas are vacuous in the model. Experimentally, we show running time improvements up to 30% in several test cases over the naive detection following the definition of Kupferman and Vardi. The results have been published in [51]. Here we present it slightly differently, by

showing that it is an instance of a more general approximation framework, that we also define and use also for query solving.

Our approximation is an over-approximation, in the sense that a set of subformulas found independently vacuous may not be mutually vacuous, but we know that any mutually vacuous set has to be a subset of that found. Thus, we can reach a full solution, that is, a mutually vacuous set, from the approximation found, by additional checks. We describe an iterative refinement technique that traverses the parse tree of the formula top-bottom in a breadth-first manner, and applies VAQUOT to the subformulas on each level; thus, we consider incrementally more and smaller non-overlapping subformulas. In other words, this iterative refinement calls VAQUOT repeatedly with an incrementally growing lattice of increasingly refined vacuity answers. This algorithm has the advantage of being able to stop early, as soon as we find a larger subformula being vacuous; in that case, we need not consider its subformulas, since then we know that they will all be vacuous (we work with monotonic formulas). The scheme pays off if it seldom reaches the lower levels of the parse tree that have many non-overlapping formulas (in the worst case, the leaves of the parse tree, with the most formulas, that is, the atomic propositions). The number of formulas that VAQUOT runs on, at any time, determines the number of elements of the lattice and thus affects the performance of our algorithm. This scheme also recovers some of the information lost due to our approximation, since it discovers the largest vacuous subformulas that would also be found by mutual vacuity detection.

3.1.4 Limitations

Our approach is an effort to improve the performance of vacuity detection according to a logic-based definition of vacuity. Other optimizations in this direction are possible, but we think it is more important to address the main shortcoming of approaches such as ours, namely that the vacuity answers may be hard to interpret, which is what we observed while doing our experiments with VAQUOT. One problem is that the partitioning of a model into the system versus its environment is lost during model checking. The causes of vacuity initially observed

in practice came from errors in the environment modeling (see the antecedent failures), but subsequent vacuity detection approaches have lost this track. Only Beer *et al.* [11] take into consideration the reactivity of the system to the environment implicitly, through the definition of the logic w-ACTL.

Another problem is that the analyst expects feedback for his/her description of the model and property, which are often different than the compiled model and temporal formula. When vacuity of a formula is established in some of its subformulas, the user needs to undertake a non-trivial effort to identify the parts of the model description responsible for that vacuity, as we ourselves often experienced. A related problem is that vacuity is property-centric and thus likely to give many false alarms for model debugging. Our experience with several test cases suggests that most of the vacuity reported by current algorithms does not indicate errors, but only that properties can be simplified.

These problems are addressed in other recent work [14, 31], and some of our own [27], which explore alternative notions of vacuity.

3.2 Query Solving

3.2.1 The problem

A related property-based framework that helps in model debugging is that of query solving. In this framework, users can ask some partial questions, called temporal logic *queries*, to explore which properties hold of their models.

For instance, instead of asking *whether* a switch becomes "on" eventually along all paths, in CTL: AF *switch*, a query asks *what* value the switch eventually takes along all paths, as a CTL query: $AF\ ?\{$*switch*$\}$. This is useful when trying to understand large models that are not accompanied by clear specifications of their behavior, which is common in system maintenance.

Definition 1 (Query and solution [25]) *A CTL query is a CTL formula with a missing propositional subformula, designated by a placeholder ("?"). A query is* positive *if the placeholder appears in the scope of an even number of negations, and* negative *otherwise. A* solution *to the query is any propositional formula that, when substituted for the placeholder, results in a CTL formula that holds in the model.*

In our example, both "on" and "off" are solutions, if both AF *switch* and $AF\ \neg$*switch* hold in the model.

Since there are 2^{2^n} distinct Boolean formulas over any given n variables, a naive algorithm that tries each possible formula as a replacement for "?" in the query and then model checks the resulting temporal formula makes a double-exponential number of calls to a model checker. This gives an upper bound on the complexity of the problem.

3.2.2 Related work

The precursor of query solving is the problem of inferring invariants of systems, that is, properties that hold in every state [64]; in query terms, that means solving $AG\ ?$. Based on experience with analyzing large software, Chan [25] identifies the need for a generalization to the inference of invariants and, as a result, formalizes for the first time the problem of solving temporal-logic queries, using CTL. Realizing that the problem is in the worst case double exponential in the number of state variables, Chan defines a subset of CTL queries such that the set of their solutions has a minimal element: a strongest subformula, that implies all other subformulas in the set. Chan calls these queries *valid*. The naive method Chan envisions for solving such queries is: try out all minterms (*i.e.*, conjunctions of all propositions or their negations) and report the disjunction of all that are solutions, if the query is positive (*i.e.*, monotonic in the empty position); if the query is negative (*i.e.*, anti-monotonic), try out all negations of minterms and report the conjunction of all such negations that are solutions. This amounts to solving a number of model-checking problems that is exponential in the number of atomic propositions. Since this complexity is still not practical, Chan finds a syntactic subclass of valid queries for which he

devises an algorithm with the same complexity as model checking. The algorithm works as symbolic CTL model checking, except that it computes fixpoint formulas forward, from the set of initial states, rather than backward, from the states satisfying the innermost propositional formulas, for those missing propositional formulas in the queries. He also defines a grammar for generating these restricted valid queries. The query for inferring invariants *is* among these valid queries.

Chan's work is generalized and extended by Bruns and Godefroid [17]. They introduce lattices to represent the structure of the solution-space of temporal-logic queries and extend classical tree-automata-based model checking to model checking over these lattices to solve queries. They use the fact that propositional formulas are ordered by implication. For any kind of query (*i.e.*, not only valid ones), the query solving problem (that they call *query checking*) is formalized as the problem of finding the set of strongest solutions to the query, with respect to implication. Bruns and Godefroid define the lattice of sets of strongest solutions. The elements of the lattice represent all possible sets of query solutions. For positive queries, these sets can be uniquely represented by their minimal elements, that are the strongest formulas that are solutions to the query. The meet and join operations of the lattice correspond to set intersection and union, respectively, computed in terms of the minimal elements. The authors devise a model checking algorithm that constructs an alternating tree-automaton for the query and for the model, computes their product, and checks it for emptiness, just like in classical automata-based model-checking, except that the translation for the query introduces values from the lattice, and conjunction and disjunction are replaced by the lattice meet and join. The algorithm is so that its output is a value of the lattice that gives the set of strongest solutions to the query. This method also shows how solutions to queries can be computed compositionally, from solutions to subqueries. The complexity of this approach, however, is the same as that of trying all possible solutions naively, so still double-exponential in the number of state variables.

Gurfinkel *et al.* [57, 28] study the problem more thoroughly, generalizing queries to have more than one placeholder, with mixed (positive and negative) occurrences. They also discuss

several new applications of query solving in software engineering. The authors also use the lattice introduced by Bruns and Godefroid, but give a symbolic model checking algorithm over this lattice, for queries with multiple placeholders. The algorithm is based on a translation of CTL queries to formulas containing lattice elements, and on a multi-valued semantics which replaces classical conjunction and disjunction with lattice meet and join. Given a query, the algorithm results in the lattice value representing the set query solutions. The authors implement the algorithm with a multi-valued model checking tool and report its evaluation on an example of a cruise control system that shows better performance than the double-exponential upper bound.

In a parallel development, Hornus and Schnoebelen [63] study the complexity of computing strongest solutions to CTL* queries, and deciding uniqueness of these solutions, where queries are general (not necessarily valid). They show that the problem of checking whether a given query has a unique strongest solution in a *given* system, and computing this solution, can be solved with a linear number of model-checking runs. The result is generalized to checking completeness of a set of strongest solutions. They also describe a method by which computing a first strongest solution requires a linear number of model-checking runs, a second one - a quadratic number, and so on. They show that these increasing costs are unavoidable, since even counting the strongest solutions is intractable. They establish that the double-exponential upper bound on the complexity of general query solving is also a lower bound.

In a series of papers [88, 90, 89], Samer and Veith continue Chan's work of finding subclasses of valid queries. In [88], the authors point out and correct an error in Chan's initial grammar for valid CTL queries, and discuss the challenge of deciding syntactically whether a query is valid. In [89], they give an analogous grammar for valid LTL queries. In [90], they formally establish a connection between vacuity detection and query solving, and briefly discuss how it leads to extensions of query solving frameworks that allow temporal rather than only propositional solutions. The connection to vacuity simply states that a formula is vacuous in a subformula, iff the query obtained by replacing that subformula with a placeholder, has

false as its strongest solution, for positive queries (true for negative queries). Since then all possible formulas are solutions to the query, they do not affect the value of the formula, hence the vacuity. Our work explores a further connection between the two problems: we do not consider vacuity detection as a special case of query solving (thus reducing the easier problem of vacuity detection to the harder problem of query solving), as pointed to by [90], but solve vacuity detection independently, and keeping its relative easiness, even if using very similar techniques as for query solving.

Zhang and Cleaveland [96] define and solve queries in μ-calculus for Presburger system, that are systems whose behavior is described by Presburger formulas, with integer-valued variables and linear inequalities on those variables. Their method uses tableaux for evaluating μ-calculus queries; the tableaux are tree-structured proofs whose leaves contain assignments of integer values to variables. The placeholder in the query makes some leaves have no assignments for some variables. The solutions to queries are those assignments at the tableau leaves that make the tableau/proof successful, *i.e.*, make the query into a true formula. The authors distinguish between existential and universal query checking in their framework. 'Existential' means finding query solutions for one tableau, and has the same complexity of μ-calculus model checking. 'Universal' query checking means enumerating all possible tableaux, and is exponential in the cost of μ-calculus model checking. They also report on an implementation and a case study where they evaluate their algorithm by comparison to naively model checking the formulas obtained by replacing the placeholder with all possible variable assignments. The evaluation, however, does not show significant improvement over the naive approach.

3.2.3 Our contribution

We note that for many applications, only state solutions to queries are needed, that is, only those propositional subformulas that are conjunctions of all atomic propositions or their negations. The number of such formulas is single-exponential in the number of atomic propositions. Finding only the state solutions to a query is thus an easier problem that can be solved naively

with a single-exponential number of calls to a model checker: replacing the placeholder with every possible propositional formula representing a single state, and checking the resulting formulas sequentially.

We give an algorithm, called TLQ, that finds, for any CTL query, exactly the solutions that represent single states. Our implementation is fully symbolic, consisting of a single model-checking run over a lattice of sets of state-solutions. Some potential applications of this algorithm are: finding reachable states, finding procedure summaries, or dominators/postdominators in program analysis.

Our approach is similar to that of [60]: it uses a lattice of sets of solutions, translates a query into a formula containing values from that lattice, and model-checks the resulting formula over the lattice. We show how our lattice and algorithm compute an approximation of the results of [60]. The approximation is an instance of a general approximation framework that we introduce for model checking with lattices of sets, and apply also to vacuity detection. Essentially, the lattice used in [60] contains all sets of propositional formulas that may solve a query. A propositional formula in general represents a set of states. If a formula is a solution to the query, it is not true that any single state in the set represented by that formula, also represents a solution to the query. We obtain our lattice by keeping from the lattice of [60] only those formulas that represent single states. The main novelty of our approach to query solving is that it makes the problem more tractable by restricting the shape of the solutions, as opposed to restricting the logic, as was previously done.

We also show a new application coming from genetics which motivated our approximation. The application asks to find the stable states of a gene network, which amounts to finding the state solutions to CTL query $EF\ AG\ ?$. We show that our implementation solves this problem more efficiently than an algorithm making naive replacements and checking the resulting formulas sequentially.

Our algorithm has the potential problem that it doubles the number of propositional variables and it may suffer from BDD size explosion. To handle this complexity, we describe

another iterative refinement scheme under which queries are asked gradually about more and more atomic propositions. Solutions obtained with a smaller set of atoms are used to restrict the next run of our algorithm with a superset of those atoms. This work has been published previously in [52, 50].

3.2.4 Limitations

The main limitation of our approach is imposed by applications. We have shown its effectiveness on an interesting and important problem, that of finding the stable states in gene networks, but we need to perform extensive evaluation to assess its merits. We aim, however, to keep the evaluations in tune with real applications, rather than using artificial cases. We expect that new applications may require new approximations to query solving, but we hope that our general approximation framework can help. For our particular implementation, the challenge remains to avoid BDD size explosion. We need to investigate caching schemes that can help, in addition to our iterative refinement technique.

The main drawback of query solving in general is its complexity. Any further improvements have to concentrate on practical cases where query solving is useful and identify classes of problems and any characteristics that can help make the problem more tractable in those cases. Our approach makes one step in this direction, although it only exploits one parameter of the problem: the shape of solutions. We believe that domain-dependent problem parameters should be identified and exploited in order to effectively fight against the intractability of query solving.

3.3 Assumption Generation

3.3.1 The problem

As we have already mentioned, model checking is a hard problem itself as it suffers from the well-known state-space explosion. A feasible approach to solving this problem is compositional, where properties are verified of a multi- component model without actually composing the entire model that may lead to state-space explosion. A discipline for compositional verification is established by assume-guarantee rules which show how to verify each component individually using assumptions about the rest of the components.

Definition 2 (Assumption generation) *For a model consisting of components M_1 and M_2, and a property φ, find an assumption A such that M_1 under A satisfies φ, and M_2 satisfies A.*

The simplest assume-guarantee rule ensures that if we find such an A, then the system of M_1 *and* M_2 satisfies φ.

3.3.2 Related work

Misra and Chandy [78] provide one of the earliest methodologies for compositional verification of invariant properties of networks of processes communicating through messages. They introduce a triple notation similar to the Hoare triples used in verification of sequential processes. A triple is of the form $r \mid h \mid s$ where h denotes a process and r and s are assertions. The meaning of the triple is: s holds initially in h, and if r holds at all times prior to any message transmission of h, then s holds at all times prior to and immediately following that transmission; a message transmission by h is either h sending or receiving a message. For a network $H = h_1 \parallel h_2 \parallel \ldots$ of processes h_i, the proof rule given in [78] for combining verification results is: if $r_i \mid h_i \mid s_i$ for all i, then $\bigwedge_i r_i \mid H \mid \bigwedge_i s_i$.

Jones [65] discusses a similar approach for processes communicating through shared variables. He introduces the terminology of "rely-condition" and "guarantee-condition" for the

first and last members of a triple, respectively. His proof rule stipulates that the rely conditions of each component should only depend on the rely-conditions overall, and two (or more) components must be able to coexist, in the sense that the guarantee-conditions of one should be at least as strong as the rely-conditions of the other.

Pnueli [83] formalizes compositional verification for the temporal logic LTL. He introduces the *assume-guarantee* paradigm similarly to [78, 65]. A triple is denoted $\langle\varphi\rangle M_1\langle\psi\rangle$ and means that component M_1, assuming its environment behaves as specified by LTL formula φ, ensures LTL formula ψ holds. The following assume-guarantee proof rule is given:

$$\frac{\begin{array}{c}\langle\varphi\rangle M_1\langle\chi\rangle \\ \langle\psi\rangle M_2\langle\xi\rangle \\ \theta\wedge\xi\rightarrow\varphi \\ \theta\wedge\chi\rightarrow\psi\end{array}}{\langle\theta\rangle M_1 \parallel M_2\langle\chi\wedge\xi\rangle}$$

Usually the rule is used in the following simplified form which needs only one assumption A, and we use in our work as well:

Rule ASYM

$$\frac{\begin{array}{ll}1: & \langle A\rangle\ M_1\ \langle\varphi\rangle \\ 2: & \langle\mathsf{true}\rangle\ M_2\ \langle A\rangle\end{array}}{\langle\mathsf{true}\rangle\ M_1 \parallel M_2\ \langle\varphi\rangle}$$

where φ is the property we wish to establish of the closed system. Note that the rule is asymmetric in the use of the two components, hence its name.

This rule provides a general framework for the development of compositional techniques in model checking. The different assume-guarantee model checking approaches are distinguished by: the temporal logics used for formalizing φ, the type of structures M_1 and M_2 and of their composition, and, the application of the assume-guarantee rule, which is determined by finding A.

Clarke *et al.* [37] define a particular application of the assume-guarantee rule and instantiate it for CTL formulas of asynchronous processes, and also for CTL* formulas of synchronous processes. They consider, for the asynchronous case, processes modeled as LTSs whose composition is defined as usual, with synchronization on common actions and interleaving of the rest. For the synchronous case, they consider Moore machines composed synchronously (Moore machines are like Kripke structures, except the state variables are split into input, internal, and output variables). Clarke *et al.* provide a new proof rule consisting of two symmetric applications of the assume-guarantee rule. The assume-guarantee rule is non-symmetric in the use of the component and the environment. In the new rule of [37], a second application of the assume-guarantee rule switches the roles of the component and the environment. The rule is also simplified in the sense that it uses as assumption A for one process M_1 (component or environment) simply the other process M_2, projected on the interface with M_1. Projection consists of hiding the atomic propositions or the actions not in the interface. This way, the second premise of the assume-guarantee rule is no longer needed. The new rule is:

$$\frac{\langle M_2{\downarrow_{\alpha M_1}}\rangle M_1\langle\varphi_{\alpha M_1}\rangle \qquad \langle M_1{\downarrow_{\alpha M_2}}\rangle M_2\langle\psi_{\alpha M_2}\rangle}{\langle M_2\rangle M_1\langle\varphi_{\alpha M_1}\wedge\psi_{\alpha M_2}\rangle}$$

where $\alpha M_1, \alpha M_2$ are the alphabets of the component and environment, that is, the set of actions or atomic propositions used in their models. We use an alphabet as index on a formula to show that the formula is constructed only from atoms in that alphabet. The rule is proved sound, but its application benefits verification only if the projected processes are much smaller than the original ones. Its benefit is illustrated in two case studies: a tree arbiter used to control access to shared resource [37], and the modular controller of a CPU with decoupled access and execution units [40].

Grumberg and Long [55] define assume-guarantee model checking of ACTL* formulas of synchronous processes and consider Moore machines composed synchronously. They show that for ACTL* formulas, the premises of the assume-guarantee rule reduce to simulation (re-

finement) checking. For every ACTL* formula φ, the rule can then be applied as follows:

$$\frac{M_1 \preceq A \qquad A \parallel M_2 \preceq A' \qquad M_1 \parallel A' \models \varphi}{M_1 \parallel M_2 \models \varphi}$$

where A, A' are structures representing assumptions. This rule applies to any Kripke structures with synchronous composition. For a component and its environment that are Moore machines closing each other, Grumberg and Long further show that it is sufficient to check a component by closing it with the *maximal* environment with respect to the simulation relation, that is, the environemnt that assigns non-deterministic values to the input variables of the component Moore machine in each computation step. For any environment M_2, it is shown that $M_1 \parallel M_2 \preceq M_1 \parallel M_2^{max}$, where M_2^{max} is the maximal environment. Thus, M_2^{max} is used for A', A is no longer needed, and the rule is applied as:

$$\frac{M_1 \parallel M_2^{max} \models \varphi}{M_1 \parallel M_2 \models \varphi}$$

Grumberg and Long illustrate the effectiveness of their approach on an example of a CPU controller.

Any assume-guarantee approach relies on the definition of appropriate assumptions. The work presented so far gives sufficient conditions under which reasonable assumptions can be computed. When these conditions are not applicable or are not effective, the discovery of assumptions rests in the expertise of the verification specialists and is a manual process. Other work reports implementations and case studies that suggest methodologies for choosing the right assumptions [62, 47]. The lack of more systematic assume-guarantee techniques comes from the lack of algorithmic methods for assumption discovery. Recent work has made significant progress in automating the computation of assumptions.

Cobleigh *et al.* [42] propose a fully-automated method for the assume-guarantee model checking of safety properties. The method uses a known algorithm that is able to infer or

learn a regular languages using oracles [4, 87]. To learn an unknown language L, the oracles are used to answer two types of queries: membership queries asking whether a string s is in L, and conjectures asking whether the language inferred at some point is L. When used with the assume-guarantee rule, the oracles are instances of a model checker. The unknown language is learned as an automaton representing the assumption A and whose alphabet is the interface between the component and its environment. One model checker instance answers membership queries by checking whether a given string violates the property φ when composed with M_1: $\langle s \rangle M_1 \langle \varphi \rangle$. Given a conjectured assumption automaton A, another instance checks the first premise of the rule: $\langle A \rangle M_1 \langle \varphi \rangle$, and yet another instance checks the second premise: $\langle \mathit{true} \rangle M_2 \langle A \rangle$. Counterexamples obtained at any point from model checking are used to modify the conjectured automaton and make new conjectures that are checked by the oracles again. The algorithm is guaranteed to converge at a minimal automaton that satisfies both premises of the assume-guarantee rule and represents the unknown assumption A.

Alur *et al.* [2] provide a symbolic implementation of this method. Cobleigh *et al.* [41] and Nam and Alur [79] independently extend the method to also decompose systems automatically and optimally in terms of computational resources (time and memory) used by the learning for those decompositions. Various algorithmic optimizations to the basic assumption generation framework of [42] are proposed in [24, 93], and an alternative inference mechanism to L* is proposed in [56]. Farzan *et al.* [45] extend the learning framework to liveness properties.

Flanagan and Qadeer [48] propose a different method to automatically discover assumptions for the assume-guarantee verification of multi-threaded Java programs. A guarantee for each thread is computed as a relation on the global modifications to the store done by that thread. The computation starts with the empty relation and iteratively fills it in during model checking, based on the guarantees of other threads. The assumption for each thread is then the disjunction of the guarantees of all the other threads.

Other work studies the automatic computation of assumptions independently of the assume-guarantee rule. Giannakopoulou *et al.* [53] provide an algorithm that, given an LTS of a com-

ponent and one of a safety property, finds the *weakest* environment assumption, as an LTS as well, composed with which the component satisfies the property. Similar ideas are used for *interface synthesis* for (sequential) software components in the work of Alur *et al.* [1] and Henzinger *et al.* [61]. In that context, the interfaces are automata describing the allowed sequence of calls to methods of a component.

3.3.3 Our contribution

In this work, we use action-based LTSs, as opposed to our work in vacuity detection and query solving that uses state-based Kripke structures.

We introduce a novel iterative refinement technique that extends the learning-based framework of [42] so that the alphabet of the assumption being learned is also inferred during learning. In the original framework of [42], the alphabet of the assumption automaton being learned is fixed to consist of all the actions in the interface between components. Our intuition is that not all of these actions are needed for verifying a given property. Our refinement loop starts learning the assumption with a small alphabet containing the actions referred to explicitly in the property to be verified, and adds actions to this alphabet as needed. The need to add actions is discovered by extending the counterexample analysis during learning. Actions are added greedily, and the process converges either by concluding that the property is not satisfied, or reaching an assumption good enough to satisfy the premises of the assume-guarantee rule and conclude that the property holds.

Our refinement technique automatically introduces a notion of approximation since the intermediate assumptions being computed with a subset of the interface alphabet are approximations of the assumption that would be obtained with the full interface alphabet. We formally show that they are under-approximations: they contain fewer behaviors than the full interface assumption, and assumptions in later refinement stages contain more behaviors than those in earlier stages of the refinement. The benefit of our refinement technique is that it leads to smaller assumptions and smaller verification problems. We show experimentally that it makes

tractable many cases that were intractable with the original learning framework, and it is also more scalable than non-compositional verification. We also study its efficiency for various assume-guarantee rules, including symmetric and circular ones. The technique is not particularly tied with learning; it can be applied to other compositional verification approaches; for instance, we also use it in our second contribution, described next.

We further improve automated assumption generation by proposing an alternative to the current learning-based techniques. These techniques can only generate deterministic assumptions. It is well known that a non-deterministic automaton can be exponentially smaller than a deterministic one for the same language. We thus remove the restriction to determinism in an effort to reduce assumption and problem sizes even more than out previous work. We achieve this goal by computing assumptions as abstractions. For components M_1 and M_2, and property φ, with Rule ASYM, *Premise 2* is satisfied if A is an over-approximation of M_2, *i.e.*, if it contains the behaviors of M_2 and possibly more. We can construct such an A and refine it using counterexamples as in the well-known framework of Counterexample Guided Abstraction Refinement (CEGAR) [33]. We also incorporate our previous alphabet refinement technique in our new method. We report experimental results which show that our new algorithm, called AGAR, can be significantly better than the original learning-based techniques.

3.3.4 Limitations

The technical limitations of our work are that both the alphabet refinement and the abstraction refinement are formulated only for action-based models and for the verification of safety properties. A natural extension would be to consider state-based models and liveness properties. For alphabet refinement, we also need to explore model-dependent ways of identifying which actions to add at each refinement step. Our current methods are independent of the model; they simply compare alphabets of traces. We could use slicing techniques, and use multiple counterexamples. For our abstraction refinement technique, we need to compare how it performs as opposed to monolithic verification. With learning, we were able to show that compositional

verification scales better than non-compositional verification only after implementing our alphabet refinement optimization and implementation for n-components by recursive application of Rule ASYM. Similar implementation and evaluation of our abstraction refinement are still pending.

3.4 Summary

We have presented in more detail the problems we address in this thesis, related work, and have outlined the main contributions and limitations of our work. In the following chapters, we present the technical details of our contributions, and additional comparison with related work.

Chapter 4

Vacuity Detection and Query Solving

4.1 Introduction

Vacuity detection and query solving are similar problems that we treat here in a unified way. Both problems have been formulated as multi-valued model checking over lattices of sets, where the sets represent all possible answers to vacuity detection or all possible solutions to queries. In their most general formulation, these problems require an exponential number of runs of a classical model checker. Their multi-valued formulation allows them to be solved by a single run of a multi-valued model checker. The efficiency of the run is determined by the implementation of the lattice operations. With the general lattices of solutions being very complex, such efficient implementations are unlikely.

In this chapter, we first provide a unifying framework that allows the definition of approximations for model checking over lattices of sets in general. We then propose simpler lattices for vacuity detection and query solving that are instances of the general framework. These approximations are defined in order to solve efficiently interesting practical cases, rather than the general problems. We also describe implementations of model checking with these lattices in the classical symbolic model checker NuSMV [32], and evaluate them experimentally, showing that they outperform naive algorithms. Being symbolic, our algorithms are exposed to the

size explosion of the decision diagrams used to encode the problems, as in any symbolic approach. To handle this problem, we describe how approximations can be computed by iterative refinement, that works with incrementally larger lattices.

4.1.1 Vacuity detection

Although various approaches to vacuity detection have been proposed (see Chapter 3), few efficient algorithms and implementations have been reported. We work with CTL formulas and the definition of vacuity of Kupferman and Vardi (see Chapter 3). We choose this definition, since it does not restrict the logic as done by Beer *et al.* [11], and it is less computationally expensive than semantic definitions as those of [5, 58]: it is equivalent to a linear number of runs of a model checker, rather than an exponential number, as in the latter approaches. It also does not require special model checkers that provide proofs, as required in [80, 92].

Our algorithm is called VAQUOT and is based on techniques described in [59], where a multi-valued lattice is introduced for the detection of *mutual vacuity*, *i.e.*, the detection of sets of subformulas that are simultaneously vacuous in a formula. Model checking with this lattice outputs information about the largest vacuous subformulas of a formula in one run. But the lattice does not immediately lead to an efficient implementation. We approximate it by a simpler lattice, but with a similar model checking approach. Our algorithm, called VAQUOT, only outputs information about the individual vacuity of several fixed non-overlapping subformulas of a formula. In the worst case, the subformulas can be all the atomic propositions in the formula. We also introduce a refinement technique that runs our algorithm iteratively to find the largest vacuous subformulas of a formula. Thus, by approximation and refinement we find mutual vacuity that is hard to detect in one run with the lattice of [59].

To illustrate our approximation, consider verifying the following property of a traffic light controller: in every state, the light can be *red* or *yellow* of *green*, formalized in CTL as: $AG(\mathit{red} \vee \mathit{yellow} \vee \mathit{green})$. Suppose an error was made in modeling, and the variable for light is stuck at value ‘red’ in the model. If we check this formula on this model for the

mutual vacuity of atomic propositions, we find out that the formula is vacuous in subformula $(\mathit{yellow} \vee \mathit{green})$, *i.e.*, *yellow* and *green* are mutually vacuous. In other words, both *yellow* and *green* can be simultaneously replaced with false, and the formula still holds. The mutual vacuity detection with the lattice of [59] outputs $\{\{\mathit{yellow}, \mathit{green}\}\}$, as this is the only set of mutually vacuous propositions. In general, the mutual vacuity answer is a set of sets. If we check the same formulas for individual vacuity of the atomic propositions, as in the approach of Kupferman and Vardi, we find that each of *yellow* and *green* are vacuous, but we cannot conclude that the entire subformula $(\mathit{yellow} \vee \mathit{green})$ is vacuous. With our approximation, we obtain the same answer as Kupferman and Vardi. In contrast to the mutual vacuity answer, ours is $\{\{\mathit{yellow}\}, \{\mathit{green}\}\}$. Thus, our answer is exact (sound and complete) with respect to Kupferman and Vardi's definition of vacuity, but sound and incomplete (only an approximation) with respect to mutual vacuity.

To illustrate our refinement idea on this example, suppose the disjunction is right-associative. Then, the parse tree of the formula will have top-most subformulas (under root operator AG) *red* and $\mathit{yellow} \vee \mathit{green}$. We can introduce a fresh super-proposition called *blue*, to represent $\mathit{yellow} \vee \mathit{green}$, and run VAQUOT with propositions *red* and *blue*. We find that *blue* is vacuous, and we can stop without a need to run VAQUOT on propositions *yellow* and *green*. We thus detect the mutual vacuity of *yellow* and *green* cheaper than the mutual vacuity algorithm of [59] that explores all possible sets of mutually vacuous propositions from among *red*, *yellow*, and *green* in one run.

Given a model, and a CTL formula with a marked subset of its non-overlapping subformulas, VAQUOT checks whether the formula is true in the model, and reports all the vacuous subformulas from those marked. For any single run of VAQUOT we assume without loss of generality that the subformulas of interest are atomic propositions of the given formula. We can always replace those subformulas with fresh names that can act as atomic propositions. Following [69], we consider a proposition vacuous if it can be replaced by a constant (true or false) without affecting the value of the formula in the model. We treat different occurrences of the

same atomic proposition as different propositions. When the formula is true, VAQUOT reports whether all of its atomic propositions are vacuous (Vacuously True), none of them are vacuous (Non-Vacuously True), or some of the atomic propositions are vacuous (Vacuously True, followed by a list of the vacuous propositions). Similar answers are given when the formula is false. To find the largest subformulas that are vacuous in a formula, we can run VAQUOT iteratively by considering at each iteration the non-overlapping subformulas at the same level of the parse tree of the formula, while we scan the tree top-down in a breadth-first manner. As soon as we find a subformula as vacuous, we no longer scan its subtree (subformulas).

4.1.2 Query solving

For query solving, we noticed that, in the analysis of state-transition models, many problems reduce to questions of the type: "What are all the states that satisfy a property φ?" which are not readily expressed in temporal logic and usually require specialized algorithms, but we can formulate them as queries.

One example is finding the reachable states, which is often needed in a pre-analysis step to restrict further analysis only to those states. These states are typically found by computing a forward transitive closure of the transition relation [32]. We can see reachable states as solutions to $EF\ ?$.

Another example is the computation of "procedure summaries". A *procedure summary* is a relation between states, representing the input/output behavior of a procedure. The summary answers the question of which inputs lead to which outputs as a result of executing the procedure. They are computed in the form of "summary edges" in the control-flow graphs of programs [86, 7]. We can obtain procedure summaries by solving $EF\ ((\texttt{pc} = \texttt{PROC\_END}) \wedge ?)$, where $\texttt{pc} = \texttt{PROC\_END}$ holds in the return statement of the procedure.

Yet another example is the algorithm for finding *dominators/postdominators* in program analysis, proposed in [3]. A state t is a postdominator of a state s if all paths from s eventually reach t, and t is a dominator of s if all paths to s pass through t. Dominators/postdominators

are solutions to the query $AF\ ?$ (*i.e.*, what propositional formulas eventually hold on all paths). This insight gives us a uniform formulation of these problems and allows for easy creation of solutions to other, similar, problems. For example, a problem reported in genetics research [21, 44] called for finding *stable states* of a model, that are those states which, once reached, are never left by the system. This is easily formulated as $EFAG\ ?$, meaning "what are the reachable states in which the system will remain forever?".

With our point of view, we can characterize an important class of useful problems by a common requirement: solutions to queries are single states of the model. For example, a query $AF\ ?$ on the model in Figure 4.4 has solutions $(p \wedge \neg q \wedge r)$ and $(q \wedge r)$. The first corresponds to the state s_0 and is a state solution. The second corresponds to a set of states $\{s_1, s_2\}$ but neither s_1 nor s_2 is a solution by itself. When only state solutions are needed, we can formulate a restricted *state query* by constraining the solutions to be single states, rather than arbitrary propositional formulas (that represent *sets* of states). A naive state-query solving algorithm is to repeatedly substitute each state of the model for the placeholder, and return those for which the resulting CTL formula holds. This approach is linear in the size of the state space and in the cost of CTL model checking. While significantly more efficient than general query solving, this approach is not "fully" symbolic, since it requires many runs of a model-checker.

Similarly to vacuity detection, [60] proposes a lattice of query solutions and a multi-valued model checking algorithm over that lattice for solving general queries. We provide a symbolic algorithm, called TLQ, for solving the state queries that approximates the general query-solving approach of [60], by using only a simpler lattice of state solutions. We also describe a minimalist implementation which only modifies the interface of the model-checker NuSMV [32]. While the complexity of our approach is the same as in the corresponding naive approach, we show empirically that TLQ can perform better than the naive, using a case study from genetics [44]. We also describe an iterative refinement technique in which queries are asked about gradually more atomic propositions. We solve first the query over as few propositions as feasible, then use the solutions found with those propositions to restrict a next run that asks the

query over the atomic propositions just considered plus few others, and so on.

4.1.3 Outline

In Section 4.2, we provide additional background in lattice theory and multi-valued model checking. In Sections 4.3 and 4.4 we review the formulations of vacuity detection and query solving as multi-valued checks. Section 4.5 describes our general approximation framework. The particular approximations and algorithms for both problems, with their implementations and evaluations are in Sections 4.6 and 4.7. We conclude and discuss future work in Section 4.8.

4.2 Background

Lattice theory

Definition 3 (Finite lattice) *A* finite lattice *is a pair (*$\mathbf{L}, \sqsubseteq$*), where* $\mathbf{L}$ *is a finite set and* $\sqsubseteq$ *is a partial order on* $\mathbf{L}$*, such that every finite subset* $B \subseteq \mathbf{L}$ *has a least upper bound (called* join *and written* $\sqcup B$*) and a greatest lower bound (called* meet *and written* $\sqcap B$*).*

Definition 4 (Minimal, maximal elements) *An element* $b \in B \subseteq \mathbf{L}$ *is* minimal(maximal) *if for all* $a \in B$*, if* $a \sqsubseteq b (b \sqsubseteq a)$*, then* $a = b$*. For any finite lattice there exist 'top'* $\top = \sqcup\mathbf{L}$ *and 'bottom'* $\bot = \sqcap\mathbf{L}$*, that are the* maximum *and respectively* minimum *elements in the lattice.*

When the ordering $\sqsubseteq$ is clear from the context, we simply refer to the lattice as $\mathbf{L}$.

Definition 5 *A lattice is* distributive *if meet and join distribute over each other, i.e., for any* $a, b, c \in \mathbf{L}$*:*

$$\begin{aligned} a \sqcup (b \sqcap c) &= (a \sqcup b) \sqcap (a \sqcup c) \\ a \sqcap (b \sqcup c) &= (a \sqcap b) \sqcup (a \sqcap c) \end{aligned}$$

Definition 6 (DeMorgan Algebra) *A* De Morgan *algebra is a triple* $(\mathbf{L}, \sqsubseteq, \neg)$*, where* $(\mathbf{L}, \sqsubseteq)$ *is a finite distributive lattice and* $\neg$ *is any operation that is an involution, i.e.,* $\neg\neg\ell = \ell$*, and satisfies De Morgan laws.*

Propositional formulas form a lattice we commonly work with. For a set of atomic propositions P, let $\mathcal{F}(P)$ be the set of propositional formulas over P. For example, $\mathcal{F}(\{p\}) = \{\mathsf{true}, \mathsf{false}, p, \neg p\}$. This set forms a finite lattice ordered by implication (see Figure 4.1(a)). Since $p \Rightarrow \mathsf{true}$, p is under true in this lattice. Meet and join in this lattice correspond to logical operators $\wedge$ and $\vee$, respectively.

Definition 7 (Up-set, Down-set) *A subset $B \subseteq \mathbf{L}$ is called* upward closed *or an* up-set*, if for any $a, b \in \mathbf{L}$, if $b \in B$ and $b \sqsubseteq a$, then $a \in B$. In that case, B can be identified by the set N of its minimal elements, and we write $B = \uparrow N$.*

A subset $B \subseteq \mathbf{L}$ is called downward closed *or a* down-set*, if for any $a, b \in \mathbf{L}$, if $b \in B$ and $a \sqsubseteq b$, then $a \in B$. In that case, B can be identified by the set N of its maximal elements, and we write $B = \downarrow N$.*

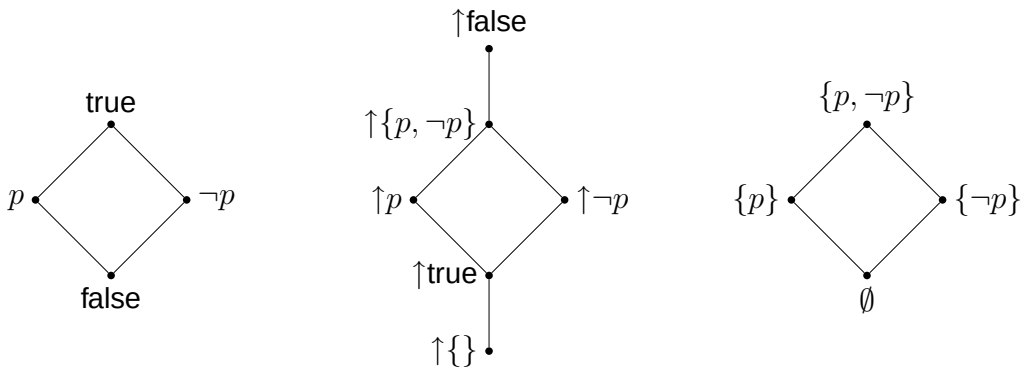
For example, for the lattice $(\mathcal{F}(\{p\}), \Rightarrow)$ shown in Figure 4.1(a), $\uparrow\{p, \neg p\} = \{p, \neg p, \mathsf{true}\}$. The set $\{p, \neg p\}$ is not an up-set, whereas $\{p, \neg p, \mathsf{true}\}$ is. For singletons, we write $\uparrow a$ for $\uparrow\{a\}$, and the same for down-sets.

Definition 8 (Up-set lattice) *For any $A \subseteq \mathcal{L}$, $\uparrow A = \uparrow M$, where M is the set of minimal elements in A. We write $\mathcal{U}(\mathbf{L})$ for the set of all upsets of $\mathbf{L}$, i.e., $A \subseteq \mathbf{L}$ iff $\uparrow A \in \mathcal{U}(\mathbf{L})$. $\mathcal{U}(\mathbf{L})$ is closed under union and intersection, and therefore forms a lattice ordered by set inclusion. We call $(\mathcal{U}(\mathbf{L}), \subseteq)$ the* up-set lattice *of $\mathbf{L}$.*

The up-set lattice of $\mathcal{F}(\{p\})$ is shown in Figure 4.1(b).

Definition 9 (Join-irreducible element) *An element j in a lattice $\mathbf{L}$ is* join-irreducible *if $j \neq \bot$ and j cannot be decomposed as the join of other lattice elements, i.e., for any x and y in $\mathbf{L}$, $j = x \sqcup y$ implies $j = x$ or $j = y$.*

Every element of a finite distributive lattice has a unique representation as a join of join-irreducible elements.



(a) (b) (c)

Figure 4.1: Lattices for $P = \{p\}$: (a) $(\mathcal{F}(P), \Rightarrow)$; (b) $(\mathcal{U}(\mathcal{F}(P)), \subseteq)$; (c) $(2^{\mathcal{M}(P)}, \subseteq)$.

For example, the join-irreducible elements of the lattice in Figure 4.1(a) are p and $\neg p$, and of the one in Figure 4.1(b) — $\uparrow$true, $\uparrow p$, $\uparrow \neg p$, and $\uparrow$false.

Definition 10 (Minterm) *In the lattice of propositional formulas $\mathcal{F}(P)$, a join-irreducible element is a conjunction in which every atomic proposition of P appears, positive or negated. Such conjunctions are called* minterms *and we denote their set by $\mathcal{M}(P)$.*

For example,

$$\mathcal{M}(\{p, q\}) = \{p \wedge q, p \wedge \neg q, \neg p \wedge q, \neg p \wedge \neg q\}.$$

Multi-valued CTL model checking

Consider the classical CTL fixpoint semantics given in Chapter 1. Recall that a formula φ holds in a Kripke structure M, written $M \models \varphi$, if it holds in the initial state, *i.e.*, $[\![\varphi]\!](s_0) = \mathsf{true}$. The complexity of model-checking a CTL formula φ on a Kripke structure M is $O(|M| \times |\varphi|)$, where $|M| = |S| + |R|$.

Multi-valued CTL model checking [26] is a generalization of model checking from a classical logic to an arbitrary *De Morgan* algebra. Conjunction and disjunction are the meet and

join operations of $(\mathbf{L}, \sqsubseteq)$, respectively. When the ordering and the negation operation of an algebra $(\mathbf{L}, \sqsubseteq, \neg)$ are clear from the context, we refer to it as $\mathbf{L}$. In our work, we only use a version of multi-valued model checking where the model remains classical, *i.e.*, both the transition relation and the atomic propositions are two-valued, where Boolean values true and false are replaced by the $\top$ and $\bot$ of $\mathbf{L}$, respectively. Only the properties are specified in a multi-valued extension of CTL over a given De Morgan algebra $\mathbf{L}$, called χCTL($\mathbf{L}$). The logic χCTL($\mathbf{L}$) has the same syntax as CTL, except that the allowed constants are all $\ell \in \mathbf{L}$. The semantics of χCTL($\mathbf{L}$) is analogous to that of CTL; $[\![\varphi]\!]$ is extended to $[\![\varphi]\!] : S \to \mathbf{L}$ as follows:

$$
\begin{aligned}
[\![\ell]\!](s) &\triangleq \ell, \text{ for } \ell \in \mathbf{L} \\
[\![a]\!](s) &\triangleq \top, \text{ if } a \in I(s) \text{ else } \bot \\
[\![\neg\varphi]\!](s) &\triangleq \neg[\![\varphi]\!](s) \\
[\![\varphi \wedge \psi]\!](s) &\triangleq [\![\varphi]\!](s) \sqcap [\![\psi]\!](s) \\
[\![\varphi \vee \psi]\!](s) &\triangleq [\![\varphi]\!](s) \sqcup [\![\psi]\!](s) \\
[\![EX\ \varphi]\!](s) &\triangleq \bigsqcup_{t \in S}(R(s,t) \sqcap [\![\varphi]\!](t)) \\
[\![EG\ \varphi]\!](s) &\triangleq [\![\nu Z.\varphi \sqcap EXZ]\!](s) \\
[\![E[\varphi\ U\ \psi]]\!](s) &\triangleq [\![\mu Z.\psi \sqcup (\varphi \sqcap EXZ)]\!](s)
\end{aligned}
$$

The complexity of model checking a χCTL($\mathbf{L}$) formula φ on a Kripke structure M is still $O(|M| \times |\varphi|)$ as for classical CTL, *provided that* meet, join, and quantification can be computed in constant time, which depends on the lattice [26].

4.3 Vacuity Detection as a Multi-Valued Check

We review here the lattice of mutual vacuity information and the symbolic mutual vacuity detection algorithm from [59].

The discussion is in terms of atomic propositions but it applies to any non-overlapping subformulas of a formula. We treat multiple occurrences of the same subformula as different subformulas. Let φ be a CTL formula with atomic propositions a, b, each occurring once,

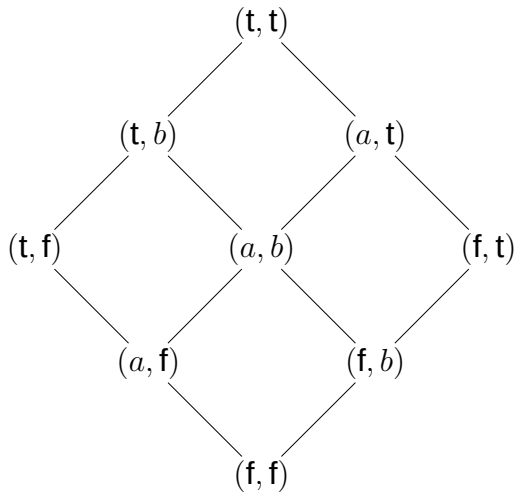


Figure 4.2: Lattice of replacements for (a, b).

positively. Thus, φ is monotonic in both a and b. According to the definition of vacuity of Kupferman and Vardi (see Chapter 3), in order to compute the value of φ in a given model and decide if it is vacuous in a, we need to model check witnesses $\varphi, \varphi[a \leftarrow \mathsf{true}], \varphi[a \leftarrow \mathsf{false}]$. In other words, the replacements for a are $a, \mathsf{true}, \mathsf{false}$, respectively. Similarly for b we have replacements $b, \mathsf{true}, \mathsf{false}$. These create the lattice $\mathbf{L} = \{\mathsf{false}, a, \mathsf{true}\} \times \{\mathsf{false}, b, \mathsf{true}\}$ of possible replacements for (a, b). These replacements are ordered according to the way they affect the value of φ. For example, if $(\mathsf{false}, \mathsf{false})$ makes φ false, (false, b) and $(\mathsf{false}, \mathsf{true})$ can only make φ *truer* an truer, since φ is monotonic in b and a stays the same; that is,

$$\varphi[a \leftarrow \mathsf{false}, b \leftarrow \mathsf{false}] \sqsubseteq \varphi[a \leftarrow \mathsf{false}] \sqsubseteq \varphi[a \leftarrow \mathsf{false}, b \leftarrow \mathsf{true}]$$

in the boolean order. Thus,

$$(\mathsf{false}, \mathsf{false}) \sqsubseteq (\mathsf{false}, b) \sqsubseteq (\mathsf{false}, \mathsf{true})$$

in the lattice of replacements. Lattice $\mathbf{L}$ is shown in Figure 4.2, where we use for short t,f for $\mathsf{true}, \mathsf{false}$.

For the same reason of monotonicity, if a replacement makes φ true, all the replacements above it in $\mathbf{L}$ make φ true as well. In other words, all the replacements in $\uparrow(x, y)$ make φ true if (x, y) makes it true. Similarly, if (x, y) makes φ false, all replacements in $\downarrow(x, y)$ make φ false.

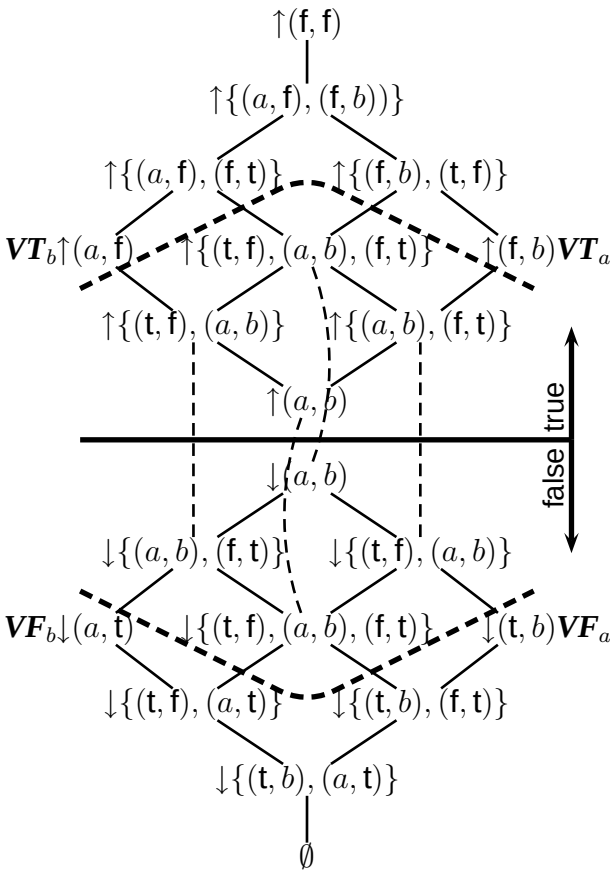


Figure 4.3: Mutual vacuity lattice of up-(down-)sets of replacements for (a, b).

This induces the lattice of sets of replacements that make φ true/false shown in Figure 4.3 and that we refer to as the *mutual vacuity lattice*. Note that we changed the presentation of the lattice from [59] to have the bottom half in term of down-sets of replacements that make the formula false, rather than upsets that make formula true. In [59], the lattice of sets is ordered by set inclusion. In our presentation, the upper half is ordered by set inclusion, whereas the bottom half is ordered by reverse set inclusion. The presentation we give here makes it easier to define our approximation later on. Even if in our representation the mutual vacuity lattice is not the up-set lattice for $\mathbf{L}$, it is still isomorphic to the up-set lattice, so we abuse notation and still denote it by $\mathcal{U}(\mathbf{L})$.

The elements of this lattice give information about the value of φ and the vacuity of a, b. All values in the upper half indicate φ is true, and all those in the bottom half are indicate

φ is false. Any value above or below the dotted boundaries in the two halves indicate some vacuity: if the value is above $\uparrow(a, \mathsf{f})$, it indicates that φ is vacuous in b: $\uparrow(a, \mathsf{f})$ includes all replacements where a stays and b is replaced by $\mathsf{false}, b, \mathsf{true}$, all making φ true, which by definition means b is vacuous; for this reason, we denote $\uparrow(a, \mathsf{f})$ by VT_b ('Vacuously True in b'); similarly, any value above $\uparrow(\mathsf{f}, b)$ indicates φ is vacuous in a, so $\uparrow(\mathsf{f}, b)$ is VT_a ('Vacuously True in a'). A similar discussion applies to values below the dotted boundary in the bottom half. Values $\uparrow\{(a, \mathsf{f}), (\mathsf{f}, b)\}$ and $\uparrow\{(a, \mathsf{t}), (\mathsf{t}, b)\}$ mean φ is vacuous in both a and b separately, while $\uparrow(\mathsf{f}, \mathsf{f})$ and $\uparrow(\mathsf{t}, \mathsf{t})$ show a, b are vacuous at the same time (mutually).

To compute the truth value and the vacuity of φ using this lattice, a new semantics is given to the atoms a and b in the formula, in terms of their truth values and vacuity as formulas by themselves, based on the following case analysis. Proposition a, as a formula by itself, is either true, and then it is vacuously true in b since b does not occur in a, or it is false, and still vacuously so in b. We can encode this symbolically as $(a \wedge VT_b) \vee (\neg a \wedge VF_b)$ or, equivalently, $a \wedge VT_b \vee VF_b$. To see why the two expressions are equivalent, consider the two possible values for a, and the fact that $\bot \preceq VF_b \preceq VT_b$: when a is true or $\top$, both expressions evaluate to VT_b, and when a is false or $\bot$, they both evaluate to VF_b). Similarly, for b, its truth and vacuity semantics is $b \wedge VT_a \vee VF_a$. Thus the algorithm for vacuity detection using the mutual vacuity lattice works by model checking

$$\varphi' = \varphi[a \leftarrow a \wedge VT_b \vee VF_b, b \leftarrow b \wedge VT_a \vee VF_a]$$

over this lattice.

A naive approach to detect mutual vacuity is: for every possible subset of atomic propositions, to replace all the propositions in the set simultaneously with constants false or true, and check the resulting formulas sequentially, as checking the witnesses for individual vacuity in the approach of Kupferman and Vardi. This requires a number of calls to a classical model checker that is exponential in the number of atomic propositions.

The correctness of this algorithm follows from the fact shown in [57] that model checking φ' is equivalent to checking classically $j \sqsubseteq \varphi'$, for all join-irreducibles j of the lattice. The

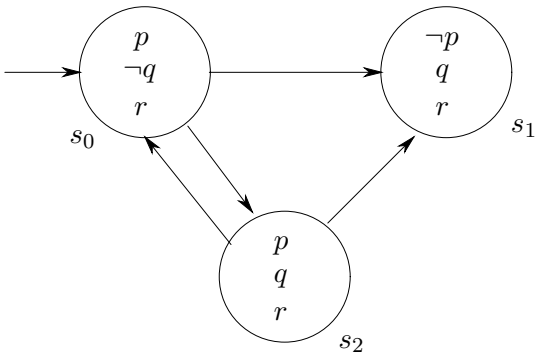


Figure 4.4: A simple Kripke structure.

intuition is that any value of the lattice can be guessed by asking 'yes/no' questions about how it compares to the join-irreducibles, since any value in the lattice can be written as a join of join-irreducible elements (see Chapter 3. Intuitively, this is the generalization to lattice orders of the game of guessing a natural number (the set of the naturals has a linear order) by asking repeatedly the question: 'is the number grater than x', and adjusting x accordingly in the next question. The $\sqsubseteq$ operator commutes with the temporal and propositional operators so that all checks $j \sqsubseteq \varphi'$ eventually reduce to the checks of witnesses as in the naive approach. For example, VT_b is a join-irreducible of the mutual vacuity lattice. When we take $j \sqsubseteq \varphi'$, we get

$$\varphi[a \leftarrow (VT_b \sqsubseteq a) \wedge (VT_b \sqsubseteq VT_b) \vee (VT_b \sqsubseteq VF_b), b \leftarrow (VT_b \sqsubseteq b) \wedge (VT_b \sqsubseteq VT_a) \vee (VT_b \sqsubseteq VF_a)]$$

which is $\varphi[b \leftarrow \mathsf{false}]$, since a, b map to either $\top$ or $\bot$. This is one of the witnesses!

Implementing this algorithm efficiently depends on finding an efficient implementation of the lattice operations with up-(down-)sets. It is not clear how these could be implemented in constant-time. Also, it is not clear how to compute the negation efficiently, in either representation: ours, with up- and down-sets, or that of [59] with all up-sets.

4.4 Query Checking as a Multi-Valued Check

Let M be a Kripke structure with a set A of atomic propositions. Recall that a CTL query, denoted by $\varphi[?]$, is a CTL formula containing a *placeholder* "$?$" for a propositional subformula

(over the atomic propositions in A). The CTL formula obtained by substituting the placeholder in $\varphi[?]$ by a formula $\alpha \in \mathcal{F}(A)$ is denoted by $\varphi[\alpha]$. A formula α is a solution to a query if its substitution into the query results in a CTL formula that holds on M, *i.e.*, if $M \models \varphi[\alpha]$. For example, $(p \wedge \neg q \wedge r)$ and $(q \wedge r)$ are among the solutions to the query $AF\ ?$ on the model of Figure 4.4, whereas $\neg r$ is not.

In our work, we consider queries in *negation normal form*, where negation is applied only to the atomic propositions, or to the placeholder. We further restrict our attention to queries with a single placeholder, although perhaps with multiple occurrences. For a query $\varphi[?]$, a substitution $\varphi[\alpha]$ means that all occurrences of the placeholder are replaced by α. For example, if $\varphi[?] = EF\ (? \wedge AX\ ?)$, then $\varphi[p \vee q] = EF\ ((p \vee q) \wedge AX\ (p \vee q))$. We assume that occurrences of the placeholder are either non-negated everywhere, or negated everywhere, *i.e.*, the query is either *positive* or *negative*, respectively. For now, we limit the presentation to positive queries.

The general CTL query-solving problem is: given a CTL query on a model, find all its propositional solutions. For instance, the answer to the query $AF\ ?$ on the model in Figure 4.4 is the set consisting of $(p \wedge \neg q \wedge r), (q \wedge r)$ and every other formula implied by these, including p, $(q \vee r)$, and true. If α is a solution to a query, then any β such that $\alpha \Rightarrow \beta$ (*i.e.*, any weaker β) is also a solution, due to the monotonicity of positive queries [25]. Thus, the set of all possible solutions is an up-set; it is sufficient for the query-checker to output the strongest solutions, since the rest can be inferred from them.

One can restrict a query to a subset $P \subseteq A$ [17]. We then denote the query by $\varphi[?P]$, and its solutions become formulas in $\mathcal{F}(P)$. For instance, solving $AF\ ?\{p, q\}$ on the model of Figure 4.4 should result in $(p \wedge \neg q)$ and q as the strongest solutions, together with all those implied by them. We write $\varphi[?]$ for $\varphi[?A]$.

If P consists of n atomic propositions, there are 2^{2^n} possible distinct solutions to $\varphi[?P]$. A "naive" method for finding all solutions would model check $\varphi[\alpha]$ for every possible propositional formula α over P, and collect all those α's for which $\varphi[\alpha]$ holds in the model. The

complexity of this naive approach is 2^{2^n} times that of usual model-checking.

A symbolic algorithm for solving the general query-solving problem was described in [60] and has been implemented in the TLQSolver tool [28]. We review this approach below.

Since an answer to $\varphi[?P]$ is an upset, the up-set lattice $\mathcal{U}(\mathcal{F}(P))$ is the space of all possible answers [17]. For instance, the lattice for $AF\ ?\{p\}$ is shown in Figure 4.1(b). In the model in Figure 4.4, the answer to this query is $\{p, \mathsf{true}\}$, encoded as $\uparrow\{p\}$, since p is the strongest solution.

Symbolic query solving is implemented by model checking over the up-set lattice. The algorithm is based on a state semantics of the placeholder. Suppose query $?\{p\}$ is evaluated in a state s. Either p holds in s, in which case the answer to the query should be $\uparrow p$, or $\neg p$ holds, in which case the answer is $\uparrow\neg p$. Thus we have:

$$[\![?\{p\}]\!](s) = \begin{cases} \uparrow p & \text{if } p \in I(s), \\ \uparrow\neg p & \text{if } p \notin I(s). \end{cases}$$

This case analysis can be logically encoded by the formula $(p \wedge \uparrow p) \vee (\neg p \wedge \uparrow\neg p)$.

Let us now consider a general query $?P$ in a state s (where $?$ ranges over a set of atomic propositions P). We note that the case analysis corresponding to the one above can be given in terms of minterms. Minterms are the strongest formulas that may hold in a state; they also are mutually exclusive and complete — exactly one minterm j holds in any state s, and then $\uparrow j$ is the answer to $?P$ at s. This semantics is encoded in the following translation of the placeholder:

$$\mathcal{T}(?P) = \bigvee_{j \in \mathcal{M}(P)} (j \wedge \uparrow j).$$

The symbolic algorithm is defined as follows: given a query $\varphi[?P]$, first obtain $\varphi[\mathcal{T}(?P)]$, which is a χCTL formula (over the lattice $\mathcal{U}(\mathcal{F}(P))$), and then model check this formula. The semantics of the formula is given by a function from S to $\mathcal{U}(\mathcal{F}(P))$, as described in Section 4.2. Thus model checking this formula results in a value from $\mathcal{U}(\mathcal{F}(P))$. That value was shown in [60] to represent all propositional solutions to $\varphi[?P]$. For example, the query $AF\ ?$ on the

model of Figure 4.4 becomes

$$\begin{aligned} AF \quad & ((p \wedge q \wedge r \wedge \uparrow(p \wedge q \wedge r)) \vee \\ & (p \wedge q \wedge \neg r \wedge \uparrow(p \wedge q \wedge \neg r)) \vee \\ & (p \wedge \neg q \wedge r \wedge \uparrow(p \wedge \neg q \wedge r)) \vee \\ & (p \wedge \neg q \wedge \neg r \wedge \uparrow(p \wedge \neg q \wedge \neg r)) \vee \\ & \ldots). \end{aligned}$$

The result of model-checking this formula is $\uparrow\{p \wedge \neg q \wedge r, q \wedge r\}$.

The complexity of this algorithm is the same as in the naive approach. In practice, however, TLQSolver was shown to perform better than the naive algorithm [60, 28].

4.5 Approximations

The efficiency of model checking over a lattice is determined by the complexity of the lattice operations. In this section, we show a general approximation framework for reasoning over any lattice of sets. The framework defines sufficient conditions for finding a simpler lattice from a complex one, so that model checking over the simpler lattice gives an approximation of the answer over the complex one. This allows to still obtain partial solutions to problems such as vacuity detection and query solving that have intractable lattices in general. We later show how our approximations for these two problems are instances of this general framework.

Let U be any finite set. Its powerset lattice is $(2^U, \subseteq)$. Let $(\mathbf{L}, \subseteq)$ be any sublattice of the powerset lattice, *i.e.*, $\mathbf{L} \subseteq 2^U$.

Definition 11 (Approximation) *A function $f : \mathbf{L} \to 2^U$ is an* approximation *if:*

1. *it satisfies $f(B) \subseteq B$ for any $B \in \mathbf{L}$ (i.e., $f(B)$ is an under-approximation of B), and*
2. *it is a lattice homomorphism, i.e., it respects the lattice operations: $f(B \cap C) = f(B) \cap f(C)$, and $f(B \cup C) = f(B) \cup f(C)$.*

From the definition of f, the image $f(\mathbf{L})$ of $\mathcal{L}$ through f is a sublattice of 2^U, having $f(\top)$ and $f(\bot)$ as its maximum and minimum elements, respectively.

We consider an approximation to be correct if it is preserved by model checking: reasoning over the smaller lattice is the approximation of reasoning over the larger one. Let φ be a *positive* χCTL(**L**) formula, *i.e.*, which does not contain negation. We define its translation $\mathcal{A}(\varphi)$ into $f(\mathbf{L})$ recursively on the structure of the formula as expected, eventually replacing any constant $B \in \mathbf{L}$ occurring in φ by $f(B)$. The following theorem simply states that the result of model checking $\mathcal{A}(\varphi)$ is the approximation of the result of model checking φ. Its proof follows by structural induction from the semantics of χCTL, and uses the fact that approximations are homomorphisms. [66] proves a similar result, albeit in a somewhat different context.

Theorem 1 (Correctness of approximations) *Let M be a classical Kripke structure, $\mathbf{L}$ be a finite distributive lattice of sets, f be an approximation function on $\mathbf{L}$, and φ be a positive χCTL(**L**) formula. Let $\mathcal{A}(\varphi)$ be the translation of φ into $f(\mathbf{L})$. Then for any state s of M,*

$$f([\![\varphi]\!](s)) = [\![\mathcal{A}(\varphi)]\!](s).$$

Proof:

By induction on the structure of φ.

For $B \in \mathbf{L}$, with $f(B) = C \subseteq B$, $f([\![B]\!](s)) = f(B) = C = [\![C]\!](s) = C$.

For $a \in A$, $f([\![a]\!](s)) = [\![a]\!](s) = [\![\mathcal{A}(a)]\!](s)$ (approximation preserves $\top$ and $\bot$, which are the possible values of $[\![a]\!](s)$, and $\mathcal{A}(a) = a$).

For $\varphi \wedge \psi$ and $EX\varphi$, the claim follows from the fact that f preserves $\sqcap$ and $\sqcup$ (second condition of Definition 11) and that the transition relation of the model is classical:

$$f([\![\varphi \wedge \psi]\!](s)) = f([\![\varphi]\!](s) \sqcap [\![\psi]\!](s)) = [\![\mathcal{A}(\varphi)]\!](s) \sqcap [\![\mathcal{A}(\psi)]\!](s) = [\![\mathcal{A}(\varphi \wedge \psi)]\!](s)$$

$$f([\![EX\varphi]\!](s)) = \bigsqcup_{t \in S} \{R(t,s) \sqcap [\![\mathcal{A}(\varphi)]\!](s)\} = [\![\mathcal{A}(EX\varphi)]\!](s)$$

(the translation applies recursively on the structure of formulas).

The cases for the fixpoints are similar, noting that a fixpoint is computed in a finite number of iterations stating from $Z = \top$ for the greatest fixpoint (EG), or $Z = \bot$ for the least fixpoint EU. If we expand those iterations for the fixpoints, we get formulas with the other operators already considered in this proof. □

Note that since models are classical and formulas are positive, we do not require the lattice to be a DeMorgan algebra. We leave the treatment of negation dependent on the particular instance and application of the framework.

4.6 Approximation and Refinement for Vacuity Detection

4.6.1 Vacuity detection algorithm

We first present our vacuity lattice and our algorithm VAQUOT independently, then show how they are an instance of the general approximation framework applied to the vacuity lattice from Section 4.3. As before, the presentation is in terms of atomic propositions, but it applies to any subset of non-overlapping subformulas of a formula.

As in the approach described in Section 4.3, the basis of VAQUOT is a multi-valued "vacuity" lattice and a translation of CTL formulas into this lattice. Instead of the formulas being interpreted over the Boolean lattice $(\{\mathsf{true}, \mathsf{false}\}, \leq)$, they are interpreted over the *vacuity lattice* $\mathbf{L}_V = (\{\mathsf{true}, \mathsf{false}\} \times 2^A, \sqsubseteq)$, where 2^A is the powerset of the set A of atomic propositions. An element $(t, s) \in \mathbf{L}_V$ is a possible result of vacuity detection, showing that the formula has truth value t, and the largest subset of its atomic propositions that are vacuous is s. For any $u, v \in 2^A$, $(\mathsf{false}, u) \sqsubseteq (\mathsf{true}, v)$, $(\mathsf{true}, u) \sqsubseteq (\mathsf{true}, v)$ iff $u \subseteq v$, and $(\mathsf{false}, u) \sqsubseteq (\mathsf{false}, v)$ iff $v \subseteq u$ (note the reversal of set inclusion). The top element of $\mathbf{L}_V$ is (true, A), or Vacuously True in all propositions. The bottom element is (false, A), or Vacuously False in all propositions. The vacuity lattices for two and three atomic propositions a, b, c are depicted in Figure 4.5.

In VAQUOT, we replace each atomic proposition a of φ by $((a \wedge \mathit{VT}_{A \setminus a}) \vee \mathit{VF}_{A \setminus a})$, where

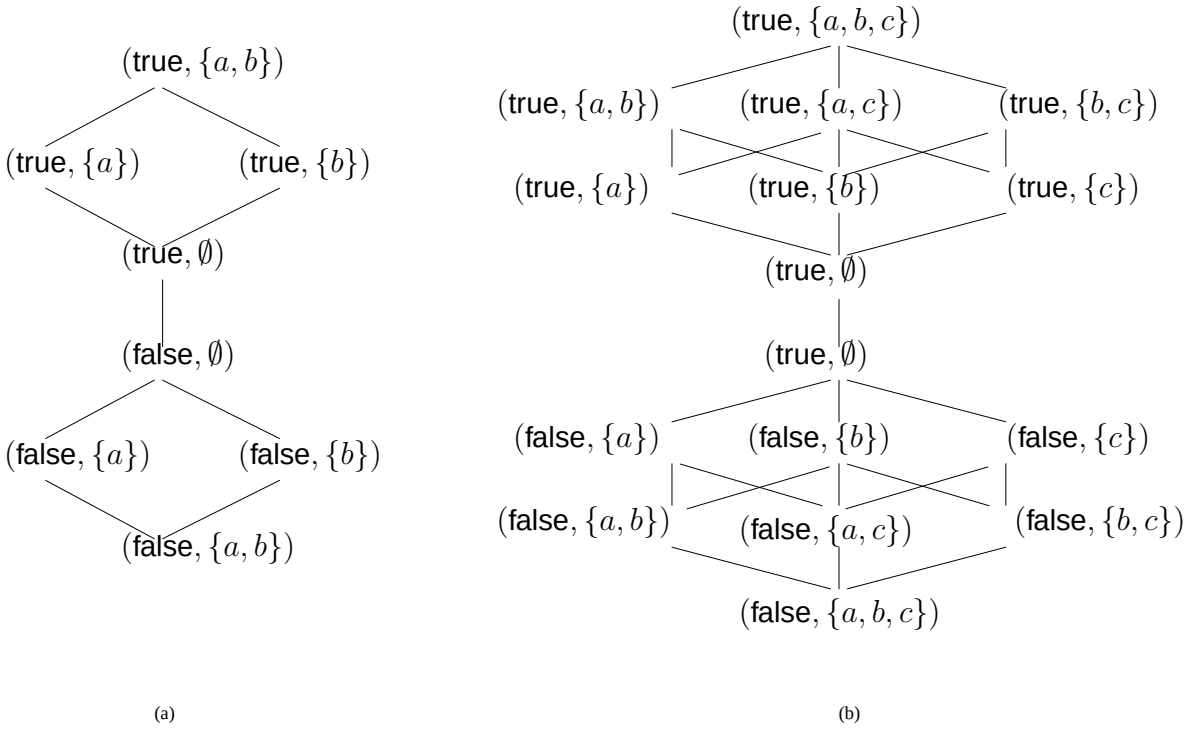


Figure 4.5: Vacuity lattices for a) two and b) three atomic propositions.

$VT_{A\setminus a}$ and $VF_{A\setminus a}$ denote lattice values $(\mathsf{true}, A \setminus \{a\})$, and $(\mathsf{false}, A \setminus \{a\})$, respectively. All replacements are done simultaneously. The resulting multi-valued formula is then model checked.

4.6.2 Correctness of approximation

Our lattice is isomorphic to the lattice obtained from the mutual vacuity lattice $\mathcal{U}(\mathbf{L})$ from Section 4.3 by an approximation that keeps from every set of $\mathcal{U}(\mathbf{L})$ only the elements representing independent vacuous propositions. If the set of atomic propositions is $A = \{a_1, \ldots, a_k\}$, for all $i = 1, \ldots, k$, the replacement $(a_1, \ldots, a_{i-1}, \mathsf{false}, a_{i+1}, \ldots, a_k)$ represents 'Vacuously True in a_i', that we denote by VT_i. Similarly, $(a_1, \ldots, a_{i-1}, \mathsf{true}, a_{i+1}, \ldots, a_k)$ represents 'Vacuously False in a_i', denoted by VF_i. Let $VT = \{VT_i \mid i = 1, \ldots, k\}$, and $VF = \{VF_i \mid i = 1, \ldots, k\}$.

Formally, we define the approximation function $f_V : \mathcal{U}(\mathbf{L}) \to 2^{VT} \cup 2^{VF}$ by:

$$f_V(W) = \begin{cases} W \cap VT & \text{if } W \text{ is an up-set} \\ W \cap VF & \text{if } W \text{ is a down-set.} \end{cases}$$

The isomorphism simply maps a set of replacements of the form $(a_1, \ldots, a_{i-1}, \mathsf{false}, a_{i+1}, \ldots, a_k)$ or $(a_1, \ldots, a_{i-1}, \mathsf{true}, a_{i+1}, \ldots, a_k)$ to (true, B) or (false, B), respectively, where B is the set of propositions whose vacuity the replacements represent. Formally, $g : 2^{VT} \cup 2^{VF} \to \mathbf{L}_V$ is defined by

$$g(U) = \begin{cases} (\mathsf{true}, \{a_i \mid i \in T\}) & \text{if } U \subseteq VT, T = \{i \in \{1, \ldots, k\} \mid VT_i \in U\} \\ (\mathsf{false}, \{a_i \mid i \in T\}) & \text{if } U \subseteq VF, T = \{i \in \{1, \ldots, k\} \mid VF_i \in U\}. \end{cases}$$

Theorem 2 (Correctness of vacuity approximation with VAQUOT**)** *Let M be a Kripke structure, and φ a CTL formula (not necessarily positive), with translations φ' into the mutual vacuity lattice $\mathcal{U}(\mathbf{L})$ and φ'' into our vacuity lattice $\mathbf{L}_V$. If the result of model checking φ' over the mutual vacuity lattice $\mathcal{U}(\mathbf{L})$ is the set l of replacements indicating that the formula is true/false with (independent or mutual) vacuity of a subset B of propositions, the result of* VAQUOT *checking φ'' is (true, B) or (false, B), indicating that the formula is true/false respectively, and propositions in B are independently vacuous.*

Proof:

The approximation f_V, as defined, satisfies the conditions in the general approximation framework (Section 4.5, as it maps sets to their subsets and preserves set operations. The isomorphism introduces truth values that do not affect the approximation, so they also respect the operations, *including negation*. Thus, Theorem 1 applies, and we obtain the correctness of our algorithm, stating that the result of VAQUOT correctly indicates the truth value of φ and the independent vacuity of its atomic propositions. □

Intuitively, by approximation we lose the information about the *mutuality* of the vacuity. We later show how we can recover some of this information running VAQUOT iteratively with refinement of subformulas. Our vacuity lattice leads immediately to an efficient implementation

with a classical model checker, as we describe next. Thus, it provides a feasible approximation to the otherwise very hard problem of detecting mutual vacuity.

4.6.3 Implementation

In our implementation, we encode each value of the vacuity lattice $\mathbf{L}_V$ as a 32-bit word. The least-significant bit represents the truth: 1 for true, 0 for false. The other bits represent the vacuity: 0 for vacuous, 1 for non-vacuous. For instance, for a formula φ with atomic propositions a, b, c, the lattice value $(\mathsf{true}, \{a, c\})$ is represented by the word $00\ldots00101$, where the rightmost bits 0101 mean, respectively, that a and c are vacuous, b is not, and the truth value is true. Thus, lattice operations can be efficiently implemented bitwise. The fixed word length, which could be increased from 32 to 64 or 128, limits the number of atomic propositions in the formulas we can check efficiently to 31, 63, 127, respectively. Bit vectors of arbitrary length could be used, at the cost of increasing the complexity of lattice operations.

The implementation of VAQUOT is built on top of NuSMV, which uses the CUDD package for the implementation of binary decision diagrams (BDDs) [32]. We have implemented multi-valued decision diagrams using CUDD ADDs which allow integers in their leaves, and changed the interface between NuSMV and CUDD so that our multi-valued operations are performed instead of their BDD counterparts. These modifications do not affect the complexity of decision diagram operations or fixpoint computations, but they may affect performance, since the decision diagrams may be larger. Our changes are compatible with the various NuSMV optimizations (*e.g.*, cone of influence, dynamic reordering, partitioning). The tool is available as a patch for NuSMV v. 2.1.2, from `www.cs.toronto.edu/fm/vaquot.html`.

4.6.4 Experiments

A few experiments comparing VAQUOT with basic model checking and with a naive approach to vacuity detection are reported in Table 4.1. The naive approach according to Kupferman

Table 4.1: Experimental results with VAQUOT.

Model	Formulas		Basic MC		Naive VD		VAQUOT
	Total	Vacuous	Memory	Time	Witnesses	Time	
`elevator3l`	45	26	43.5	1690.16	399	5228.94	1441.22
`guidance`	23	16	10	54.18	244	306.99	274.81
`production -cell`	15	15	7.2	42.41	187	228.87	184.08
`abp10`	4	3	10.6	83.18	26	316.63	304.51
`fgs5`	6	2	106	189.57	82	239.04	191.92
`msi_wtrans`	15	3	10.3	30.21	81	53.63	83.98
`luckySeven`	4	0	12.9	469.33	20	1257.11	842.48
`eisenberg`	5	4	3	11.31	25	35.77	39.77
`ticTacToe`	42	3	8.9	15.81	363	68.72	102.51

and Vardi consists of separately replacing each atomic proposition by true and then by false and check the resulting formulas, in addition to the original formula; all these formulas are called *witnesses*. The number of witnesses reported in Table 4.1 is the actual number of formulas checked in the naive approach, which we implemented on top of NuSMV as well. The experiments were performed on a Dell PC with a 2.4 GHz Intel Celeron CPU and 512 MB of RAM, running Linux 2.4.20. Models `guidance`, `production-cell`, `abp10`, and `msi_wtrans`, and most of their properties are from the NuSMV distribution. `elevator3l` is a model of a three-floor elevator system written by a student taking the Automated Verification class at Univ. of Toronto, and `fgs5` is a proprietary model for a flight-guidance system. Models `luckySeven`, `eisenberg`, and `ticTacToe` are SMV translations of their Verilog counterparts distributed with the VIS model checker. For each model, we report the total number of formulas checked and how many were found vacuous (Formulas), the total memory (in MB) and time (in seconds) used by model-checking without vacuity detection (Basic MC), the total number of witnesses and the time used by the naive vacuity detection (Naive VD), and the

running time of VAQUOT. As it can be seen, VAQUOT performs better than the naive approach in most cases, and by a considerable margin in some: our algorithm avoids much of the redundant work performed by the naive approach. In the cases where VAQUOT performs worse, we observed that the sizes of the decision diagrams are the bottleneck, and we are investigating ways to overcome this. It may seem surprising that such many formulas were found vacuous for the `elevator3l` and `guidance` models, and not for the others. A reason for this would be that those providing the properties for these models did not understand the models well: in the former case, the student did not write a good model, and did not formalize the properties correctly; in the latter case, the person writing the properties and submitting them with the model to the NuSMV archive was not very familiar with the model or the application domain (as can be inferred from the case documentation in the NuSMV archive).

4.6.5 Refinement

VAQUOT can detect vacuity in any subset of non-overlapping subformulas of a formula. Vacuity of larger subformulas is more useful to be reported to users than vacuity of smaller subformulas. Thus, it is important to find the largest subformulas that are vacuous in a formula. This is the main motivation behind the mutual vacuity defined by [59]. We can solve this problem using VAQUOT iteratively on gradually larger subformulas. We proceed by scanning the parse tree of the formula (built by the model checker) top-down in breadth-first manner. At each level, we consider the set of subformulas at that level, which are non-overlapping. We replace them with fresh names and run VAQUOT on them as if they were the atomic propositions. For the subformulas found vacuous among them, we do not need to explore their subtrees further, and we report them. For those not found vacuous, we proceed deeper in the tree and repeat the process. This iterative refinement technique is less prone to suffer from BDD explosion if it prunes much of the parse tree.

4.6.6 Comparison with related work

The method of [84] is the closest to ours, from related work. In [84], witnesses are generated and checked in parallel and compositionally, by a bottom-up exploration of the parse tree of a formula, with explicit caching of intermediate results. The representation of witnesses is explicit as well. All these are implicit in the multi-valued decision diagrams in our implementation. True and false formulas are treated differently, whereas VAQUOT handles both uniformly in one pass. Extensive experiments and comparisons between the two methods remain for future work; the results shown in Table 4.1, specifically, for the last three examples (used also in [84]), indicate that both tools exhibit a similar improvement over the naive approach, but for `eisenberg` and `ticTacToe`, VAQUOT found more vacuous passes.

Complementary to our vacuity checking of CTL formulas using BDD-based techniques, the work of [92] addresses vacuity checking of LTL formulas, implemented using SAT-based methods. In a parallel development, [27] re-examines the meaning of vacuity in terms of system versus environment behavior, and argues that current vacuity checking methodology produces too many false positives, that is, cases of vacuity that do not indicate problems. As an alternative, that work proposes checking when formulas pass/fail solely due to errors in the environment model, and shows on a realistic case study that this new methodology discovers truly problematic cases of vacuity. Similar concerns are addressed in [31, 14].

4.7 Approximation and Refinement for Query Solving

4.7.1 State solutions to queries

Without loss of generality, we consider only CTL formulas in *negation normal form*, where negation is applied only to atomic propositions [35].

Let M be a Kripke structure with a set A of atomic propositions. In general query solving, solutions to queries are arbitrary propositional formulas. On the other hand, for *state queries*,

solutions are restricted to be single states. To represent a single state, a propositional formula needs to be a minterm over A. In *symbolic* model checking, any state s of M is uniquely represented by the minterm that holds in s. For example, in the model of Figure 4.4, state s_0 is represented by $(p \wedge \neg q \wedge r)$, state s_2 by $(p \wedge q \wedge r)$, etc. Thus, for a state query, an answer to the query is a set of minterms, rather than an upset of propositional formulas. For instance, for the query $AF\ ?$, on the model of Figure 4.4, the state-query answer is $\{p \wedge \neg q \wedge r\}$, whereas the general query answer is $\uparrow\{r \wedge q, p \wedge \neg q \wedge r\}$. While it is still true that if j is a solution, everything in $\uparrow j$ is also a solution, we no longer view answers as upsets, since we are interested only in minterms, and j is the only minterm in the set $\uparrow j$ (minterms are incomparable by implication). We can thus formulate state-query solving as *minterm-query solving*: given a CTL query on a model, find all its minterm solutions. We show how to solve this for any query $\varphi[?P]$, and any subset $P \subseteq A$. When $P = A$, the minterms obtained are the state solutions.

Given a query $\varphi[?P]$, a naive algorithm would model check $\varphi[\alpha]$ for every minterm α. If n is the number of atomic propositions in P, there are 2^n possible minterms, and this algorithm has complexity 2^n times that of model-checking. Minterm query solving is thus much easier to solve than general query solving.

Of course, any algorithm for general query solving, such as the symbolic approach described in Section 4.4, solves minterm queries as well: from the answer with all solutions, we can extract only those which are minterms. This approach, however, is much more expensive than needed. Below, we propose a method that is tailored to just minterm-query solving, while remaining symbolic.

4.7.2 Minterm-query solving

Since an answer to a minterm query is a set of minterms, the space of all answers is the powerset $2^{\mathcal{M}(P)}$ that forms a lattice ordered by set inclusion. For example, the lattice $2^{\mathcal{M}(\{p\})}$ is shown in Figure 4.1(c). Our symbolic algorithm evaluates queries over this lattice. We first adjust the semantics of the placeholder to minterms. Suppose we evaluate $?\{p\}$ in a state s. Either p

holds in s, and then the answer should be $\{p\}$, or $\neg p$ holds, and then the answer is $\{\neg p\}$. Thus, we have

$$[\![?\{p\}]\!](s) = \begin{cases} \{p\} & \text{if } p \in I(s), \\ \{\neg p\} & \text{if } p \notin I(s). \end{cases}$$

This is encoded by the formula $(p \wedge \{p\}) \vee (\neg p \wedge \{\neg p\})$. In general, for a query $?P$, exactly one minterm j holds in s, and in that case $\{j\}$ is the answer to the query. This gives the following translation of placeholder:

$$\mathcal{A}_m(?P) \triangleq \bigvee_{j \in \mathcal{M}(P)} (j \wedge \{j\}).$$

Our minterm-query solving algorithm, TLQ, is now defined as follows: given a query $\varphi[?P]$ on a model M, compute $\varphi[\mathcal{A}_m(?P)]$, and then model check this over $2^{\mathcal{M}(P)}$.

For example, for $AF\ ?$, on the model of Figure 4.4, we model check

$$\begin{aligned} AF \quad & ((p \wedge q \wedge r \wedge \{p \wedge q \wedge r\}) \vee \\ & (p \wedge q \wedge \neg r \wedge \{p \wedge q \wedge \neg r\}) \vee \\ & (p \wedge \neg q \wedge r \wedge \{p \wedge \neg q \wedge r\}) \vee \\ & (p \wedge \neg q \wedge \neg r \wedge \{p \wedge \neg q \wedge \neg r\}) \vee \\ & \ldots), \end{aligned}$$

and obtain the answer $\{p \wedge \neg q \wedge r\}$, that is indeed the only minterm solution for this model.

4.7.3 Correctness of approximation

To prove our algorithm correct, we need to show that its answer is the set of all minterm solutions. We prove this claim by relating our algorithm to the general algorithm in Section 4.4. We show that, while the general algorithm computes the set $B \in \mathcal{U}(\mathcal{F}(P))$ of all solutions, ours results in the subset $N \subseteq B$ that consists of only the minterms from B. We first establish an approximation mapping from $\mathcal{U}(\mathcal{F}(P))$ to $2^{\mathcal{M}(P)}$ that, for any upset $B \in \mathcal{U}(\mathcal{F}(P))$, returns the subset $N \subseteq B$ of minterms.

Definition 12 (Minterm approximation) *Let P be a set of atomic propositions.* Minterm approximation $f_m : \mathcal{U}(\mathcal{F}(P)) \rightarrow 2^{\mathcal{M}(P)}$ *is* $f_m(B) \triangleq B \cap \mathcal{M}(P)$, *for any* $B \in \mathcal{U}(\mathcal{F}(P))$.

With this definition, $\mathcal{A}_m(?P)$ is obtained from $\mathcal{T}(?P)$ by replacing $\uparrow j$ with $f_m(\uparrow j) = \{j\}$. The minterm approximation preserves set operations; this follows immediately from the fact that any set of propositional formulas can be partitioned into minterms and non-minterms.

Proposition 1 *The minterm approximation $f_m : \mathcal{U}(\mathcal{F}(P)) \rightarrow 2^{\mathcal{M}(P)}$ is a lattice homomorphism, i.e., it preserves the set operations: for any $B, B' \in \mathcal{U}(\mathcal{F}(P))$, $f_m(B) \cup f_m(B') = f_m(B \cup B')$ and $f_m(B) \cap f_m(B') = f_m(B \cap B')$.*

By Proposition 1, and since model checking is performed using only set operations, we can show that the approximation preserves model-checking results. Model checking $\varphi[\mathcal{A}_m(?P)]$ is the minterm approximation of checking $\varphi[\mathcal{T}(?P)]$. In other words, our algorithm results in set of all minterm solutions, which concludes the correctness argument.

Theorem 3 (Correctness of minterm approximation) *For any state s of M,*

$$f_m([\![\varphi[\mathcal{T}(?P)]]\!](s)) = [\![\varphi[\mathcal{A}_m(?P)]]\!](s).$$

Proof:

The claim is a corollary to Theorem 1. Our minterm approximation satisfies condition (1) of Definition 11, since $f_m(B) = B \cap \mathcal{M}(P) \subseteq B$, and it also satisfies condition (2) by Proposition 1. Thus, f_m is an approximation to which Theorem 1 applies, yielding Theorem 3. □

In summary, for $P = A$, we have the following correct symbolic state-query solving algorithm : given a query $\varphi[?]$ on a model M, translate it to $\varphi[\mathcal{A}_m(?A)]$, and then model check this over $2^{\mathcal{M}(A)}$.

The worst-case complexity of TLQ is the same as that of the naive approach. With an efficient encoding of the approximate lattice, however, our approach can outperform the naive one in practice, as we show in Section 4.7.7.

4.7.4 Implementation

Although TLQ is defined as model checking over a lattice, we can implement it using a classical symbolic model checker. This is done by encoding the lattice elements in $2^{\mathcal{M}(P)}$ such that lattice operations are already implemented by a symbolic model checker. The key observation is that the lattice $(2^{\mathcal{M}(P)}, \subseteq)$ is isomorphic to the lattice of propositional formulas $(\mathcal{F}(P), \Rightarrow)$. This can be seen, for instance, by comparing the lattices in Figures 4.1(a) and 4.1(c). Thus, the elements of $2^{\mathcal{M}(P)}$ can be encoded as propositional formulas, and the operations become propositional disjunction and conjunction. A symbolic model checker, such as NuSMV [32], which we used in our implementation, already has data structures for representing propositional formulas and algorithms to compute their disjunction and conjunction — BDDs [94]. The only modifications we made to NuSMV were parsing the input and reporting the result.

While parsing the queries, we implemented the translation $\mathcal{A}_m$ defined in Section 4.7.2. In this translation, for every minterm j, we give a propositional encoding to $\{j\}$. We cannot simply use j to encode $\{j\}$. The lattice elements need to be *constants* with respect to the model, and j is not a constant — it is a propositional formula that contains model variables. We can, however, obtain an encoding for $\{j\}$, by renaming j to a similar propositional formula over fresh variables. For instance, we encode $\{p \wedge \neg q \wedge r\}$ as $x \wedge \neg y \wedge z$. The lattice operations are correctly implemented with this encoding since they are Boolean set operations that are implemented as Boolean formula operations. Thus, our query translation results in a CTL formula with double the number of propositional variables compared to the model. For example, the translation of $AF\ ?\{p, q\}$ is

$$\begin{aligned} AF \quad & ((p \wedge q \wedge x \wedge y) \vee \\ & (p \wedge \neg q \wedge x \wedge \neg y) \vee \\ & (\neg p \wedge q \wedge \neg x \wedge y) \vee \\ & (\neg p \wedge \neg q \wedge \neg x \wedge \neg y)). \end{aligned}$$

We input this formula into NuSMV, and obtain the set of minterm solutions as a propositional formula over the encoding variables $x, y, \ldots$. For $AF\ ?\{p, q\}$, on the model in Figure 4.4, we

obtain the result $x \wedge \neg y$, corresponding to the only minterm solution $p \wedge \neg q$.

4.7.5 Exactness of minterm approximation

In this section, we address the applicability of minterm-query solving to general query solving. When the minterm solutions are the strongest solutions to a query, minterm-query solving solves the general query solving problem as well, as all solutions to that query can be inferred from the minterms. In that case, we say that the minterm approximation is *exact*. We would like to identify those CTL queries that admit exact minterm approximations, independently of the model. The next proposition follows easily from the fact that any propositional formula is a disjunction of minterms.

Proposition 2 *A positive query $\varphi[?P]$ has an exact minterm approximation in any model iff $\varphi[?P]$ is distributive over disjunction, i.e., $\varphi[\alpha \vee \beta] = \varphi[\alpha] \vee \varphi[\beta]$.*

Proof:

Any propositional formula can be written as a disjunction of minterms. Thus, if a formula is a solution to a query, and the query is distributive over disjunction, the minterms in the representation of the formula are all solutions to the query. Thus the strongest solutions of such a query are minterms. □

An example of a query that admits an exact approximation is $EF\ ?$; its strongest solutions are always minterms, representing the reachable states. In [25], Chan showed that deciding whether a query is distributive over *conjunction* is EXPTIME-complete. We obtain a similar result.

Theorem 4 *Deciding whether a CTL query is distributive over disjunction is EXPTIME-complete.*

Proof:

By duality, from the result of [25]. □

Since the decision problem is hard, it would be useful to have a grammar that is guaranteed to generate queries which distribute over disjunction. Chan defined a grammar for queries distributive over conjunction, that was later corrected by Samer and Veith [88]. We can obtain a grammar for queries distributive over disjunction, from the grammar in [88], by duality.

4.7.6 Negative queries

The minterm approximation defined in Section 4.7.2 is restricted to positive queries. The general approximation framework defined above makes it easy to derive a minterm approximation for negative queries. We denote a negative query by $\varphi[\neg ?P]$. To obtain the minterm solutions to $\varphi[\neg ?P]$, we can check $\varphi[?P]$, that is, ignore the negation and treat the query as positive. For example, to check the negative query $AF\ \neg ?\{p, q\}$, we check $AF\ ?\{p, q\}$ instead. The minterm solutions to the original negative query are the duals of the *maxterm* solutions to $\varphi[?P]$. A maxterm is a *disjunction* where all the atomic propositions are, positive or negated. We denote by $\mathcal{X}(P)$ the set of maxterms over a set P of atomic propositions. For example, $\mathcal{X}(\{p, q\}) = \{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$. A minterm j is a solution to $\varphi[\neg ?P]$ iff its negation $\neg j$ is a maxterm solution to $\varphi[?P]$. We thus need to define a *maxterm approximation* $f_x : \mathcal{U}(\mathcal{F}(P)) \to 2^{\mathcal{X}(P)}$ for positive queries. We define f_x such that, for any upset B, it returns the subset of maxterms in that set, *i.e.*, $f_x(B) = B \cap \mathcal{X}(P)$. According to Definition 11, f_x is an approximation: (1) holds by f_x's definition, and (2) follows from the fact that any set of propositional formulas can be partitioned into maxterms and non-maxterms. We define the translation:

$$\mathcal{A}_x(?P) \triangleq \bigvee_{j \in \mathcal{M}(P)} (j \wedge f_x(\uparrow j)).$$

Then, by Theorem 1, model-checking $\varphi[\mathcal{A}_x(?P)]$ results in all the maxterm solutions to $\varphi[?P]$. By negating every resulting maxterm, we obtain all minterm solutions to $\varphi[\neg ?P]$. For example, maxterm solutions to $AF\ ?\{p, q\}$ for the model of Figure 4.4 is the set $\mathcal{X}(\{p, q\})$; thus, the minterm solutions to $AF\ \neg ?\{p, q\}$ are the entire set $\mathcal{M}(\{p, q\})$.

4.7.7 Case study

In this section, we study the problem of finding stable states of a model, and evaluate the performance of our implementation by comparing it to the naive approach to state-query solving.

In a study published in plant research, a model of gene interaction has been proposed to compute the "stable states" of a system of genes [44]. This work defined stable states as reachable gene configurations that no longer change, and used discrete dynamical systems to find such states. A different publication, [21], advocated the use of Kripke structures as appropriate models of biological systems, where model checking can answer some of the relevant questions about their behaviour. [21] also noted that query solving might be useful as well, but did not report any applications of this technique. Motivated by [21], we repeated the study of [44] using our state-query solving approach.

The model of [44] consists of 15 genes, each with a "level of expression" that is either boolean (0 or 1), or ternary (0,1, or 2). The laws of interaction among genes have been established experimentally and are presented as logical tables. The model was translated into a NuSMV model with 15 variables, one per gene, of which 8 are boolean and the rest are ternary, turning the laws into NuSMV next-state relations. The model has 559,872 states and is in the Appendix.

The problem of finding all stable states of the model and the initial states leading to them is formulated as the minterm-query solving of $EFAG?$, where $?$ ranges over all variables. Performance of our symbolic algorithm (Section 4.7.1) and the naive state-query solving algorithm for this query is summarized in the top row of the Table 4.2, where the times are reported in minutes. Our algorithm was implemented using NuSMV as described in Section 4.7.4. The naive algorithm was also implemented using NuSMV by generating all possible minterms over the model variables, replacing each for the placeholder in $EFAG?$ and calling NuSMV to check the resulting formulas. Both algorithms were run on a Pentium 4 processor with 2.8GHz and 1 GB of RAM. Our algorithm gave an answer in under two hours, being about 20% faster than the naive.

Table 4.2: Experimental results for query solving.

	Model	Query	Algorithms	
			TLQ	**Naive**
1	original	$EF\ AG\ ?$	117	145
2	mutant 1	$EF\ AG\ ?$	116	144
3	mutant 2	$EF\ AG\ ?$	117	145
4	mutant 3	$EF\ AG\ ?$	117	146
5	original	$AG\ ?$	116	145
6	original	$EF\ ?$	118	146
7	original	$AF\ ?$	117	145

To have a larger basis of comparison between the two algorithms, we varied the model (see rows 2-4), and the checked queries (see rows 5-7). Each "mutant" was obtained by permanently switching a different gene off, as indicated in [44]. The performance gain of our algorithm is robust to these changes.

4.7.8 Refinement

Performance improvements observed in our case study may not be attainable for every model. If the model is sufficiently small, our algorithm is likely to be faster. As models grow, however, our algorithm may suffer from BDD size explosion since it doubles the number of states variables.

To handle this problem, we envision an iterative refinement scheme. Suppose we are interested in checking a query $AF\ ?$ with two propositions, a and b. We first check $AF\ ?\{a\}$ and $AF\ ?\{b\}$. If no value is found for a proposition, then the query has no minterm solutions. Otherwise, the results correspond to the values each proposition has in all minterm solutions. For example, suppose we obtain $a = \mathsf{false}$, whereas b can be either true or false. We proceed by checking a query for each pair of propositions, using for the placeholder replacement only

those values found in the previous step. For example, we check $AF?\{a, b\}$, replacing $?$ by $(\neg a \wedge b \wedge \{\neg a \wedge b\}) \vee (\neg a \wedge \neg b \wedge \{\neg a \wedge \neg b\})$. We continue with checking triples of propositions using the valued obtained for pairs, and so on, until the query is checked on all atomic propositions, or it has been established that no answer exists.

4.7.9 Comparison with related work

While several approaches have been proposed to solve general query solving, none are effective for solving the state queries. The original algorithm of Chan [25] was very efficient (same cost as CTL model checking), but was restricted to *valid* queries, *i.e.*, queries whose solutions can be characterized by a single propositional formula. This is too restrictive for our purposes. For example, neither of the queries $EF\ ?$, $AF\ ?$, nor the stable states query $EF\ AG\ ?$ are valid. Bruns and Godefroid [17] generalized query solving to all CTL queries by proposing an automata-based CTL model checking algorithm over a lattice of sets of all possible solutions. This algorithm is exponential in the size of the state space. Gurfinkel and Chechik [60] have also provided a symbolic algorithm for general query solving. The algorithm is based on reducing query solving to multi-valued model checking and is implemented in a tool TLQ-Solver [28]. While empirically faster than the corresponding naive approach of substituting every propositional formula for the placeholder, this algorithm still has the same worst-case complexity as that in [17], and remains applicable only to modest-sized query-solving problems. An algorithm proposed by Hornus and Schnoebelen [63] finds solutions to any query, one by one, with increasing complexity: a first solution is found in time linear in the size of the state space, a second, in quadratic time, and so on. However, since the search for solutions is not controlled by their shape, finding all state solutions can still take exponential time. Other query-solving methods do not apply directly to solve our state queries, as it is exclusively concerned either with syntactic characterizations of queries[89], or with extensions, rather than restrictions, of query solving [90, 96].

There is a also a very close connection between query solving and sanity checks such

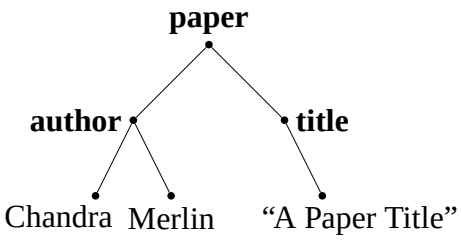


Figure 4.6: An XML example (adapted from [54]).

as vacuity and coverage [68]. All these problems require checking several “mutants” of the property or of the model to obtain the final solution. The algorithm for solving state queries presented in this paper bears many similarities to the algorithms described in [68]. Since query solving is more general, we believe it can provide a uniform framework for studying all these problems.

4.8 Conclusions and Future Work

We have identified and formalized approximate answers to vacuity detection and to query solving, which are of practical interest and can be solved more efficiently than the general versions of these problems. We have presented symbolic algorithms that compute these approximations, and described their implementations using the NuSMV model checker. We showed the efficiency of our implementations by experimental evaluation on practical cases. We have also described iterative refinement techniques that consider incrementally larger lattices to handle the size explosion of the problems.

We have presented a new application of query solving, and in particular, of our state-query solving, to finding stable states in gene networks. In the rest of this section we present another possible application open for investigation.

State query solving can be applied to querying XML documents, which are modeled as trees. A simple example, of a fragment from a document containing information about research papers and adapted from [54], is shown in Figure 4.6. An example query is “what are the titles

of all papers authored by Chandra?". Viewing tree nodes as states and edges as transitions yields a state-transition model, on which CTL properties can be evaluated [76]. Unfortunately, our example, like many other XML queries, needs to refer to both past and future, and is expressed as a CTL+Past formula as follows [54]:

$$EX^{\text{past}}\ (\text{title} \wedge EX^{\text{past}}\ (\text{paper} \wedge EX\ (\text{author} \wedge EX\ \text{Chandra}))).$$

Such formulas cannot be evaluated without modifying the internals of standard model-checkers. Formulating this question as a query yields

$$\text{paper} \wedge EX\ (\text{title} \wedge EX\ ?) \wedge EX\ (\text{author} \wedge EX\ \text{Chandra}),$$

whose desired solutions are states (here, the node labeled "A Paper Title"), and which avoids the use of the past and can be solved by our approach without modifying existing model checkers.

The main direction of investigation remains finding new interesting applications of the existing approximations or applications that require other similar approximations. We also expect to fine-tune our algorithms to fit new classes of practical problems.

Chapter 5

Assumption Generation

5.1 Introduction

This chapter presents our contributions to the automatic generation of assumptions for compositional verification in the assume-guarantee style. These contributions consist of introducing new iterative refinement techniques and demonstrating that they significantly improve upon current automated assumption generation methods. This material has been published in [49, 82, 16].

5.1.1 Interface alphabet refinement

Our first contribution is related to the way the interfaces between components are handled during assumption generation. Interfaces consist of *all* communication points through which the components may influence each other's behavior. Our assumption is that good design practice encourages system architectures with small components, and therefore the complexity of a system resides not in the individual components, but in the inter-component communication. Since interfaces determine this communication, our intuition is to manage complexity by managing the interfaces. In the learning framework of [42], the alphabet of the assumption automata being built includes *all* the actions in the component interface. A case study presented

in [81] shows, however, a smaller alphabet that is sufficient to prove a property. This smaller alphabet is determined through manual inspection and with it, assume-guarantee reasoning achieves orders of magnitude improvement over monolithic (*i.e.*, non-compositional) model checking [81]. Motivated by the successful use of a smaller alphabet in learning, we show how to automate the process of discovering a smaller alphabet that is sufficient for checking the desired properties. Smaller alphabet means smaller interface between components, which may lead to smaller assumptions, and hence to smaller verification problems.

We introduce a novel technique called *alphabet refinement* that extends the learning framework so that it starts with a small subset of the interface alphabet and adds actions into it as necessary, until a required property is shown to hold or to be violated in the system. Actions to be added are discovered by analysis of the counterexamples obtained from model checking the components. We study the properties of alphabet refinement and show experimentally that it leads to significant time and memory savings as compared to the original learning framework [42] and achieves better scalability than monolithic model checking.

We have implemented our algorithm within the LTSA model checking tool [72], but the algorithm is applicable to and may benefit any of the previous learning-based approaches [2, 79, 91], and it may also benefit other compositional analysis techniques. Compositional Reachability Analysis (CRA), for example, computes abstractions of component behaviors based on their interfaces. In the context of property checking [30], smaller interfaces may result in more compact abstractions, leading to smaller state spaces when components are put together.

5.1.2 Abstraction refinement

Our second contribution provides an alternative to the learning-based assumption generation techniques. Recall that the simplest assume-guarantee rule checks if a system composed of components M_1 and M_2 satisfies a property φ by checking that M_1 under assumption A satisfies φ (*Premise 1*) and discharging A on the environment M_2 (*Premise 2*). Learning-based

approaches using L* [42, 2, 22] work by iteratively making conjectures in the form of automata that represent intermediate assumptions. Each conjectured assumption A is used to check the two premises of the rule. The process ends if A passes both premises of the rule, in which case the property holds in the system, or if we uncover a real violation. Otherwise, a counterexample is returned, and L* modifies the conjecture. The work in [56] uses sampling rather than L* to learn the assumptions in a similar way.

We propose an alternative called AGAR (Assume-Guarantee Abstraction Refinement), that replaces the iterative assumption refinement using learning with iterative abstraction refinement. It follows from the observation that for universal properties, *Premise 2* amounts to checking that A is a conservative abstraction of M_2, *i.e.*, an abstraction that preserves all of M_2's execution paths. The algorithm works by iteratively computing assumptions as conservative abstractions of the interface behavior of M_2, *i.e.*, the behavior that only concerns the interaction with M_1. In each iteration, the computed assumption A satisfies *Premise 2* of the assume-guarantee rule by construction, and is only checked for *Premise 1*. If the check is successful, we conclude that $M_1 \parallel M_2$ satisfies the property; if the check fails, we get a counterexample trace that we analyze to see if it corresponds to a real error in $M_1 \parallel M_2$ or it is spurious due to the over-approximation in the abstraction. If it is spurious, we used it to refine A and then repeat the entire process.

Unlike learning-based assumption generation, AGAR does not constrain assumptions to be *deterministic*. It is well-known that a deterministic automaton can be, in the worst case, exponentially larger than a non-deterministic one accepting the same language. Therefore, the assumptions constructed with AGAR in the worst case can be exponentially smaller than those obtained with learning, resulting in smaller verification problems. To reduce the assumption sizes even further, we also combine the abstraction refinement with our previous *interface alphabet refinement*, which extends AGAR so that initially it constructs the abstraction A with a small subset of the interface alphabet and adds actions to the alphabet as necessary until the required property is shown to hold or to be violated in the system. Actions to be added are

discovered also by counterexample analysis.

We have implemented AGAR with alphabet refinement in the explicit state model checker LTSA [72] and performed a series of experiments which demonstrate that it can achieve better performance than L*.

5.1.3 Outline

We introduce some necessary background about LTSA and learning-based assume-guarantee reasoning in Section 5.2. We present our alphabet refinement algorithm, its properties, and evaluation, in Section 5.3. Section 5.4 presents our algorithm for abstraction refinement in assumption generation, and its evaluation. We conclude the chapter and give some pointers to future work in Section 5.5.

5.2 Background

5.2.1 Labeled Transition Systems (LTSs) Analysis (LTSA)

LTSA is an explicit-state model checker that analyzes finite-state systems modeled as labeled transition systems (LTSs). Let $\mathcal{A}$ be the universal set of observable actions and let τ denote the unobservable action. Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$, be an LTS, where: Q is the set of states; $\alpha M \subseteq \mathcal{A}$ is the set of observable actions called the *alphabet* of M; $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is the transition relation, and q_0 is the initial state. The LTS M is *non-deterministic* if it contains τ-transitions or if $\exists (q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*. We use π to denote a special *error state* that has no outgoing transitions, and Π to denote the LTS $\langle \{\pi\}, \mathcal{A}, \emptyset, \pi \rangle$. For the parallel composition of LTSs M_1 and M_2, if any of them is Π, then the composition $M = M_1 \parallel M_2$ is also Π.

Paths and traces

Recall that a *path* in an LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ is a sequence p of alternating states and (observable or unobservable actions) of M, $p = q_{i_0}, a_0, q_{i_1}, a_1, \ldots, a_{n-1}, q_{i_n}$ such that for every $k \in \{0, \ldots, n-1\}$ we have $(q_{i_k}, a_k, q_{i_{k+1}}) \in \delta$.

The *trace of path* p, denoted $\sigma(p)$ is the sequence $b_0, b_1, \ldots, b_l$ of actions along p, obtained by removing all τ from $a_0, \ldots, a_{n-1}$. A state q *reaches* a state q' in M with a sequence of actions t, denoted $q \stackrel{t}{\Rightarrow} q'$, if there exists a path p from q to q' in M whose trace is t, *i.e.*, $\sigma(p) = t$. A *trace of* M is the trace of a path in M starting from q_0. The set of all traces of M forms the *language* of M, denoted $\mathcal{L}(M)$. For any trace $t = a_0, a_1, \ldots, a_{n-1}$, a *trace LTS* can be constructed whose only transitions are $q_0 \stackrel{a_0}{\rightarrow} q_1 \stackrel{a_1}{\rightarrow} q_2 \ldots \stackrel{a_{n-1}}{\rightarrow} q_n$. We sometimes abuse the notation and denote by t both a trace and its trace LTS. The meaning should be clear from the context. For $\Sigma' \subseteq \Sigma$, $t{\downarrow_{\Sigma'}}$ is the trace obtained by removing from t all actions $a \notin \Sigma$. Similarly, $M{\downarrow_{\Sigma'}}$ is an LTS over Σ obtained from M by renaming to τ all the action labels not in Σ. Let t_1, t_2 be two traces. Let Σ_1, Σ_2 be the sets of actions occurring in t_1, t_2, respectively. By the *symmetric difference* of t_1 and t_2 we mean the symmetric difference of sets Σ_1 and Σ_2.

Safety properties

We call a deterministic LTS not containing π a *safety LTS*. A safety property φ is specified as a *safety LTS* whose language $\mathcal{L}(\varphi)$ defines the set of acceptable behaviors over $\alpha\varphi$.

An LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ satisfies $\varphi = \langle Q_\varphi, \Sigma_\varphi, \delta_\varphi, q_0^\varphi \rangle$, denoted $M \models \varphi$, iff $\forall t \in \mathcal{L}(M) \cdot t{\downarrow_{\Sigma_\varphi}} \in \mathcal{L}(\varphi)$. For checking a property φ, its safety LTS is *completed* by adding error state π and transitions on all the missing outgoing actions from all states into π so that the resulting transition relation is (left-)total (when seen as in $(Q \times (\Sigma \cup \{\tau\})) \times Q$) and deterministic; the resulting LTS is denoted by φ_{err}. LTSA checks $M \models \varphi$ by computing $M \parallel \varphi_{err}$ and checking if π is reachable in the resulting LTS.

As an example (from [42], consider a simple communication channel that consists of two components whose LTSs are shown in Fig. 5.1(a). Note that the initial state of all LTSs in this

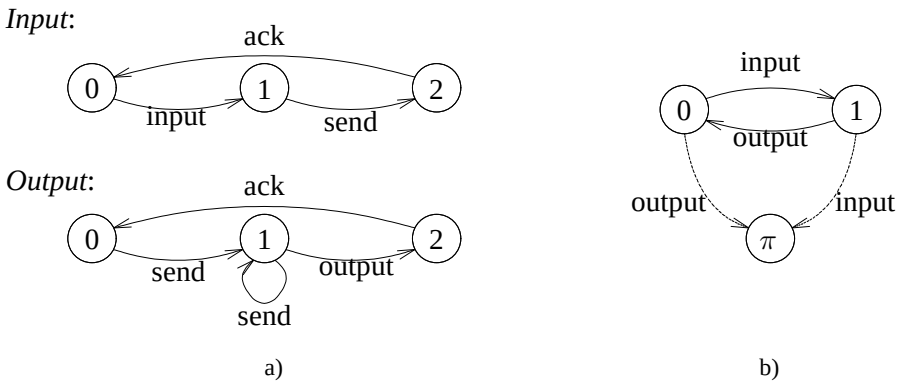


Figure 5.1: (a) Example LTSs; (b) *Order* property.

work is state 0. The *Input* LTS receives an input on action *input*, then sends it to the *Output* LTS on action *send* and then receives an acknowledgement on action *ack*. After being sent some data on action *send*, *Output* produces some output on action *output* and acknowledges that it has finished on action *ack*. At this point, both LTSs return to their initial states so the process can be repeated. For an example of a safety property, the *Order* LTS in Fig. 5.1(b) captures a desired behavior of the communication channel from Fig. 5.1(a). The property comprises states 0 and 1, and the transitions denoted by solid arrows. It expresses the fact that inputs and outputs come in matched pairs, with the input always preceding the output. The dashed arrows represent transitions to the error state that were added to obtain $Order_{err}$.

5.2.2 Assume-guarantee rules

Recall that in the assume-guarantee paradigm a formula is a triple $\langle A \rangle M \langle \varphi \rangle$, where M is a component, φ is a property, and A is an assumption about M's environment. The formula is true if whenever M is part of a system satisfying A, then the system must also guarantee φ [55]. In LTSA, checking $\langle A \rangle M \langle \varphi \rangle$ reduces to checking $A \parallel M \models \varphi$ [72]. We work with a number of symmetric and asymmetric rules for assume-guarantee reasoning.

Recall the simple rule from Chapter 2:

Rule ASYM

$$\frac{\begin{array}{ll} 1: & \langle A \rangle\ M_1\ \langle \varphi \rangle \\ 2: & \langle \mathsf{true} \rangle\ M_2\ \langle A \rangle \end{array}}{\langle \mathsf{true} \rangle\ M_1 \parallel M_2\ \langle \varphi \rangle}$$

Soundness of the rule follows from the fact that $\langle \mathsf{true} \rangle\ M_2\ \langle A \rangle$ implies $\langle \mathsf{true} \rangle\ M_1 \parallel M_2\ \langle A \rangle$ and from the definition of assume-guarantee triples. Completeness holds trivially, by substituting M_2 for A. Note that the rule is not symmetric in its use of the two components, and does not support circularity. Despite the simplicity of the rule, automating the discovery of the assumption A even for this rule has been a long-standing challenge until recently.

Another rule is similar to ASYM but involves some form of circular reasoning. It appears originally in [55] for reasoning about two components. We extend it to reasoning about $n \geq 2$ components.

Rule CIRC-N

$$\frac{\begin{array}{ll} 1: & \langle A_1 \rangle\ M_1\ \langle \varphi \rangle \\ 2: & \langle A_2 \rangle\ M_2\ \langle A_1 \rangle \\ \vdots & \\ n: & \langle A_n \rangle\ M_n\ \langle A_{n-1} \rangle \\ n+1: & \langle \mathsf{true} \rangle\ M_1\ \langle A_n \rangle \end{array}}{\langle \mathsf{true} \rangle\ M_1 \parallel \cdots \parallel M_n\ \langle \varphi \rangle}$$

Soundness and completeness of this rule follow from [55]. Note that this rule is similar to the rule ASYM applied recursively for $n+1$ components, where the first and the last component coincide (hence the term "circular").

Although sound and complete, the rules presented so far are not always satisfactory since they are not symmetric in the use of the components. The work in [9] proposes a set of symmetric rules that are sound and complete. They are symmetric in the sense that they establish and discharge assumptions for each component at the same time.

In the next rule, the co-assumption coA_i for M_i is the complement of A_i, *i.e.*, an LTS whose language is the complement of the language of A_i.

Rule SYM-N

$$\frac{\begin{array}{ll} 1: & \langle A_1 \rangle\ M_1\ \langle \varphi \rangle \\ \vdots & \\ n: & \langle A_n \rangle\ M_n\ \langle \varphi \rangle \\ n+1: & \mathcal{L}\left(coA_1 \parallel \cdots \parallel coA_n\right) \subseteq \mathcal{L}\left(\varphi\right) \end{array}}{\langle \mathsf{true} \rangle\ M_1 \parallel \cdots \parallel M_n\ \langle \varphi \rangle}$$

We require $\alpha\varphi \subseteq \alpha M_1 \cup \cdots \cup \alpha M_n$ and for $i \in \{1, \ldots, n\}$, $\alpha A_i \subseteq (\alpha M_1 \cap \cdots \cap \alpha M_n) \cup \alpha\varphi$. Informally, each A_i is a postulated environment assumption for the component M_i to satisfy property φ.

5.2.3 The L* learning algorithm

L* was developed by Angluin [4] and later improved by Rivest and Shapire [87]. It learns an unknown regular language U over alphabet Σ and produces a deterministic finite state automaton (DFA) that accepts it. L* interacts with a *Minimally Adequate Teacher* that answers two types of questions from L*. The first type is a *membership query* asking whether a string $s \in \Sigma^*$ is in U. For the second type, the learning algorithm generates a *conjecture* A and asks whether $\mathcal{L}(A) = U$. If $\mathcal{L}(A) \neq U$ the Teacher returns a counterexample, which is a string s in the symmetric difference of $\mathcal{L}(A)$ and U. L* is guaranteed to terminate with a minimal automaton A for U. If A has n states, L* makes at most $n-1$ incorrect conjectures. The number of membership queries made by L* is $O(kn^2 + n \log m)$, where k is the size of Σ, n is the number of states in the minimal DFA for U, and m is the length of the longest counterexample returned when a conjecture is made.

5.2.4 Interface alphabet and weakest assumption

In the framework of [42], an important notion is that of the *weakest assumption* [53] depending on the interface between the component and its environment.

Definition 13 (Weakest Assumption for Σ) *Let M_1 be an LTS for a component, φ be a safety LTS for a property required of M_1, and Σ be the interface of the component to the environment. The weakest assumption $A_{w,\Sigma}$ of M_1 for Σ and for property φ is a deterministic LTS such that: 1) $\alpha A_{w,\Sigma} = \Sigma$, and 2) for any component M_2, $M_1 \parallel (M_2{\downarrow_\Sigma}) \models \varphi$ iff $M_2 \models A_{w,\Sigma}$*

Projection of M_2 to Σ forces M_2 to communicate with our module only through Σ (second condition above). [53] showed that the weakest assumptions exist for components expressed as LTSs and safety properties and provided an algorithm for computing these assumptions.

The definition above refers to *any* environment component M_2 that interacts with component M_1 via an alphabet Σ. When M_2 is given, there is a natural notion of the complete *interface* between M_1 and its environment M_2, when property φ is checked.

Definition 14 (Interface Alphabet) *Let M_1 and M_2 be component LTSs, and φ be a safety LTS. The interface alphabet Σ_I of M_1 (with respect to M_2 and φ) is defined as: $\Sigma_I = (\alpha M_1 \cup \alpha\varphi) \cap \alpha M_2$.*

Definition 15 (Weakest Assumption) *Given M_1, M_2 and φ as above, the weakest assumption A_w is defined as A_{w,Σ_I}.*

Note that, to deal with any system-level property, properties in definition 14 are allowed to include actions that are not in αM_1 but are in αM_2. These actions need to be in the interface since they are controllable by M_2. Moreover, from the above definitions, it follows that the assumption A_w is indeed the *weakest*: it characterizes all the environments M_2 that, together with M_1, satisfy property φ, *i.e.*, $M_1 \parallel M_2 \models \varphi$ iff $M_2 \models A_w$.

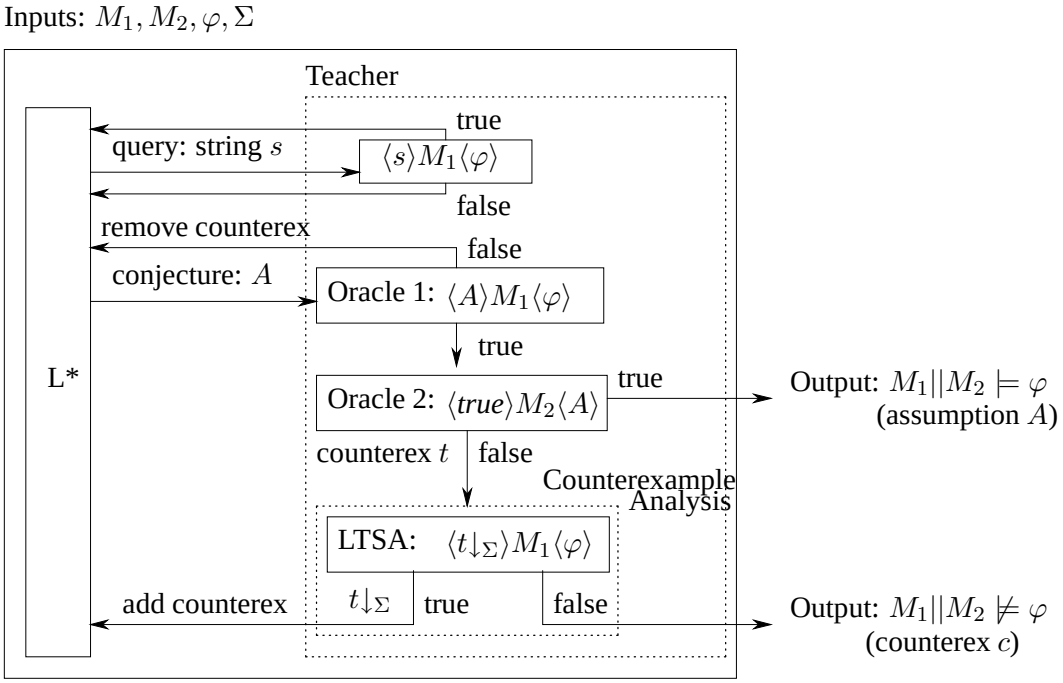


Figure 5.2: Learning framework (from [81]).

5.2.5 Learning framework

With Rule ASYM

The original learning framework from [42] was given for Rule ASYM and is illustrated in Figure 5.2. The framework checks $M_1 \parallel M_2 \models \varphi$ by checking the two premises of the assume-guarantee rule separately, and using the conjectures A from L* as assumptions. The automaton A output by L* is, in the worst case, the *weakest assumption* A_w. The alphabet given to the learner is fixed to $\Sigma = \Sigma_I$.

The Teacher is implemented using model checking. For membership queries on string s, the Teacher uses LTSA to check $\langle s \rangle M_1 \langle \varphi \rangle$. If true, then $s \in \mathcal{L}(A_w)$, so the Teacher returns true. Otherwise, the answer to the query is false. The conjectures returned by L* are intermediate assumptions A. The Teacher implements two *oracles*: *Oracle 1* guides L* towards a conjecture that makes $\langle A \rangle M_1 \langle \varphi \rangle$ true. Once this is accomplished, *Oracle 2* is invoked to discharge A on M_2. If this is true, then the assume guarantee rule guarantees that φ holds on $M_1 \parallel M_2$. The

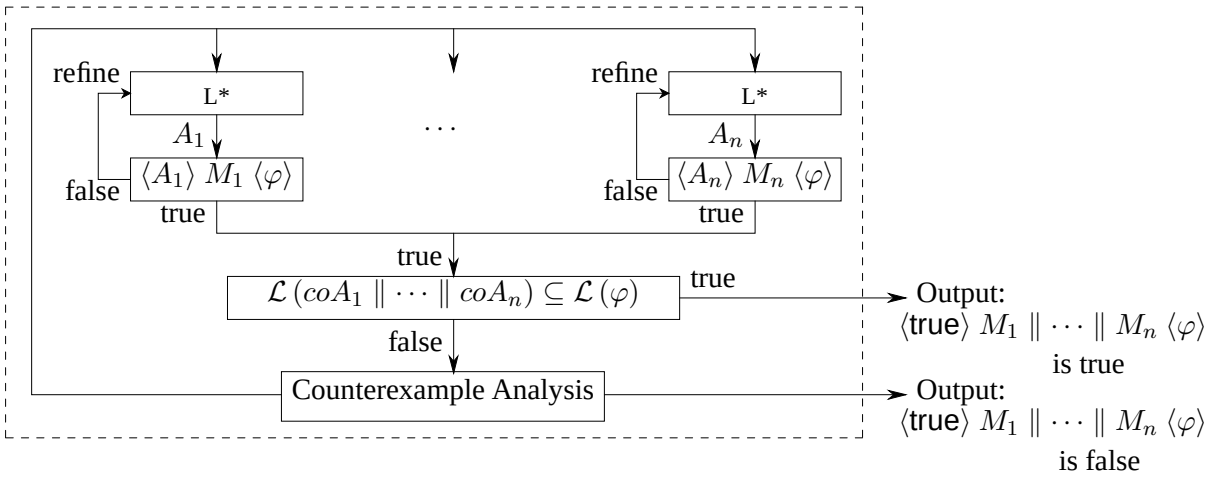


Figure 5.3: Learning framework for rule SYM-N (from [9]).

Teacher then returns true and the computed assumption A. Note that A is not necessarily A_w, it can be *stronger* than A_w, *i.e.*, $\mathcal{L}(A) \subseteq \mathcal{L}(A_w)$, but the computed assumption is good enough to prove that the property holds or is violated. If model checking returns a counterexample, further analysis is needed to determine if φ is indeed violated in $M_1 \parallel M_2$ or if A is imprecise due to learning, in which case A needs to be modified.

Counterexample analysis. Trace t is the counterexample from Oracle 2 obtained by model checking $\langle \mathit{true} \rangle M_2 \langle A \rangle$. To determine if t is a real counterexample, *i.e.*, if it leads to error on $M_1 \parallel M_2 \models \varphi$, the Teacher analyzes t on $M_1 \parallel \varphi_{err}$. In doing so, the Teacher needs to first project t onto the assumption alphabet Σ, that is the interface of M_2 to $M_1 \parallel \varphi_{err}$. Then the Teacher uses LTSA to check $\langle t{\downarrow_\Sigma} \rangle M_1 \langle \varphi \rangle$. If the error state is not reached during the model checking, t is not a real counterexample, and $t{\downarrow_\Sigma}$ is returned to the learner L* to modify its conjecture. If the error state is reached, the model checker returns a counterexample c that witnesses the violation of φ on M_1 in the context of $t{\downarrow_\Sigma}$. With the assumption alphabet $\Sigma = \Sigma_I$, c is guaranteed to be a real error trace on $M_1 \parallel M_2 \parallel \varphi_{err}$ [42]. However, as we shall see in the next section, if $\Sigma \subset \Sigma_I$, c is not necessarily a real counterexample and further analysis is needed.

With Rule SYM-N

The framework has been extended to other rules in [9]; for Rule SYM-N, it is illustrated in Fig. 5.3. To obtain appropriate assumptions, the framework applies the compositional rule in an iterative fashion. At each iteration L* is used to generate appropriate assumptions for each component, based on querying the system and on the results of the previous iteration. Each assumption is then checked to establish the premises of Rule SYM-N. We use separate instances of L* to iteratively learn $A_{w1}, \ldots A_{wn}$.

As before, the Teacher needed by L* is implemented with calls to the model checker. The conjectures returned by L* are the intermediate assumptions $A_1, \ldots A_n$. The Teacher implements $n + 1$ oracles, one for each premise in the SYM-N rule:

- *Oracles* $1, \ldots n$ guide the corresponding L* instances towards conjectures that make the corresponding premise of rule SYM-N true. Once this is accomplished,

- *Oracle* $n + 1$ is invoked to check the last premise of the rule, *i.e.*,

$$\mathcal{L}\left(coA_1 \parallel \cdots \parallel coA_n\right) \subseteq \mathcal{L}\left(\varphi\right)$$

  If this is true, rule SYM-N guarantees that $M_1 \parallel \cdots \parallel M_n$ satisfies φ.

If the result of *Oracle* $n + 1$ is false (with counterexample trace t), by counterexample analysis we identify either that φ is indeed violated in $M_1 \parallel \cdots \parallel M_n$ or that some of the candidate assumptions need to be modified. If (some of the) assumptions need to be refined in the next iteration, then behaviors must be added to those assumptions. The result will be that at least the behavior that the counterexample represents will be allowed by those assumptions during the next iteration. The new assumptions may of course be too abstract, and therefore the entire process must be repeated.

Counterexample analysis. Counterexample t is analyzed in a way similar to the analysis for Rule ASYM, *i.e.*, we analyze t to determine whether it indeed corresponds to a violation in

$M_1 \parallel \cdots \parallel M_n$. This is checked by simulating t on $M_i \parallel co\varphi$, for all $i = 1 \ldots n$. The following cases arise:

- If t is a violating trace of all components $M_1, \ldots M_n$, then $M_1 \parallel \cdots \parallel M_n$ indeed violates φ, which is reported to the user.

- If t is not a violating trace of at least one component M_i, then we use t to weaken the corresponding assumption(s).

5.2.6 Experimental data

Models

In our experiments, we use the following case studies (all these models were analyzed before, using the original assume guarantee framework, without refinement):

- *Gas Station* [41] describes a self-serve gas station consisting of k customers, two pumps, and an operator, for $k = 3, 4, 5$.

- *Chiron* [41] models a graphical user interface consisting of k "artists", a wrapper, a manager, a client initialization module, a dispatcher, and two event dispatchers, for $k = 2...5$.

- *MER* [81] models flight software component for JPL's Mars Exploration Rovers. It contains k users competing for resources that are managed by a resource arbiter, for $k = 2..6$.

- *Rover Executive* [42] is a model of a subsystem for the Ames K9 Rover. The model is comprised of a main component 'Executive' and an 'ExecCondChecker' component that is responsible for monitoring state conditions.

Decompositions

We use the decompositions of these models into components as given with their original descriptions, that we call n*-way decompositions,* but also their decompositions into two supercomponents obtained by grouping the given modules. We call the latter*2-way decompositions*, and for Gas Station and Chiron they are the decompositions found in [41] to be the best for the performance of the learning framework. For Gas Station, the decomposition is: the operator and the first pump in one component, and the rest of the modules in the other. For Chiron, the event dispatchers are one component, and the rest of the modules are the other. For MER we use the decomposition where half of the users are in one component, and the other half with the arbiter in the other. For the Rover we use the two components as described in [42].

Properties

In [41], 4 properties for Gas Station and 9 properties for Chiron were checked, to study how various 2-way model decompositions affect the performance of learning (without alphabet refinement). For most of these properties, the learning approach performs better than non-compositional verification and it produces small (one-state) assumptions. For some other properties, learning does not perform that well, and produces much larger assumptions. To stress-test our approaches, we selected the latter, more challenging, properties for our study here:

- For Gas Station, we check the property that the operator correctly gives change to a customer for the pump that he/she used.
- For Chiron, we check Property 1, stating that the dispatcher notifies artists of an event before receiving a next event, and Property 2, stating that the dispatcher only notifies artists of an event after it receives that event.
- For MER, we check a mutual exclusion property stating that communication and driving cannot happen at the same time as they share common resources.

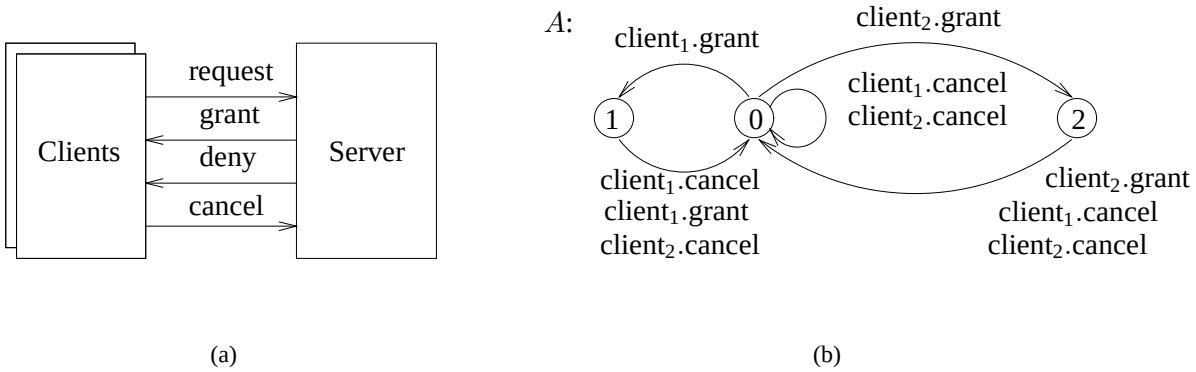


Figure 5.4: Client-Server example: (a) complete interface and (b) derived assumption with a subset of the interface alphabet.

- For Rover, the property we check states that for a specific shared variable, if the Executive reads its value, then the ExecCondChecker should not read it before the Executive clears it first.

Also note that for Gas station and Chiron we used the same configurations (values for k) as reported in [41].

5.3 Alphabet Refinement

In this section we present our algorithm for interface alphabet refinement, motivated first by an example. We also present some theoretical properties of the algorithm and then an experimental evaluation. We conclude after further comparison with related work.

5.3.1 Motivating example

To illustrate the benefits of smaller interface alphabets for assume guarantee reasoning, consider a simple client-server application (from [81]). It consists of a *server* component and two identical *client* components that communicate through shared actions. Each client sends *requests* for reservations to use a common resource, waits for the server to *grant* the reservation,

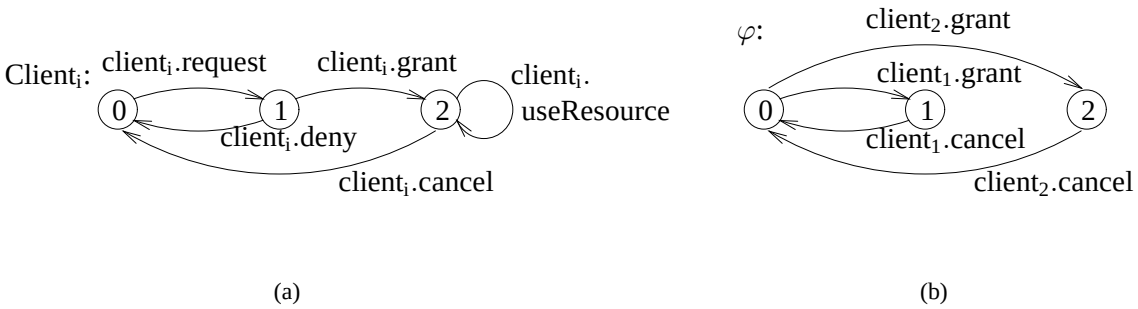


Figure 5.5: Example LTS for (a) a client and (b) a mutual exclusion property (b)).

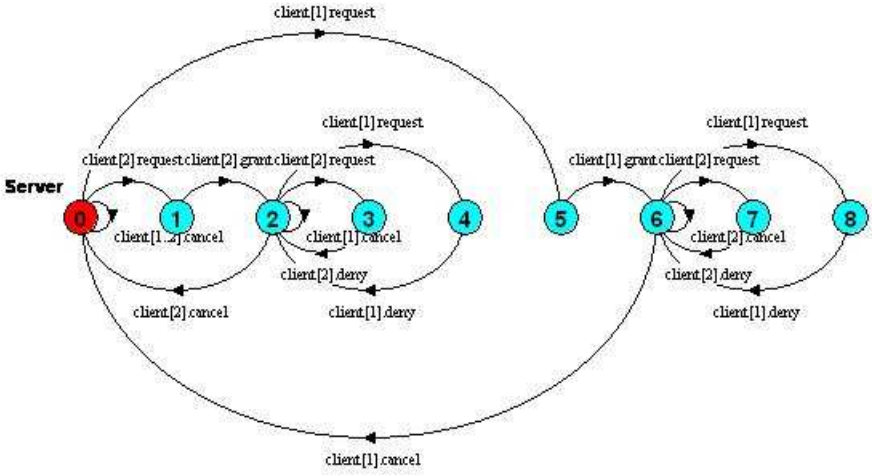


Figure 5.6: Client-Server example: LTS for Server (as displayed by the LTSA tool).

uses the resource, and then *cancels* the reservation. For example, the LTS of a client is shown in Figure 5.5(a), where $i = 1, 2$.

The server can *grant* or *deny* a request, ensuring that the resource is used only by one client at a time. The LTS of the server is in Figure 5.6.

The mutual exclusion property in Figure 5.5(b) captures the desired behaviour of the client-server application discussed earlier.. To check the property in a compositional way, assume that

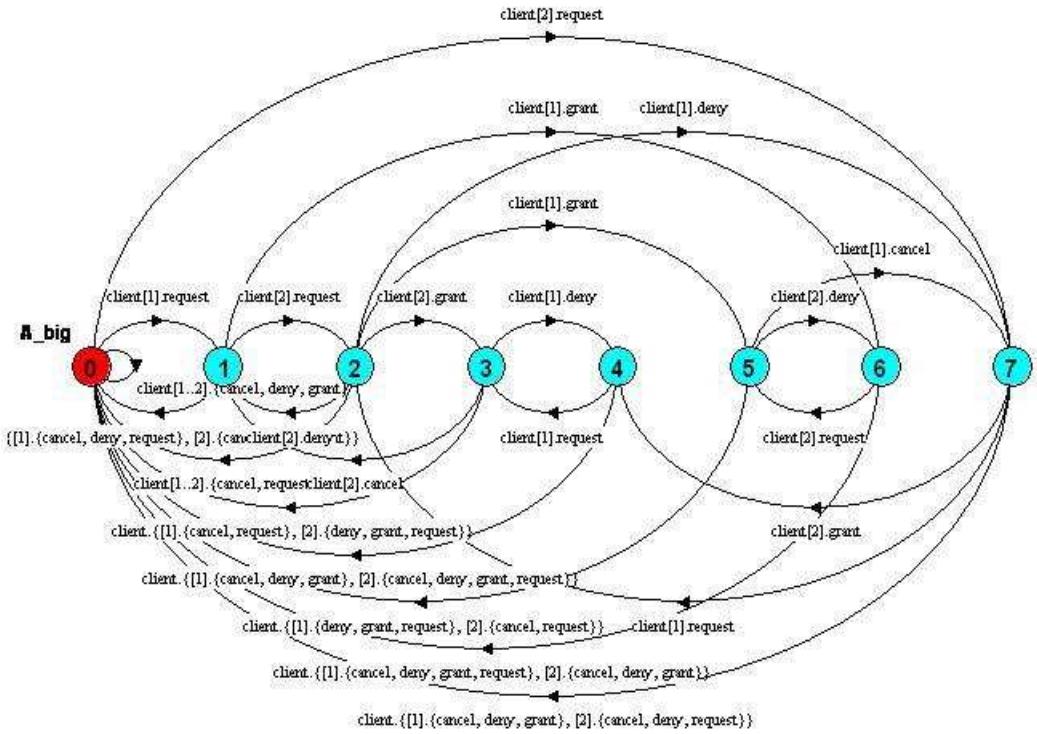


Figure 5.7: Client-Server example: assumption obtained with the complete interface alphabet (as displayed by the LTSA tool).

we break up the system into: $M_1 = \text{Client}_1 \parallel \text{Client}_2$ and $M_2 = \text{Server}$. The *complete* alphabet of the interface between $M_1 \parallel \varphi$ and M_2 (see Figure 5.4(a)) is: $\{\text{client}_1.\text{cancel}, \text{client}_1.\text{grant}, \text{client}_1.\text{deny}, \text{client}_1.\text{request}, \text{client}_2.\text{cancel}, \text{client}_2.\text{grant}, \text{client}_2.\text{deny}, \text{client}_2.\text{request}\}$.

Using this alphabet and the learning method of [42] yields an assumption with 8 states, as shown in Figure 5.7. However, a (much) smaller assumption is sufficient for proving the mutual exclusion property (see Figure 5.4(b)). The assumption alphabet is $\{\text{client}_1.\text{cancel}, \text{client}_1.\text{grant}, \text{client}_2.\text{cancel}, \text{client}_2.\text{grant}\}$, which is a strict subset of the complete interface alphabet (and is, in fact, the alphabet of the property). This assumption has just 3 states, and enables more efficient verification than the 8-state assumption obtained with the complete alphabet. In the following sections, we present techniques to infer smaller interface alphabets (and the corresponding assumptions) automatically.

5.3.2 Algorithm

Let M_1 and M_2 be components, φ be a property, Σ_I be the interface alphabet, and Σ be an alphabet such that $\Sigma \subset \Sigma_I$. Assume that we use the learning framework of the previous section with Rule ASYM, but we now set this smaller Σ to be the alphabet of the assumption that the framework learns. From the soundness of Rule ASYM, if the framework reports true, $M_1 \parallel M_2 \models \varphi$. When it reports false, it is because it finds a trace t in M_2 that falsifies $\langle t{\downarrow_\Sigma}\rangle M_1\langle\varphi\rangle$. This, however, does not necessarily mean that $M_1 \parallel M_2 \not\models \varphi$. Real violations are discovered by our original framework only when the alphabet is Σ_I, and are traces t' of M_2 that falsify $\langle t'{\downarrow_{\Sigma_I}}\rangle M_1\langle\varphi\rangle$[1].

We illustrate this with the client-server example. Assume $\Sigma = \{\text{client}_1\text{.cancel}, \text{client}_1\text{.grant}, \text{client}_2\text{.grant}\}$, smaller than $\Sigma_I = \{\text{client}_1\text{.cancel}, \text{client}_1\text{.grant}, \text{client}_1\text{.deny}, \text{client}_1\text{.request}, \text{client}_2\text{.cancel}, \text{client}_2\text{.grant}, \text{client}_2\text{.deny}, \text{client}_2\text{.request}\}$. Learning with Σ produces trace: $t = \langle\text{client}_2\text{.request}, \text{client}_2\text{.grant}, \text{client}_2\text{.cancel}, \text{client}_1\text{.request}, \text{client}_1\text{.grant}\rangle$. Projected to Σ, this becomes $t{\downarrow_\Sigma} = \langle\text{client}_2\text{.grant}, \text{client}_1\text{.grant}\rangle$. In the context of $t{\downarrow_\Sigma}$, $M_1 =$ Clients violates the property since $\text{Client}_1 \parallel \text{Client}_2 \parallel \varphi_{err}$ contains the following behavior (see Figure 5.4):

$$(0,0,0) \xrightarrow{\text{client}_1\text{.request}} (1,0,0) \xrightarrow{\text{client}_2\text{.request}} (1,1,0) \xrightarrow{\text{client}_2\text{.grant}} (1,2,2) \xrightarrow{\text{client}_1\text{.grant}} (2,2,\textit{error}).$$

Learning therefore reports *false*. This behavior is not feasible, however, in the context of $t{\downarrow_{\Sigma_I}} = \langle\text{client}_2\text{.request}, \text{client}_2\text{.grant}, \text{client}_2\text{.cancel}, \text{client}_1\text{.request}, \text{client}_1\text{.grant}\rangle$. This trace requires a client_2.cancel to occur before the client_1.grant. Thus, in the context of Σ_I the above violating behavior would be infeasible. We conclude that when applying the learning framework with alphabets smaller that Σ_I, if *true* is reported then the property holds in the system, but violations reported may be spurious.

We propose a technique called *alphabet refinement*, which extends the learning framework to deal with smaller alphabets than Σ_I while avoiding spurious counterexamples. The steps of the algorithm are as follows (see Figure 5.8 (a)):

[1] In the assume guarantee triples: $t{\downarrow_\Sigma}$, $t'{\downarrow_{\Sigma_I}}$ are trace LTSs with alphabets Σ, Σ_I respectively.

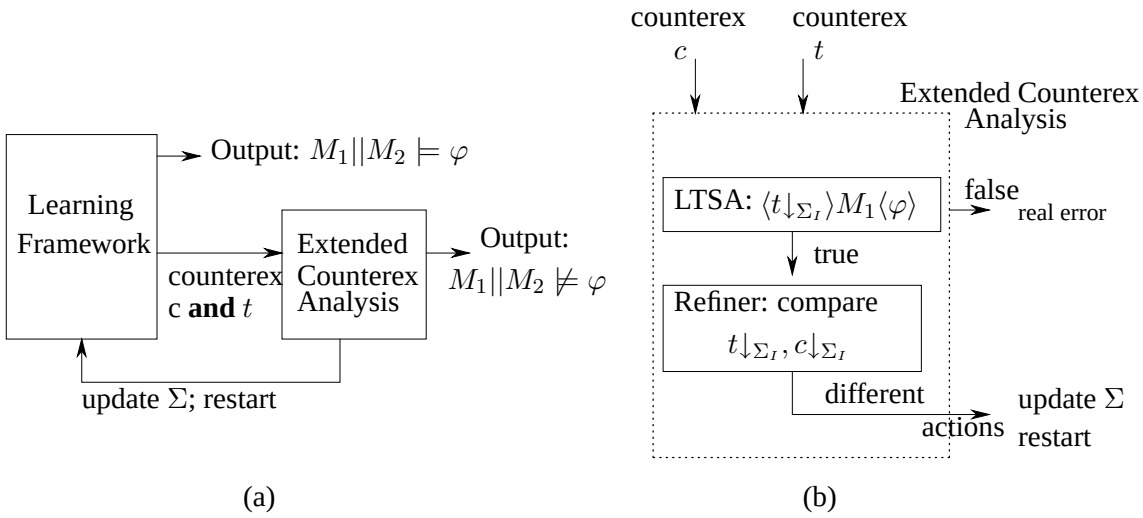


Figure 5.8: (a) Learning with alphabet refinement and (b) additional counterexample analysis.

1. **Initialize** Σ to a set S such that $S \subseteq \Sigma_I$.

2. Use the classic learning framework for Σ. If the framework returns *true,* then report *true* and go to step 4 (END). If the framework returns false with counterexamples c (and t), go to the next step.

3. Perform **extended counterexample analysis** for c. If c is a real counterexample, then report *false* and go to step 4 (END). If c is spurious, then **refine** Σ, which consists of adding to Σ actions from Σ_I. Go to step 2.

4. END of algorithm.

When spurious counterexamples are detected, the refiner extends the alphabet with actions in the alphabet of the weakest assumption and the learning of assumptions is restarted. In the worst case, Σ_I is reached, and as proved in our previous work, learning then only reports real counterexamples. In the above high-level algorithm, the highlighted steps 1) alphabet initialization, 2) extended counterexample analysis and 3) alphabet refinement are further specified in the following.

Alphabet initialization. The correctness of our algorithm is insensitive to the initial alphabet.

We implement two options: 1) we set the initial alphabet to the empty set to allow the algorithm to only take into account actions that it discovers, and 2) we set the initial alphabet to those actions in the alphabet of the property that are also in Σ_I,*i.e.*, $\alpha\varphi \cap \Sigma_I$ (in the experiments we use the latter). The intuition for the latter option is that these interface actions are likely to be significant in proving the property, since they are involved in its definition. A good initial guess of the alphabet may achieve big savings in terms of time since it results in fewer refinement iterations.

Extended counterexample analysis. An additional counterexample analysis is appended to our original learning framework as illustrated in Figure 5.8(a). The steps of this analysis are shown in Figure 5.8(b). The extension takes as inputs both the counterexample t returned by Oracle 2, and the counterexample c that is returned by the original counterexample analysis. We modified our "classic" learning framework (Figure 5.2) to return both c **and** t to be used in alphabet refinement (as explained below). As discussed, c is obtained because $\langle t{\downarrow_\Sigma}\rangle M_1\langle\varphi\rangle$ does not hold. The next step is to check whether in fact t uncovers a real violation in the system. As illustrated by our client-server example, the results of checking $M_1 \parallel \varphi_{err}$ in the context of t projected to different alphabets may be different. The correct results are obtained by projecting t on the alphabet Σ_I of the weakest assumption. Counterexample analysis therefore calls LTSA to check $\langle t{\downarrow_{\Sigma_I}}\rangle M_1\langle\varphi\rangle$. If LTSA finds an error, the resulting counterexample c is a real counterexample. If error is not reached, the alphabet Σ needs to be refined. Refinement proceeds as described next.

Alphabet refinement. When spurious counterexamples are detected, we need to enrich the current alphabet Σ so that these counterexamples are eventually eliminated. A counterexample c is spurious if in the context of $t{\downarrow_{\Sigma_I}}$ it would not be obtained. Our refinement heuristics are therefore based on comparing c and $t{\downarrow_{\Sigma_I}}$ to discover actions in Σ_I to be added to the learning alphabet (for this reason c is also projected on Σ_I in the refinement process). We have currently implemented the following heuristics:

AllDiff: adds all the actions in the symmetric difference of $t{\downarrow_{\Sigma_I}}$ and $c{\downarrow_{\Sigma_I}}$; a potential problem is that is that it may add too many actions too soon, but if it happens to add useful actions, it may terminate after fewer iterations;

Forward: scans the traces in parallel from beginning to end looking for the first index i where they disagree; if such an i is found, both actions $t{\downarrow_{\Sigma_I}}\ (i), c{\downarrow_{\Sigma_I}}\ (i)$ are added to the alphabet.

Backward: same as Forward but scans from the end of the traces to the beginning.

So far, we have discussed our algorithm for two components. We have extended alphabet refinement to n modules $M_1, \ldots M_n$, for any $n \geq 2$. Previous work extended learning (without refinement) to n components [42, 81]. To check if $M_1 \parallel \ldots \parallel M_n$ satisfies φ, we decompose it into: M_1 and $M_2' = M_2 \parallel ... \parallel M_n$ and the learning algorithm (without refinement) is invoked recursively for checking the second premise of the assume-guarantee rule.

Learning with alphabet refinement uses recursion in a similar way. At each recursive invocation for M_j, we solve the following problem: find assumption A_j and alphabet Σ_{A_j} such that the rule premises hold:

Oracle 1: $M_j \parallel A_j \models A_{j-1}$

Oracle 2: $M_{j+1} \parallel ... \parallel M_n \models A_j$

Here A_{j-1} is the assumption for M_{j-1} and plays the role of the property for the current recursive call. Thus, the alphabet of the weakest assumption for this recursive invocation is $\Sigma_I^j = (\alpha M_j \cup \alpha A_{j-1}) \cap (\alpha M_{j+1} \cup \ldots \cup \alpha M_n)$. If Oracle 2 returns a counterexample, then the counterexample analysis and alphabet refinement proceed exactly as in the 2 component case. At a new recursive call for M_i with a new A_{i-1}, the alphabet of the weakest assumption is recomputed.

5.3.3 Properties of alphabet refinement

In this section, we prove the main properties of our algorithm. We first re-state the correctness and termination of learning *without* refinement as proven in [42].

Theorem 5 (Termination and correctness [42]) *Given components M_1 and M_2, and property φ, the learning framework in [42] terminates and it returns true if $M_1 \parallel M_2 \models \varphi$ and false otherwise.*

For correctness and termination of learning with alphabet refinement, we first show progress of refinement, meaning that at each refinement stage, new actions are discovered to be added to Σ.

Proposition 3 (Progress of alphabet refinement) *Let $\Sigma \subset \Sigma_I$ be the alphabet of the assumption at the current alphabet refinement stage. Let t be a trace of $M_2 \parallel A_{err}$ such that $t{\downarrow_\Sigma}$ leads to error on $M_1 \parallel \varphi_{err}$ by an error trace c, but $t{\downarrow_{\Sigma_I}}$ does not lead to error on $M_1 \parallel \varphi_{err}$. Then $t{\downarrow_{\Sigma_I}} \neq c{\downarrow_{\Sigma_I}}$ and there exists an action in their symmetric difference that is not in Σ.*

Proof:

We prove by contradiction that $t{\downarrow_{\Sigma_I}} \neq c{\downarrow_{\Sigma_I}}$. Suppose $t{\downarrow_{\Sigma_I}} = c{\downarrow_{\Sigma_I}}$. We know that c is an error trace on $M_1 \parallel \varphi$. Since actions of c that are not in Σ_I are internal to $M_1 \parallel \varphi$, then $c{\downarrow_{\Sigma_I}}$ also leads to error on $M_1 \parallel \varphi_{err}$. But then $t{\downarrow_{\Sigma_I}}$ leads to error on $M_1 \parallel \varphi_{err}$, which is a contradiction.

We now show that there exists an action in the difference between $t{\downarrow_{\Sigma_I}}$ and $c{\downarrow_{\Sigma_I}}$ that is not in Σ (this action will be added to Σ by alphabet refinement). Trace $t{\downarrow_{\Sigma_I}}$ is $t{\downarrow_\Sigma}$, with some interleaved actions from Σ_I. Similarly, $c{\downarrow_{\Sigma_I}}$ is $t{\downarrow_\Sigma}$ with some interleaved actions from Σ_I, since c is obtained by composing the trace LTS $t{\downarrow_\Sigma}$ with $M_1 \parallel \varphi_{err}$. Thus $t{\downarrow_\Sigma} = c{\downarrow_\Sigma}$. We again proceed by contradiction. If all the actions in the symmetric difference between $t{\downarrow_{\Sigma_I}}$ and $c{\downarrow_{\Sigma_I}}$ were in Σ, we would have $t{\downarrow_{\Sigma_I}} = t{\downarrow_\Sigma} = c{\downarrow_\Sigma} = c{\downarrow_{\Sigma_I}}$, which contradicts $t{\downarrow_{\Sigma_I}} \neq c{\downarrow_{\Sigma_I}}$. □

Intuitively, correctness for two (and n) components follows from the assume guarantee rule and the extended counterexample analysis. Termination follows from termination of the original

framework, from the progress property and also from the finiteness of Σ_I and of n. Moreover, from the progress property it follows that the refinement algorithm for two components has at most $|\Sigma_I|$ iterations.

In order to formally prove termination and correctness of learning with alphabet refinement, we use the following lemma.

Lemma 1 *For any component M_1, property φ, and interface alphabet Σ, $\langle A_{w,\Sigma}\rangle\langle M_1\rangle\langle\varphi\rangle$ holds.*

Proof:

$A_{w,\Sigma}{\downarrow_\Sigma}= A_{w,\Sigma}$. If in Definition 13 we substitute $A_{w,\Sigma}$ for M_2, we obtain that: $M_1 \parallel A_{w,\Sigma_1} \models \varphi$ if and only if $A_{w,\Sigma_1} \models A_{w,\Sigma}$. But the latter holds trivially, so we conclude that $M_1 \parallel A_{w,\Sigma_1} \models \varphi$, which is equivalent to $\langle A_{w,\Sigma}\rangle\langle M_1\rangle\langle\varphi\rangle$, always holds. □

Theorem 6 (Termination and correctness with alphabet refinement – 2 components) *Given components M_1 and M_2, and property φ, the algorithm* with alphabet refinement *terminates and returns true if $M_1 \parallel M_2 \models \varphi$ and false otherwise.*

Proof:

Correctness: When the teacher returns true, then correctness is guaranteed by the assume-guarantee compositional rule. If the teacher returns false, the extended counterexample analysis reports an error for a trace t of M_2, such that $t{\downarrow_{\Sigma_I}}$ in the context of M_1 violates the property (the same test is used in the algorithm from [42]) hence $M_1 \parallel M_2$ violates the property.

Termination: From the correctness of L*, we know that at each refinement stage (with alphabet Σ), if L* keeps receiving counterexamples, it is guaranteed to generate $A_{w,\Sigma}$. At that point, Oracle 1 will return true (from Lemma 1). Therefore, Oracle 2 will be applied, which will return either true, and terminate, or a counterexample t. This counterexample is a trace that is not in $\mathcal{L}(A_{w,\Sigma})$. It is either a real counter example (in which case the algorithm terminates) or it is a trace t such that $t{\downarrow_\Sigma}$ leads to error on $M_1 \parallel \varphi_{err}$ by an error trace c, but $t{\downarrow_{\Sigma_I}}$ does not lead to error on $M_1 \parallel \varphi_{err}$. Then from Theorem 3, we know that $t{\downarrow_{\Sigma_I}} \neq c{\downarrow_{\Sigma_I}}$ and there exists

an action in their symmetric difference that is not in Σ. The refiner will add this action (or more actions depending on the refinement strategy) to Σ and the learning algorithm is repeated for this new alphabet. Since Σ_I is finite, in the worst case, Σ grows into Σ_I, for which termination and correctness follow from Theorem 5. □

Theorem 7 (Termination and correctness with alphabet refinement – n components) *Given components $M_1, \ldots, M_n$ and property φ, the recursive algorithm* with alphabet refinement *terminates and returns true if $M_1 \parallel \ldots \parallel M_n \models \varphi$ and false otherwise.*

Proof:

The proof proceeds by induction on n and it follows from theorem above. □

We also note a property of weakest assumptions, which states that by adding actions to an alphabet Σ, the corresponding weakest assumption becomes *weaker* (*i.e.*, contains more behaviors) than the previous one.

Proposition 4 *Assume components M_1 and M_2, property φ and the corresponding interface alphabet Σ_I. Let Σ, Σ' be sets of actions such that: $\Sigma \subset \Sigma' \subset \Sigma_I$. Then: $\mathcal{L}(A_{w,\Sigma}) \subseteq \mathcal{L}(A_{w,\Sigma'}) \subseteq \mathcal{L}(A_{w,\Sigma_I})$.*

Proof:

Since $\Sigma \subseteq \Sigma'$, we know that $A_{w,\Sigma}{\downarrow_{\Sigma'}} = A_{w,\Sigma}$. By substituting, in Definition 13, $A_{w,\Sigma}$ for M_2, we obtain that: $\langle \text{true} \rangle M_1 \parallel (A_{w,\Sigma})\langle\varphi\rangle$ if and only if $\langle \text{true} \rangle A_{w,\Sigma}\langle A_{w,\Sigma'}\rangle$. From Proposition 1 we know that $\langle \mathit{true} \rangle M_1 \parallel (A_{w,\Sigma})\langle\varphi\rangle$. Therefore, $\langle \text{true} \rangle A_{w,\Sigma}\langle A_{w,\Sigma'}\rangle$ holds, which implies that $\mathcal{L}(A_{w,\Sigma}) \subseteq \mathcal{L}(A_{w,\Sigma'})$. Similarly, $\mathcal{L}(A_{w,\Sigma'}) \subseteq \mathcal{L}(A_{w,\Sigma_I})$. □

With alphabet refinement, our framework adds actions to the alphabet, which translates into adding more behaviors to the weakest assumption that L* tries to prove. This means that at each refinement stage i, when the learner is started with a new alphabet Σ_i such that $\Sigma_{i-1} \subset \Sigma_i$, the learner will try to learn an assumption A_{w,Σ_i} that is weaker than $A_{w,\Sigma_{i-1}}$, which was the goal of the learner in the previous stage. Moreover, all these assumptions are *under-approximations*

of the weakest assumption A_{w,Σ_I} that is necessary and sufficient to prove the desired property. Of course, as mentioned before, at each refinement stage the learner might stop earlier, *i.e.*, before computing the corresponding weakest assumption. The above property allows re-use of learning results across refinement stages: when learning $A_{w,\Sigma'}$, the learner can start from the table computed for $A_{w,\Sigma}$ in the previous refinement stage (instead of starting from scratch).

5.3.4 Extensions to other rules

Alphabet refinement also applies to the rules CIRC-N and SYM-N. As mentioned, CIRC-N is a special case of the recursive application of rule ASYM for $n+1$ components, where the first and last component coincide. Therefore alphabet refinement applies to CIRC-N as we described here.

For rule SYM-N, the counterexample analysis for the error trace t obtained from checking premise $n+1$ is extended for each component M_i, for $i = 1 \ldots n$. The extension works similarly to that for ASYM discussed earlier in this section. The error trace t is simulated on each $M_i \parallel co\varphi$ with the current assumption alphabet:

- if t is violating for some i, then we check whether t, with the entire alphabet of the weakest assumption for i is still violating. If it is, then t is a real error trace for M_i. If it is not, the alphabet of the current assumption for i is refined with actions from the alphabet of the corresponding weakest assumption;

- if t is a real error trace for all i, then it is reported as a real violation of the property on the entire system.

If alphabet refinement takes place for some i, the learning of the assumption for this i is restarted with the refined alphabet, and premise $n+1$ is re-checked with the new learned assumption for i.

5.3.5 Experiments

We implemented alphabet refinement for learning with rules ASYM, SYM-N, CIRC-N in LTSA and compared to learning without alphabet refinement on the models and properties described in Section 5.2. The goal of the evaluation was to assess the effect of alphabet refinement on learning, to compare this effect for the different rules, and to also assess the effect of alphabet refinement on the scalability of compositional verification by learning, as compared to non-compositional verification.

Experimental set-up

We performed five sets of experiments.

1. We compared the different alphabet refinement heuristics for Rule ASYM and *2-way* decompositions (using an experimental set-up similar to [41]).

2. We used the same setup and the best heuristic found in the first set, to compare learning *with alphabet refinement* to learning *without* alphabet refinement.

3. We compared the recursive implementation of the refinement algorithm for the same rule, with monolithic (non-compositional) verification, for increasing number of components.

4. We used the same setup as in the third set of experiments for Rules CIRC-N and

5. SYM-N, omitting monolithic verification since these rules do not outperform ASYM.

For the first two sets of experiments, we used 2-way decompositions as described in Section 5.2. For the next two sets, we implemented an additional heuristic for computing the *ordering* in which the modules are considered by the recursive learning with refinement. The heuristic is meant to minimize the interface between modules and follows from the observation that the ordering of the modules in the sequence $M_1, \ldots, M_n$ influences the sizes of the interface alphabets $\Sigma_I^1, \ldots \Sigma_I^n$ that are used by the recursive algorithm. We generated offline all

possible orders and associated interface alphabets and chose the order that minimizes the sum $\sum_{j=1..n} |\Sigma_I^j|$.

All experiments were performed on a Dell PC with a 2.8 GHz Intel Pentium 4 CPU and 1.0 GB RAM, running Linux Fedora Core 4 and using Sun's Java SDK version 1.5.

Experimental results

The results are shown in Tables 5.1,5.2, 5.3, 5.4, and 5.5. In the tables, $|A|$ is the *maximum* assumption size reached during learning, 'Mem.' is the *maximum* memory used by LTSA to check assume-guarantee triples, measured in MB, and 'Time' is the total CPU running time, measured in seconds. Column 'Monolithic' reports the memory and run-time of non-compositional model checking. We set a limit of 30 minutes for each run. The sign '–' indicates that the limit of 1GB of memory or the time limit has been exceeded. For these cases, the data is reported as it was when the limit was reached.

In Table 5.1, we show the performance of the different alphabet refinement heuristics, for two-way decompositions of the systems we studied. As these results indicate that 'bwd' heuristic is slightly better than the others, we used this heuristic for alphabet refinement in the rest of the experiments.

Table 5.2 shows the effect of alphabet refinement on learning.

Table 5.3 shows the performance of the recursive implementation of learning with rule ASYM, with and without alphabet refinement, as well as that of monolithic (non-compositional) verification, for increasing number of components.

The results for rules CIRC-N and SYM-N are in Tables 5.4 and 5.5, respectively.

Discussion. The results in all tables show that alphabet refinement improves learning significantly. Table 5.2 shows that alphabet refinement improved the assumption size in all cases, and in a few, up to two orders of magnitude (see Gas Station with $k = 2, 3$, Chiron, Property 2, with $k = 5$, MER with $k = 3$). It improved memory consumption in 10 out of 15 cases. It also improved running time, as for Gas Station and for MER with $k = 3, 4$ learning without

Table 5.1: Comparison of three different alphabet refinement heuristics for Rule ASYM and 2-way decompositions.

Case	k	**Refinement + bwd**			**Refinement + fwd**			**Refinement + allDiff**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	8	3.29	2.70	37	6.47	36.52	18	4.58	7.76
Station	4	8	24.06	19.58	37	46.95	256.82	18	36.06	52.72
	5	8	248.17	183.70	20	414.19	–	18	360.04	530.71
Chiron,	2	8	1.22	3.53	8	1.22	1.86	8	1.22	1.90
Prop. 1	3	20	6.10	23.82	20	6.06	7.40	20	6.06	7.77
	4	38	44.20	154.00	38	44.20	33.13	38	44.20	35.32
	5	110	–	300	110	–	300	110	–	300
Chiron,	2	3	1.05	0.73	3	1.05	0.73	3	1.05	0.74
Prop. 2	3	3	2.20	0.93	3	2.20	0.92	3	2.20	0.92
	4	3	8.13	1.69	3	8.13	1.67	3	8.13	1.67
	5	3	163.85	18.08	3	163.85	18.05	3	163.85	17.99
MER	2	6	1.78	1.01	6	1.78	1.02	6	1.78	1.01
	3	8	10.56	11.86	8	10.56	11.86	8	10.56	11.85
	4	10	514.41	1193.53	10	514.41	1225.95	10	514.41	1226.80
Rover	2	4	2.37	2.53	11	2.67	4.17	11	2.54	2.88

refinement did not finish within the time limit, whereas with refinement it did. The benefit of alphabet refinement is even more obvious in Table 5.3 where 'No refinement' exceeded the time limit in all but one case, whereas refinement completed in 14 of 16 cases, producing smaller assumption sizes in all the cases, and up to two orders of magnitude smaller in a few; the memory consumption was also improved in all cases, and up to two orders of magnitude in a few of them.

The results in Table 5.3 indicate that learning with refinement scales better than without refinement for increasing number of components. Also, as k increases, the memory and time

Table 5.2: Comparison of learning for 2-way decompositions and Rule ASYM, with and without alphabet refinement.

Case	k	**No refinement**			**Refinement + bwd**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	177	4.34	–	8	3.29	2.70
Station	4	195	100.21	–	8	24.06	19.58
	5	53	263.38	–	8	248.17	183.70
Chiron,	2	9	1.30	1.23	8	1.22	3.53
Prop. 1	3	21	5.70	5.71	20	6.10	23.82
	4	39	27.10	28.00	38	44.20	154.00
	5	111	569.24	607.72	110	–	300
Chiron,	2	9	116	110	3	1.05	0.73
Prop. 2	3	25	4.45	6.39	3	2.20	0.93
	4	45	25.49	32.18	3	8.13	1.69
	5	122	131.49	246.84	3	163.85	18.08
MER	2	40	6.57	7.84	6	1.78	1.01
	3	377	158.97	–	8	10.56	11.86
	4	38	391.24	–	10	514.41	1193.53
Rover	2	11	2.65	1.82	4	2.37	2.53

consumption for ‘Refinement’ grows slower than that of ‘Monolithic’. For Gas Station, Chiron (Property 2), and MER, for small values of k, ‘Refinement’ consumes more memory than ‘Monolithic’, but as k increases, the gap is narrowing, and for the largest value of k ‘Refinement’ becomes better than ‘Monolithic’. This leads to cases where, for a large enough parameter value, ‘Monolithic’ runs out of memory, whereas ‘Refinement’ succeeds, as it is the case for MER with $k = 6$.

Tables 5.5 and 5.4 indicate that the effect of alphabet refinement is insensitive to the rule being used.

Table 5.3: Comparison of recursive learning for ASYM with and without alphabet refinement, and monolithic verification.

Case	k	ASYM			ASYM + **ref**			**Monolithic**	
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	Mem.	Time
Gas	3	473	109.97	–	25	2.41	13.29	1.41	0.034
Station	4	287	203.05	–	25	3.42	22.50	2.29	0.13
	5	268	283.18	–	25	5.34	46.90	6.33	0.78
Chiron,	2	352	343.62	–	4	0.93	2.38	0.88	0.041
Prop. 1	3	182	114.57	–	4	1.18	2.77	1.53	0.062
	4	182	116.66	–	4	2.13	3.53	2.75	0.147
	5	182	115.07	–	4	7.82	6.56	13.39	1.202
Chiron,	2	190	107.45	–	11	1.68	40.11	1.21	0.035
Prop. 2	3	245	68.15	–	114	28	–	1.63	0.072
	4	245	70.26	–	103	23.81	–	2.89	0.173
	5	245	76.10	–	76	32.03	–	15.70	1.53
MER	2	40	8.65	21.90	6	1.23	1.60	1.04	0.024
	3	501	240.06	–	8	3.54	4.76	4.05	0.111
	4	273	101.59	–	10	9.61	13.68	14.29	1.46
	5	200	78.10	–	12	19.03	35.23	14.24	27.73
	6	162	84.95	–	14	47.09	91.82	–	600

Chiron, Property 2, was a challenging case for learning with (or without) alphabet refinement and asymmetric rules. We looked at it more closely. After inspecting the models, we noticed that several modules do not influence Property 2. However, these modules do communicate with the rest of the system through actions that appear in the counterexamples reported by our framework. As a result, alphabet refinement introduces 'un-necessary' actions. If we eliminate these modules, the property still holds in the remaining system. The performance of learning with refinement is greatly improved when applied to this reduced system (*e.g.*, for $k = 3$, the size of the largest assumption is 13) and is better than monolithic. We may be able

Table 5.4: Comparison of learning for CIRC-N with and without alphabet refinement.

Case	k	CIRC-N			CIRC-N + **ref**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	205	108.96	–	25	2.43	15.10
Station	4	205	107.00	–	25	3.66	25.90
	5	199	105.89	–	25	5.77	58.74
Chiron,	2	259	78.03	–	4	0.96	2.71
Prop. 1	3	253	77.26	–	4	1.20	3.11
	4	253	77.90	–	4	2.21	3.88
	5	253	81.43	–	4	7.77	7.14
Chiron,	2	67	100.91	–	327	44.17	–
Prop. 2	3	245	75.76	–	114	26.61	–
	4	245	77.93	–	103	23.93	–
	5	245	81.33	–	76	32.07	–
MER	2	148	597.30	–	6	1.89	1.51
	3	281	292.01	–	8	3.53	4.00
	4	239	237.22	–	10	9.60	10.64
	5	221	115.37	–	12	19.03	27.56
	6	200	88.00	–	14	47.09	79.17

to develop refinement heuristics that are less sensitive to such problems, but we cannot expect heuristics to always produce the optimal alphabet. Therefore, in the future, we also plan to investigate slicing-like techniques to eliminate modules that do not affect a given property. It is worth noting that for the symmetric rule this case becomes easy, so there is value in using different rules, even if ASYM shows the best performance overall.

5.3.6 Comparison with related work

Since the original work framework of [53, 42], several other frameworks that use L* for learning assumptions have been developed – [2] presents a symbolic BDD implementation using

Table 5.5: Comparison of learning for SYM-N with and without alphabet refinement.

Case	k	SYM-N			SYM-N + **ref**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	7	1.34	–	83	31.94	874.39
Station	4	7	2.05	–	139	38.98	–
	5	7	2.77	–	157	52.10	–
Chiron,	2	19	2.21	–	21	4.56	52.14
Prop. 1	3	19	2.65	–	21	4.99	65.50
	4	19	4.70	–	21	6.74	70.40
	5	19	17.65	–	21	28.38	249.3
Chiron,	2	7	1.16	–	8	0.93	6.35
Prop. 2	3	7	1.36	–	16	1.43	9.40
	4	7	2.29	–	32	3.51	16.00
	5	7	8.20	–	64	20.90	57.94
MER	2	40	6.56	9.00	6	1.69	1.64
	3	64	11.90	25.95	8	3.12	4.03
	4	88	1.82	83.18	10	9.61	9.72
	5	112	27.87	239.05	12	19.03	22.74
	6	136	47.01	608.44	14	47.01	47.90

NuSMV. This symbolic version was extended in [79] with algorithms that decompose models using hypergraph partitioning, to optimize the performance of learning on resulting decompositions. Different decompositions are also studied in [41] where the best two-way decompositions are computed for model-checking with the LTSA and FLAVERS tools. We follow a direction orthogonal to the latter two approaches and try to improve learning not by automating and optimizing decompositions, but rather by discovering small interface alphabets. Our approach can be combined with the decomposition approaches, by applying interface alphabet refinement in the context of the discovered decompositions. L* has also been used in [1] to synthesize interfaces for Java classes, and in [91] to check component compatibility after

component updates.

A similar idea to our alphabet refinement for L* in the context of assume-guarantee verification has been developed independently in [24]. In that work, L* is started with an empty alphabet, and, similar to our approach, the assumption alphabet is refined when a spurious counterexample is obtained. At each refinement stage, a new minimal alphabet is computed that eliminates all spurious counterexamples seen so far. The computation of such a minimal alphabet is shown to be NP-hard. In contrast, we use much cheaper heuristics, but do not guarantee that the computed alphabet is minimal. The approach presented in [93] improves upon assume-guarantee learning for systems that communicate based on shared memory, by using SAT based model checking and alphabet clustering.

The theoretical results in [73] show that circular assume-guarantee rules can not be both sound and complete. These results do not apply to rules such as ours that involve additional assumptions which appear only in the premises and not in the conclusions of the rules. Note that completeness is not required by our framework (however incompleteness may lead to inconclusive results).

Our approach is similar in spirit to counterexample-guided abstraction refinement (CEGAR) [36]. CEGAR computes and analyzes abstractions of programs (usually using a set of abstraction predicates) and refines them based on spurious counter-examples. However, there are some important differences between CEGAR and our algorithm. Alphabet refinement works on actions rather than predicates, it is applied compositionally in an assume-guarantee style and it computes under-approximations (of assumptions) rather than behavioral over-approximations (as it happens in CEGAR).

The work of [61] proposes a CEGAR approach to interface synthesis for Java libraries. This work does not use learning, nor does it address the use of the resulting interfaces in assume-guarantee verification.

Generating assumptions for a component is similar to generating component interfaces to handle intermediate state explosion in compositional reachability analysis. Several approaches

have been defined to automatically abstract a component's environment to obtain interfaces [29, 67, 30]. These approaches do not address the incremental refinement of interfaces, and they could benefit from our new approach.

5.4 Assumption Generation by Abstraction Refinement

In this section we present our algorithm AGAR, including an adaptation of CEGAR to LTSs with interfaces, a motivating example showing that AGAR can lead to smaller assumptions in fewer iterations than a learning-based approach, and extensions of AGAR with alphabet refinement and with recursive application to $n > 2$ components. We then show an experimental comparison with learning-based approaches, and other comparison with related work, after which we conclude.

5.4.1 Motivating example

We motivate our approach using the input-output example from Section 5.2. We show that even on this simple example AGAR leads to smaller assumptions in fewer iterations than the learning approach, and therefore it potentially leads faster to smaller verification problems.

Let $M_1 =$ *Input*, $M_2 =$ *Output*, and $\varphi =$ *Order*. As mentioned, we aim to automatically compute an assumption according to Rule ASYM. Instead of "guessing" an assumption and then checking both premises of the rule, as in the learning approaches, we *build* an abstraction that satisfies *Premise 2* by construction. Therefore, all that needs to be checked is *Premise 1*.

The initial abstraction A of *Output* is illustrated in Figure 5.9(a). Its alphabet consists of the interface between *Input* and the *Order* property on one side, and *Output* on the other, *i.e.*, the alphabet of A is $\Sigma_I = \{(\Sigma_{\mathit{Input}} \cup \Sigma_{\mathit{Order}}) \cap \Sigma_{\mathit{Output}}$. The LTS A is constructed simply by mapping all concrete states in *Output* to the same abstract state 0 which has a self-loop on every action in Σ_I and no other transitions. By construction, A is an overapproximation of M_2, *i.e.*, $\mathcal{L}(M_2{\downarrow_{\Sigma_I}}) \subseteq \mathcal{L}(A)$, and therefore *Premise 2* $\langle \mathsf{true} \rangle\ M_2\ \langle A \rangle$ holds. Checking *Premise 1*

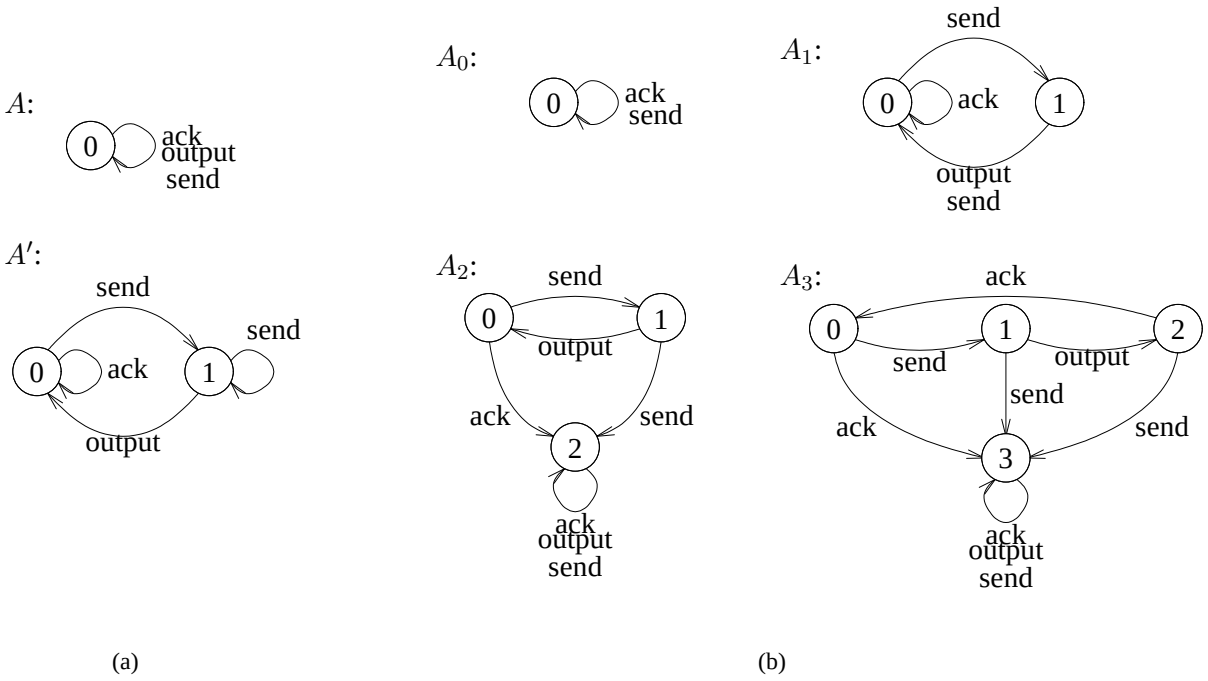


Figure 5.9: Assumptions computed (a) with AGAR and (b) with L*.

of the assume-guarantee rule using A as the assumption fails, with abstract counterexample: $0,\ \textit{output},\ 0$. We simulate this counterexample on M_2 and find that it is spurious (*i.e.*, it does not correspond to a trace in M_2), therefore A needs to be refined so that the refined abstraction no longer contains this trace. We split abstract state 0 into two new abstract states: abstract state 0, representing concrete states 0 and 2 that do not have an *output* action, and abstract state 1, representing concrete state 1 that has an *output* action, and adjust the transitions accordingly. The refined abstraction A', shown in Figure 5.9(a), is checked again for *Premise 1* and this time it passes, therefore AGAR terminates and reports that the property holds.

The sequence of assumptions learned with L* is shown in Figure 5.9(b). The assumption computed by AGAR, even if still deterministic, has half the number of states and is computed in half the number of iterations than that obtained from learning. It is possible to obtain a smaller (deterministic) assumption than in learning, even if learning is guaranteed to produce a minimal automaton: the minimum is taken over all possible automata for the same language.

In our example, the languages of the assumptions resulting from AGAR and L* are different. For instance, A' does not accept the trace: ack, output, which is a trace in A_3. Note also the different initial assumptions computed by the two algorithms. L* builds its initial assumption by collecting all the singleton traces a, for $a \in$ ack, send, output that do not lead to error in $M_1 \parallel \varphi$. Note also that our algorithm acts monotonically in terms of assumption traces: it removes spurious traces, whereas L* both adds and removes traces: it can add traces that later on may be removed following a failure of *Premise 2*, or can remove traces that later on may be added for passing *Premise 1*.

5.4.2 Assume-Guarantee Abstraction Refinement (AGAR)

The abstraction refinement presented here is an adaptation of the CEGAR framework of [36], with the following notable differences: 1) abstraction refinement is performed in the context of LTSs; abstract transitions for LTSs are computed using *closure* with respect to actions that are not in their interface alphabet, and 2) counterexample analysis is performed in an assume-guarantee style: a counterexample obtained from model checking one component is used to refine abstractions of a different component.

In this section, we start by describing, independently of the assume-guarantee rule, abstraction refinement as applied to LTSs. We then describe how we use this abstraction refinement in an iterative algorithm (AGAR) that computes assumptions for Rule ASYM. Later on, we combine AGAR with an orthogonal algorithm that performs iterative refinement of the interface alphabet between the analyzed components.

Abstraction refinement for LTSs

Abstraction. Let $C = \langle Q_C, \Sigma_C, \delta^C, q_0^C \rangle$ be an LTS that we refer to as *concrete*. Let alphabet Σ_A be such that $\Sigma_A \subseteq \Sigma_C$. An *abstraction* A of C is an LTS $\langle Q_A, \Sigma_A, \delta^A, q_0^A \rangle$ such that there exists a surjection $\alpha : Q_C \rightarrow Q_A$, called the *abstraction* function, that maps each *concrete state* $q^C \in Q_C$ to an *abstract state* $q^A \in Q_A$; q_0^A must be such that $\alpha(q_0^C) = q_0^A$. The *concretization*

function $\gamma : Q_A \rightarrow 2^{Q_C}$ is defined for any $q^A \in Q_A$ as $\gamma(q^A) = \{q^C \in Q_C \mid \alpha(q^C) = q^A\}$. Note that γ induces a partition on Q_C, namely $\{\gamma(q^A) \mid q^A \in Q_A\}$.

To define the abstract transition relation δ^A, we first introduce the notion of reachability with respect to a subset alphabet. For $q^C \in C, a \in \Sigma_C$, we define the set ***Reachable***$_C(q^C, a, \Sigma_A)$ of concrete states q_i^C reachable from q^C on action a, under the transitive closure of δ^C over actions in $(\Sigma_C \setminus \Sigma_A) \cup \{\tau\}$:

$$\textit{Reachable}_C(q^C, a, \Sigma_A) = \{q_i^C \in C | \exists t, t' \in ((\Sigma_C \setminus \Sigma_A) \cup \{\tau\})^* \cdot q^C \stackrel{t}{\Rightarrow} q_i^C \text{ or } q^C \stackrel{t,b,t'}{\Rightarrow} q_i^C\}.$$

We define the abstraction to be *existential*, but using ***Reachable***$_C$ instead of the usual transition relation of C [36]: $\exists(q_i^A, a, q_j^A) \in \delta^A$ iff

$$\exists q_i^C, q_j^C \in C \cdot \alpha(q_i^C) = q_i^A, \alpha(q_j^C) = q_j^A, \text{ and } q_j^C \in \textit{Reachable}_C(q_i^C, a, \Sigma_A) \qquad (5.1)$$

From the above definition and that of weak simulation [77], it follows that the abstraction defines a weak simulation relation between $C{\downarrow_{\Sigma_A}}$ and A. It is known that weak simulation implies trace inclusion [77]. We therefore have the following:

Proposition 5 *Given concrete LTS C and its abstraction A defined as above, $\mathcal{L}(C{\downarrow_{\Sigma_A}}) \subseteq \mathcal{L}(A)$, and consequently $\langle \mathsf{true} \rangle\ C\ \langle A \rangle$ hold.*

The CEGAR algorithm for LTSs is defined by Algorithm 1. It takes as inputs a concrete system C, an abstraction A (as defined above), and an abstract counterexample path p (in A). The algorithm analyzes the counterexample (lines 1–6) to see if it is real, in which case it is returned (line 13) or spurious, in which case it is used to refine the abstraction (lines 7–11). The refined abstraction A' is such that it no longer contains p. We discuss Algorithm 1 in more detail below.

Analysis of abstract counterexamples. Suppose we have obtained an *abstract counterexample* in the form of a path $p = q_0^A, a_1, q_1^A, a_2, \ldots, a_n, q_n^A$ in the abstraction A of C. We want to determine if it corresponds to a concrete path in C. For this we need to "play" (i.e. symbolically simulate) p in C from the initial state q_0^C. We do so considering that $\Sigma_A \subseteq \Sigma_C$ and thus we use ***Reachable***$_C$ again.

We first extend *Reachable*$_C$ to sets: for $S \subseteq Q_C$, *Reachable*$_C(S, a, \Sigma_A) = \{q_j^C \in C \mid \exists q_i^C \in S. q_j^C \in$ *Reachable*$(q_i^C, a, \Sigma_A)\}$. We play the abstract counterexample p following [36]. We start at step 0 with the set $S_0 = \{q_0^C\}$ of concrete states, and the first transition $q_0^A \xrightarrow{a_1} q_1^A$ from p. Note that $S_0 = \{q_0^C\} \cap \gamma(q_0^A)$. At each step $i \in \{1, \ldots, n\}$, we compute the set $S_i = \gamma(q_i^A) \cap$ *Reachable*$_C(S_{i-1}, a_i, \Sigma_A)$. If, for some $i \leq n$, S_i is empty, the abstract counterexample is spurious and we need to refine the abstraction to eliminate it. Otherwise, the counterexample corresponds to a concrete path.

Abstraction refinement. The abstraction refinement is performed in lines 8–10 of Algorithm 1: p is spurious because abstract state q_{i-1}^A does not distinguish between two disjoint, non-empty sets of concrete states [36]: (i) those that reach, with action a_i, states in the concretization of q_i^A (these are the states defined as $\gamma(x_{i-1}^A)$ in line 8) and (ii) those reached so far from q_0^C with the prefix $a_1, a_2, \ldots, a_{i-1}$, *i.e.*, the states in S_{i-1}.

To eliminate the spurious abstract path, we need to refine A by splitting its state q_{i-1}^A into (at least) two new abstract states that separate the (concrete) states of types (i) and (ii) (line 9). We split q_{i-1}^A into x_{i-1}^A where $\gamma(x_{i-1}^A)$ contains the set of states in (i) and z_{i-1}^A where $\gamma(z_{i-1}^A)$ contains the set of states in (ii) and any remaining states in $\gamma(q_{i-1}^A)$. Note that this results in a finer partition of the concrete states. After the splitting, we update the abstract transitions in line 10. The refined abstraction A' has the same transitions as A except for those incoming or outgoing for the split state q_{i-1}^A: they are readjusted to point to or from the states x_{i-1}^A, z_{i-1}^A according to condition 5.1. We therefore can conclude that:

Lemma 2 *If a counterexample p input to Algorithm 1 is spurious, the returned abstraction A' results in a strictly finer partition than A and does not contain p.*

The AGAR algorithm

The pseudocode that combines Algorithm 1 with Rule ASYM is given in Algorithm 2. Recall that Σ_I denotes the alphabet $(\Sigma_{M_1} \cup \Sigma_\varphi) \cap \Sigma_{M_2}$ of the interface between M_1 and M_2, with respect to φ. The algorithm checks that $M_1 \parallel M_2$ satisfies φ using Rule ASYM. It builds

Algorithm 1 CEGAR for LTSs with respect to subset alphabets

Inputs: Concrete LTS C, its abstraction A, and an abstract counterexample $p = q_0^A, a_1, q_1^A, a_2, \ldots, a_n, q_n^A$ in A.

Outputs: a concrete counterexample t, if p is not spurious, or a refined abstraction A' without path p, if p is spurious.

1: $i \leftarrow 0$

2: $S_0 \leftarrow \{q_0^C\}$

3: **while** $S_i \neq \emptyset \wedge i \leq n-1$ **do**

4: $\quad i \leftarrow i+1$

5: $\quad S_i \leftarrow \gamma(q_i^A) \cap \textit{Reachable}_C(S_{i-1}, a_i, \Sigma_A)$

6: **end while**

7: **if** $S_i = \emptyset$ **then**

8: $\quad$ split q_{i-1}^A into two new abstract states x_{i-1}^A, z_{i-1}^A s.t. $\gamma(x_{i-1}^A) = \gamma(q_{i-1}^A) \cap \{q^C \mid \textit{Reachable}_C(q^C, a_i, \Sigma_A) \cap q_i^A \neq \emptyset\}$, $\gamma(z_{i-1}^A) = \gamma(q_{i-1}^A) \setminus \gamma(x_{i-1}^A)$

9: $\quad$ build new abstraction A' with $Q_{A'} = Q_A \setminus \{q_{i-1}^A\} \cup \{x_{i-1}^A, z_{i-1}^A\}$

10: $\quad$ change only incoming and outgoing transitions for q_{i-1}^A in A to/from $\{x_{i-1}^A, z_{i-1}^A\}$ in refined abstraction A', according to Definition 5.1

11: $\quad$ **return** A'

12: **else**

13: $\quad$ **return** concrete trace $t \leftarrow \sigma(p)$

14: **end if**

abstractions A of M_2 in an iterative fashion (while loop at lines 2–15); these abstractions are used to check *Premise 1* of the assume guarantee rule using model checking (lines 3–5). If the check is successful, then, according to the rule (and since A satisfies *Premise 2* by construction), φ indeed holds in $M_1 \parallel M_2$ and the algorithm returns "true". Otherwise, a counterexample p is obtained from model checking *Premise 1* (line 7) and Algorithm 1 is invoked to check if p corresponds to a real path in M_2 (in which case it means p is a real error

Algorithm 2 AGAR: assume-guarantee verification by abstraction-refinement

Inputs: Component LTSs M_1, M_2, safety property LTS φ, and alphabet $\Sigma_A = \Sigma_I$.

Outputs: **true** if $M_1 \parallel M_2$ satisfies φ, **false** with a counterexample, otherwise.

Uses: Algorithm 1

1: Compute initial abstraction A of M_2, with a single state q_0^A having self-loops on all actions in Σ_A
2: **while true do**
3: Check *Premise 1*: $\langle A \rangle\ M_1\ \langle \varphi \rangle$
4: **if** successful **then**
5: **return true**
6: **else**
7: Get counterexample $o = q_0, b_1, q_1, b_2, \ldots, b_l, q_l$ from line 3, where each $q_i = (q_i^A, q_i^1, p_i)$
8: Project o on A to get $o' = q_0^A, b_1, q_1^A, b_2, q_2^A, \ldots b_l, q_l^A$
9: Project o' on Σ_A to get abstract counterexample $p = q_0^A, a_1, q_1^A, \ldots, a_n, q_n^A$ in A.
10: **end if**
11: Call Algorithm 1 with inputs: M_2, A, p
12: **if** Algorithm 1 returned real counterexample t **then**
13: **return false** with counterexample t
14: **else**
15: $A = A'$
16: **end if**
17: **end while**

in $M_1 \parallel M_2$ and this is reported to the user in line 11). If p is spurious, Algorithm 1 returns a refined abstraction A' for which we repeat the whole process starting from checking *Premise 1*.

Obtaining an abstract counterexample. As mentioned, we use counterexamples from failed

checks of Premise 1 (that involves checking component M_1) to refine abstractions of M_2. Obtaining an abstract counterexample involves several steps (lines 7–9). First, a counterexample from line 4 is a path $o = q_0, b_1, q_1, b_2, \ldots, b_l, q_l$ in $A \parallel M_1 \parallel \varphi_{err}$. Thus, for every $i \in \{0, l\}$, q_i is a triple of states (q_i^A, q_i^1, p_i) from $A \times M_1 \times \varphi_{err}$. We first project every triple on A to obtain the sequence $o' = q_0^A, b_1, q_1^A, b_2, q_2^A, \ldots, b_l q_l^A$; o' is not yet a path in A as it may contain actions from M_1 and φ_{err} that are not observable to A; those actions have to be between the same consecutive abstract states in the sequence, since they do not change the state of A; we eliminate from o' those actions and the duplicate abstract states that they connect, and finally obtain p that we pass to Algorithm 1.

Theorem 8 *Our algorithm (*AGAR*) computes a sequence of increasingly refined abstractions of M_2 until both premises of Rule* ASYM *are satisfied, and we conclude that the property is satisfied by $M_1 \parallel M_2$, or a real counterexample is found that shows the violation of the property on $M_1 \parallel M_2$.*

Proof:

Correctness The algorithm terminates when *Premise 1* is satisfied by the current abstraction or when a real counterexample is returned by Algorithm 1. In the former case, since the abstraction satisfies *Premise 2* by construction (Proposition 5), Rule ASYM ensures that $M_1 \parallel M_2$ indeed satisfies φ, so AGAR correctly returns answer "true". In the latter case, the counterexample returned by Algorithm 1 is a common trace of M_1 and of M_2 that leads to error in φ_{err}. This shows that property φ is violated on $M_1 \parallel M_2$ and in this case again AGAR correctly returns answer "false".

Termination AGAR continues to refine the abstraction until a real counterexample is reported or the property holds. Refining the abstraction always results in a finer partition of its states (Lemma 2), and is thus guaranteed to terminate since in the worst case it converges to M_2 which is finite-state. □

If M_2 has n states, AGAR makes at most n refinement iterations, and in each iteration, counterexample analysis performs at most m transitive closure operations (for computing

Reachable$_{M_2}$), each of cost $O(n^3)$, where m is the length of the longest counterexample analyzed. This bound is not very tight as the closure steps are done on-the-fly to seldom exhibit worst-case behavior, and actually involve only parts of M_2's transition relation as needed.

AGAR **with alphabet refinement**

In [49] we introduced an *alphabet refinement* technique to reduce the alphabet of the assumptions learned with L*. This technique improved significantly the performance of compositional verification. We show here how alphabet refinement can be similarly introduced in AGAR. Instead of the full interface alphabet Σ_I, we start AGAR from a small subset $\Sigma_A \subseteq \Sigma_I$. A good strategy is to start from those actions in Σ_I that appear in the property to be verified, since the verification should depend on them. We then run Algorithm 2 with this small Σ_A. Alphabet refinement introduces an extra layer of approximation, due to the smaller alphabet being used.

The pseudocode is in Algorithm 3. This algorithm adds an outer loop to AGAR (lines 1–15). At each iteration, it invokes AGAR (line 2) for the current alphabet Σ_A. If AGAR returns "true", it means that alphabet Σ_A is enough for proving the property (and "true" is returned to the user). Otherwise, the returned counterexample needs to be further analyzed (lines 5–13) to see if it corresponds to a real error (which is returned to the user in line 9) or it is spurious due to the approximation introduced by the smaller interface alphabet, in which case it is used to refine this alphabet (lines 11–12).

Additional counterexample analysis As explained in [49], when $\Sigma_A \subset \Sigma_I$, the counterexamples obtained by applying Rule ASYM may be spurious, in which case Σ_A needs to be extended. Intuitively, a counterexample is real if it is still a counterexample when considered with Σ_I. For counterexample analysis, we modify Algorithm 2 to also output the trace $s = \sigma(o')$ of actions along the intermediate path o' obtained at its line 8. Since p is a path obtained from o' by eliminating transitions labeled with actions from $\Sigma_I \setminus \Sigma_A$ (See Section 5.4.2) and $t = \sigma(p)$, it follows that s is an "extension" of t to Σ_I.

We check whether $s{\downarrow_{\Sigma_I}}$ is a trace of M_2 by making it into a trace LTS ending with the error

Algorithm 3 AGAR with alphabet refinement

Inputs: Component LTSs M_1, M_2, safety property LTS φ, and alphabet $\Sigma_A \subseteq \Sigma_I$.

Outputs: **true** if $M_1 \parallel M_2$ satisfies φ, **false** with a counterexample, otherwise.

Uses: Algorithm 2

1: **while true do**
2: Call Algorithm 2 with $M_1, M_2, \varphi, \Sigma_A$.
3: **if** Algorithm 2 returned **true** **then**
4: **return true**
5: **else**
6: Obtain counterexample $t = a_1, \ldots, a_n$ from Algorithm 2 **and** trace $s = \sigma(o')$ from line 8 of Algorithm 2.
7: Check if error reachable in $s^{err}{\downarrow_{\Sigma_I}} \parallel M_2$ where $s^{err}{\downarrow_{\Sigma_I}}$ is the trace-LTS ending with an extra transition into error state π
8: **if** error reached **then**
9: **return false** with counterexample $s{\downarrow_{\Sigma_I}}$
10: **else**
11: Compare t to $s{\downarrow_{\Sigma_I}}$ to find difference action set Σ
12: $\Sigma_A \leftarrow \Sigma_A \cup \Sigma$
13: **end if**
14: **end if**
15: **end while**

state π, and whose alphabet is Σ_I (line 7). Since M_2 does not contain π, the only way to reach error is if $s{\downarrow_{\Sigma_I}}$ is a trace of M_2; if we reach error, the counterexample t is real. If $s{\downarrow_{\Sigma_I}}$ is not a trace of M_2, since t is, we need to refine the current alphabet Σ_A. At this point we have two traces, $s{\downarrow_{\Sigma_I}}$ and t that agree with respect to Σ_A and only differ on the actions from $\Sigma_I \setminus \Sigma_A$; since one trace is in M_2 and the other is not, we are guaranteed to find in their symmetric difference at least an action that we can add to Σ_A to eliminate the spurious counterexample

t. We include the new action(s) and then repeat AGAR with the new alphabet. Termination follows from the fact that the interface alphabet is finite.

5.4.3 Evaluation

We implemented AGAR with alphabet refinement for Rule 1 in the LTSA tool. We compared AGAR with learning based assume guarantee reasoning, for Rule ASYM and 2-way decompositions using the same data and experimental setup as in Section 5.3. We report the maximum assumption size (*i.e.*, number of states) reached '$|A|$'), the memory consumed ('Mem.') in MB, and the time ('Time') in seconds. A '–' indicates that the limit of 1G of memory or 30 minutes has been exceeded. For those cases, the other quantities are shown as they were when the limit was reached. We also highlight in bold font the best results.

The results for the first set of experiments are shown in Tables 5.6 and 5.7. AGAR shows better results than learning in about 75% of the cases without alphabet refinement, and in slightly more than half of the cases with alphabet refinement. We noticed that the relative sizes of $M_1 \parallel \varphi_{err}$ and M_2 seem to influence the performance of the two algorithms. The numbers of states on each side of the two-way decompositions are in Table 5.8 in rows 'S1' and 'S2', where S1 is $|M_1 \parallel \varphi_{err}|$ and S2 is $|M_2|$. For *Gas Station*, where M_2 is consistently smaller, AGAR is consistently better, while for *Chiron*, as the size of M_2 becomes much larger, the performance of AGAR seems to degrade. Furthermore, we observed that the learning runs exercise more the first component, whereas AGAR exercises both. We therefore considered a second set of experiments were we tried to compare the relative performance of the two approaches for two-way system decompositions that are more balanced in terms of number of states.

We generated off-line all the possible two-way decompositions and chose those minimizing the difference in number of states between $M_1 \parallel \varphi_{err}$ and M_2. The rest of the setup remained the same. The sizes for the balanced decompositions we found are in Table 5.9, and the results for these new decompositions are in Tables 5.10 and 5.11 (for MER, in only one case we found a more balanced partition than previously; for Rover there are no other decompositions than

Table 5.6: Comparison of AGAR and learning for Rule ASYM and 2-way decompositions, without alphabet refinement.

Case	k	AGAR			**Learning**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	**16**	**4.11**	**3.33**	177	42.83	–
Station	4	**19**	**37.43**	**23.12**	195	100.17	–
	5	**22**	**359.53**	**278.63**	45	206.61	–
Chiron,	2	10	**1.30**	**0.92**	9	**1.30**	1.69
Prop. 1	3	36	**2.59**	**5.94**	**21**	5.59	7.08
	4	160	**8.71**	152.34	**39**	27.1	**32.05**
	5	4	55.14	–	**111**	**569.23**	**676.02**
Chiron,	2	**4**	**1.07**	**0.50**	9	1.14	1.57
Prop. 2	3	**8**	**1.84**	**1.60**	25	4.45	7.72
	4	**16**	**4.01**	**18.75**	45	25.49	36.33
	5	4	52.53	–	**122**	**134.21**	**271.30**
MER	2	**34**	**1.42**	11.38	40	6.75	**9.89**
	3	**67**	**8.10**	**247.73**	335	133.34	–
	4	58	341.49	–	38	377.21	–
Rover		**10**	**4.07**	**1.80**	11	2.70	2.35

the given one).

These results show that, with these new decompositions, AGAR is consistently better in terms of time, memory and assumption size in almost all of the cases without alphabet refinement, and in slightly fewer cases with alphabet refinement[2]. The results are somewhat non-uniform as k increases because for each larger value of k we re-computed balanced decompositions independently of those for smaller values. This is why we even found smaller components for larger parameter, as for Chiron, Property 1, $k = 3$ vs. $k = 4$. All our results

[2]We did not count the cases when both algorithms ran out of limits.

Table 5.7: Comparison of AGAR and learning for Rule ASYM and 2-way decompositions, with alphabet refinement.

Case	k	AGAR			**Learning**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	**5**	**2.99**	**2.09**	8	3.28	3.40
Station	4	**5**	**22.79**	**12.80**	8	25.21	19.46
	5	**5**	216.07	**83.34**	8	**207.29**	188.98
Chiron,	2	10	1.30	**1.56**	**8**	**1.22**	5.17
Prop. 1	3	36	**2.44**	**10.23**	**20**	6.00	30.75
	4	160	**8.22**	252.06	**38**	41.50	**180.82**
	5	3	58.71	–	110	–	386.6
Chiron,	2	4	1.23	**0.62**	**3**	**1.06**	0.91
Prop. 2	3	8	**2.00**	3.65	**3**	2.28	**1.12**
	4	16	**5.08**	107.50	**3**	**7.30**	1.95
	5	1	81.89	–	**3**	**163.45**	**19.43**
MER	2	**5**	**1.42**	5.02	6	1.89	**1.28**
	3	9	11.09	180.13	**8**	**8.78**	**12.56**
	4	9	532.49	–	**10**	**489.51**	**1220.62**
Rover		**3**	2.62	**2.07**	4	2.46	3.30

also indicate that the benefits of alphabet refinement are more pronounced for learning.

We compared AGAR with the best learning implementation in the line work done at NASA. Our results do not transfer directly to other learning approaches for the simple reason that other implementations are different from the NASA implementation; they use symbolic BDD representations, or implement learning of general automata rather than just LTSs. Comparisons of AGAR with other learning implementations remain for future work.

Table 5.8: Original component sizes.

Case	**Gas Station**			**Chiron Prop. 1**				**Chiron Prop. 2**			
k	3	4	5	2	3	4	5	2	3	4	5
S1	1960	16464	134456	237	449	804	2030	258	482	846	2084
S2	643	1623	3447	102	1122	5559	129228	102	1122	5559	129228

Case	**MER**			**Rover**
k	2	3	4	
S1	143	6683	307623	544
S2	1270	7138	22886	41

Table 5.9: Balanced component sizes.

Case	**Gas Station**			**Chiron Prop. 1**				**Chiron Prop. 2**				**MER**
k	3	4	5	2	3	4	5	2	3	4	5	4
S1	1692	4608	31411	906	6104	1308	11157	168	4240	4156	16431	10045
S2	1942	6324	32768	924	6026	1513	11748	176	4186	4142	16840	66230

5.4.4 Comparison with related work

AGAR is a variant of the well-known CEGAR (Counter Example-Guided Abstraction Refinement) [36] with the notable differences that the computed abstractions keep information only about the interface behavior of M_2 that concerns the interaction with M_1 while it abstracts away its internal behavior, and that the counterexamples used for the refinement of M_2's abstractions are obtained in an assume-guarantee style by model checking the other component, M_1.

CEGAR has been used before in compositional reasoning in [23]). In that work, a conservative abstraction of every component is constructed and then all the resulting abstractions are composed and checked. If the check passes, the verification concludes successfully, otherwise the resulting abstract counterexample is analyzed on every abstraction that is refined if needed.

Table 5.10: Comparison of AGAR and learning for balanced decompositions without alphabet refinement.

Case	k	AGAR			**Learning**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	**10**	**3.35**	**3.36**	294	367.13	–
Station	4	269	174.03	–	433	188.94	–
	5	**7**	**47.91**	**184.64**	113	82.59	–
Chiron,	2	**41**	**2.45**	**5.46**	140	118.59	395.56
Prop. 1	3	**261**	**81.24**	**710.1**	391	134.57	–
	4	**54**	**7.11**	**37.91**	354	383.93	–
	5	402	73.74	–	112	90.22	–
Chiron,	2	**2**	**0.98**	**0.37**	40	5.21	8.30
Prop. 2	3	**88**	**15.45**	**102.93**	184	284.83	–
	4	**2**	**5.60**	**2.65**	408	222.54	–
	5	**79**	**44.16**	405.03	179	104.25	–
MER	4	9	27.62	–	311	104.72	–

The work does not use assume-guarantee reasoning, it does not address the reduction of the interface alphabets and it has not been compared with learning-based techniques.

A comparison of learning and CEGAR-based techniques has been performed in [15] but for a different problem: the 'interface synthesis' for a single component whose environment is unknown. In our context, this would mean generating an assumption that passes *Premise 1*, in the absence of a second component against which to check *Premise 2*. The interface being synthesized by the CEGAR-based algorithm in [15] is built as an abstraction of M_1. The work does not apply reduction to interface alphabets, nor does it address the verification of the generated interfaces against other components, *i.e.*, completing the assume-guarantee reasoning.

Table 5.11: Comparison of AGAR and learning for balanced decompositions with alphabet refinement.

Case	k	AGAR			**Learning**		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas	3	**5**	**2.16**	**3.06**	59	11.14	81.19
Station	4	10	15.57	191.96	**5**	**9.25**	**4.73**
	5	2	47.48	–	**15**	**52.41**	**71.29**
Chiron,	2	**9**	**1.91**	**3.89**	17	2.73	13.09
Prop. 1	3	**79**	**39.94**	**663.53**	217	36.12	–
	4	**45**	**9.55**	**121.66**	586	213.78	–
	5	**33**	**19.66**	**157.35**	46	30.05	686.37
Chiron,	2	**2**	**1.02**	**0.49**	3	1.04	0.91
Prop. 2	3	46	41.40	115.77	**3**	**5.97**	**2.26**
	4	**2**	**6.14**	11.90	20	9.33	**7.44**
	5	42	42.04	430.47	**3**	**21.94**	**7.00**
MER	4	2	27.60	–	**10**	**65.42**	**35.78**

5.5 Conclusions and Future Work

We have introduced a novel technique for automatic and incremental refinement of interface alphabets in compositional model checking. Our approach extends an existing framework for learning assumption automata in assume-guarantee reasoning. The extension consists of using interface alphabets smaller than the ones previously used in learning, and using counterexamples obtained from model checking the components to add actions to these alphabets as needed. We have studied the properties of the new learning algorithm and have experimented with various refinement heuristics. Our experiments show improvement with respect to previous learning approaches in terms of the sizes of resulting assumptions, and memory and time consumption, and with respect to non-compositional model checking, as the sizes of the checked models increase.

We have also introduced an assume-guarantee abstraction-refinement technique (AGAR) as an alternative to learning-based approaches. AGAR constructs assumptions as abstractions of M_2 and thus satisfies *Premise 2* of Rule ASYM by construction. It composes the abstraction with M_1 and checks the given property. If the property fails, it uses the counterexample to refine the abstraction and repeat the verification. Our preliminary results clearly indicate that AGAR is a feasible alternative to current approaches.

In future work we will address further algorithmic optimizations. Currently, after one alphabet refinement stage, we restart the learning or the abstraction process from scratch. The property formulated in Proposition 4 in Section 5.3.3 facilitates reuse of query answers obtained during learning. A query asks whether a trace projected on the current assumption alphabet leads to error on $M_1 \parallel \varphi_{err}$. If the answer is 'no', by Proposition 4 the same trace will not lead to error when the alphabet is refined. Thus, we could cache these query answers. Another feasible direction is to reuse the learning table as described in [91]. Similar optimizations are possible for AGAR. We also plan to use multiple counterexamples for refinement. This may enable faster discovery of relevant interface actions and smaller alphabets. Finally, we plan to perform more experiments to fully evaluate our techniques.

We can also extend AGAR with the following rule (for reasoning about n components).

$$\frac{\begin{array}{ll} \textit{(Premise 1)} & \langle A_1 \rangle\ M_1\ \langle \varphi \rangle \\ \textit{(Premise 2)} & \langle A_2 \rangle\ M_2\ \langle A_1 \rangle \\ \quad \ldots & \\ \textit{(Premise n)} & \langle \mathsf{true} \rangle\ M_n\ \langle A_{n-1} \rangle \end{array}}{\langle \mathsf{true} \rangle\ M_1 \parallel \ldots \parallel M_n\ \langle \varphi \rangle} \tag{5.2}$$

Learning with this rule and alphabet refinement overcomes the intermediate state explosion related to two-way decompositions (*i.e.*, when components are larger than the entire system) and demonstrates better scalability of compositional vs. non-compositional verification which we believe to be the ultimate test of any compositional technique. We expect to achieve similar results for AGAR.

The implementation of AGAR for Rule 5.2 involves the creation of $n-1$ instances AR_i of our abstraction-refinement code for computing each A_i as an abstraction of $M_{i+1} \parallel A_{i+1}$, except for A_{n-1} which abstracts M_n . Counterexamples obtained from *(Premise 1)* are used to refine the intermediate abstractions $A_1, \ldots, A_{n-1}$. When A_i is refined, all the abstractions $A_1, \ldots, A_{i-1}$ are refined as well to eliminate the spurious trace.

We also plan to explore extensions of alphabet refinement and AGAR to liveness properties, in a way similar to CEGAR with the analysis of lasso-shaped counterexamples [36]. Learning with L* and without alphabet refinement has been extended to liveness properties in [45].

Chapter 6

Conclusions and Future Work

We have introduced new approximation and iterative refinement techniques for several problems arising in model checking and have demonstrated that they make these, otherwise computationally hard problems, tractable on practical cases.

6.1 Summary of Contributions

For vacuity detection and query solving, we have formulated a general approximation framework that gives sufficient conditions to obtain simpler lattices of solutions, and have instantiated this framework to obtain specific approximation algorithms for the two problems. We have implemented our algorithms and evaluated our implementation, showing the benefit of our approximations on a number of cases. We have also described iterative refinement techniques that consider incrementally larger lattices to compute the approximations gradually.

6.1.1 Vacuity detection

In vacuity detection, the approximation consists of finding subformulas that are vacuous in a formula, independently of each other. The approximation is with respect to the problem of finding subformulas that are mutually or simultaneously vacuous. By approximation we lose

the mutuality of vacuity. But the mutually vacuous subformulas are among the sets of independently vacuous formulas. In this sense, we computed over-approximations of mutually vacuous sets of subformulas. Iterative refinement runs our algorithm repeatedly, at each iteration, on the subformulas at the same level of the parse tree of the formula, in a top-to-bottom, breadth-first traversal of this tree. It stops exploring subtrees of subformulas found vacuous. Thus, we find the largest vacuous subformulas.

6.1.2 Query solving

For query solving, we identified a class of problems that require query solutions to be single states of a model. In general, query solving finds solutions that represent sets of states (and it is not true that individual states in a set that is a solution are solutions themselves). Finding state solutions is an exponentially easier problem than general query solving. We prescribed a symbolic algorithm for solving it, by approximating the lattice used in a general symbolic algorithm. The approximation selects from each possible set of solutions only those that are single states. Thus the intractable lattice of all query answers is collapsed to a lattice that leads to an efficient implementation. We also described an iterative refinement scheme which asks the queries about more and more atomic propositions, using solutions obtained in one iteration to restrict the query asked in a next iteration.

We have presented experimental evaluations which demonstrate that our approximation algorithms for both vacuity detection and query solving perform better than naive algorithms.

6.1.3 Assumption generation

For assumption generation, we introduced two new iterative refinement techniques. One is designed to refine the alphabet of the assumptions generated with learning-based techniques. The refinement process starts from an initial alphabet containing actions referred to in the property to be verified, which are usually few. Fixing the current alphabet, learning proceeds as

usual with that alphabet, but it may find counterexamples due to the alphabet being insufficient to conclude the verification. In that case, by an extended counterexample analysis, we find actions to be added to the alphabet, and restart learning with the new alphabet. Furthermore, we introduced an abstraction refinement technique that replaced the learning algorithm. Our new algorithm constructs abstractions that satisfy one of the premises of the assume-guarantee rule by construction, and it only remains to verify another premise. If the verification fails, we use the counterexamples to refine the abstraction. We have performed an extensive experimental evaluation of our new refinement techniques which shows that they improve significantly upon previous learning-based assumption generation.

6.2 Future Work

Future work needs to address first some extensions of our algorithms and their implementations. For vacuity detection, the current implementation does not generate counterexamples when properties fail, nor does it provide any kind of witness in cases of non-vacuity. For query solving, we need to implement an interface that allows more applications, such as XML queries. The iterative refinement for both vacuity detection and query solving remains to be implemented and evaluated. For assumption generation, both alphabet refinement and abstraction refinement work only for safety properties. They need to be extended to liveness properties. Alphabet refinement is currently model-independent: it discovers actions by simply comparing traces. We could investigate slicing techniques or use multiple counterexamples to discover the actions on which the satisfaction of a property depends. Abstraction refinement should also be extended to use an n-component rule. More extensive evaluation is needed to study the performance of all of our algorithms. For vacuity detection and query solving, we need to find more cases where our approximations are useful, or find interesting cases where our approximations do not apply and we need to find other approximations. Our abstraction refinement needs to be compared to non-compositional verification.

Apart from the technical improvement of our techniques, we need to explore other ways to address the complexity of the problems we addressed here. We pointed out that current definitions of vacuity are not fully satisfying for practical uses. A good direction to follow in this sense is to define vacuity so that it is both easier to check and useful to users. The same applies to query solving. We envision most future work in this area to revolve around the needs of the users rather than mathematical generalizations. For assumption generation and compositional verification in general, the main challenge remains to show its benefits in comparison to non-compositional verification. Here also we believe that future improvements should be guided by cases coming from practice. Our work so far has considered cases that are practical, but have very abstract models, with not much information to be exploited when performing refinement. With richer models, coming from software programs, more complex dependency analysis is possible, to guide the refinement steps.

Bibliography

[1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. “Synthesis of interface specifications for Java classes”. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109. ACM, 2005.

[2] R. Alur, P. Madhusudan, and Wonhong Nam. “Symbolic Compositional Verification by Learning Assumptions”. In *Proceedings of the 17th International Conference on Computer Aided Verification, , CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005.

[3] B. Aminof, T. Ball, and O. Kupferman. ”Reasoning About Systems with Transition Fairness”. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005*, volume 3452 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2005.

[4] D. Angluin. “Learning Regular Sets from Queries and Counterexamples”. *Information and Computation*, 75(2):87–106, November 1987.

[5] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. “Enhanced Vacuity Detection in Linear Temporal Logic ”. In *Proceedings of the 15th International Conference on Computer Aided Verification, CAV 2003, Boulder, CO, USA,*

July 8-12, 2003, volume 2725 of *Lecture Notes in Computer Science*, pages 368–380. Springer, 2003.

[6] T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.

[7] T. Ball and S. Rajamani. "Bebop: A Symbolic Model Checker for Boolean Programs". In *Proceedings of the 7th International SPIN Workshop, SPIN Model Checking and Software Verification, SPIN 2000, Stanford, CA, USA, August 30 - September 1, 2000*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

[8] T. Ball and S. Rajamani. "The SLAM Toolkit". In *Proceedings of the 13th International Conference on Computer-Aided Verification, CAV 2001, Paris, France, July 18-22, 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[9] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. "Proof Rules for Automated Compositional Verification through Learning". In *Proceedings of the 2nd Workshop on Specification and Verification of Component-Based Systems, SAVCBS 2003, Helsinki, Finland, September 1–2, 2003*, pages 14–21, 2003.

[10] D. Beatty and R. Bryant. "Formally Verifying a Microprocessor Using a Simulation Methodogoly". In *Proceedings of the 31st Conference on Design Automation, DAC 1994, San Diego, California, USA, June 6-10, 1994*, pages 596–602. ACM Press, 1994.

[11] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. "Efficient Detection of Vacuity in ACTL Formulas". In *Proceedings of the 9th International Conference on Computer-Aided Veri-*

fication, CAV 1997, Haifa, Israel, June 22-25, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 1997.

[12] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. "Efficient Detection of Vacuity in Temporal Model Checking". *Formal Methods in System Design*, 18(2):141–163, March 2001.

[13] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. "Model Checking at IBM". *Formal Methods in System Design*, 22:101–108, 2003.

[14] S. Ben-David, D. Fisman, and S. Ruah. "Temporal Antecedent Failure: Refining Vacuity". In *Proceedings of the 18th International Conference on Concurrency Theory, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 492–506. Springer, 2007.

[15] D. Beyer, T. A. Henzinger, and V. Singh. "Algorithms for Interface Synthesis". In *Proceedings of the 19th Internation Conference on Computer Aided Verification, CAV 2007, Berlin, Germany, July 3-7, 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2007.

[16] M. Gheorghiu Bobaru, C. S. Pasareanu, and D. Giannakopoulou. "Automated Assume-Guarantee Reasoning by Abstraction Refinement". In *Proceedings of the 20th International Conference on Computer Aided Verification, CAV 2008, Princeton, NJ, USA, July 7-14, 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2008.

[17] G. Bruns and P. Godefroid. "Temporal Logic Query-Checking". In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, LICS 2001, Boston, MA, USA, 16–19 June, 2001*, pages 409–417. IEEE Computer Society, 2001.

[18] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation.". *Transactions on Computers*, 8(C-35):677–691, 1986.

[19] J.R. Burch, E.M. Clarke, K.L. McMillan, D.J. Dill, and L.J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the 5th Annual Symposium on Logic in Computer Science, LICS 1990, Philadelphia, PA, USA, June, 1990*, pages 428–439. IEEE Computer Society, 1990.

[20] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi. "Regular Vacuity". In *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005*, volume 3725 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2005.

[21] N. Chabrier-Rivier, M. Chiaverini, V. Danos, F. Fages, and V. Schachter. "Modeling and Querying Biomolecular Interaction Networks". *Theoretical Computer Science*, 325(1):25–44, 2004.

[22] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. "Automated Assume-Guarantee Reasoning for Simulation Conformance". In *Proceedings of the 17th International Conference on Computer Aided Verification, , CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 534–547. Springer, 2005.

[23] S. Chaki, J. Ouaknine, K. Yorav, and E. Clarke. "Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach". *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.

[24] S. Chaki and O. Strichman. "Optimized L*-based Assume-Guarantee Reasoning". In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2007.

[25] W. Chan. “Temporal-Logic Queries”. In *Proceedings of the 12th International Conference on Computer Aided Verification, CAV 2000, Chicago, IL, USA, July 15-19, 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463. Springer, 2000.

[26] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. “Multi-Valued Symbolic Model-Checking”. *ACM Transactions on Software Engineering and Methodology*, 12(4):1–38, October 2003.

[27] M. Chechik, M. Gheorghiu, and A. Gurfinkel. “Finding Environment Guarantees”. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007*, volume 4422 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2007.

[28] M. Chechik and A. Gurfinkel. “TLQSolver: A Temporal Logic Query Checker”. In *Proceedings of the 15th International Conference on Computer Aided Verification, CAV 2003, Boulder, CO, USA, July 8-12, 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 210–214. Springer, 2003.

[29] S.C. Cheung and J. Kramer. “Checking Subsystem Safety Properties in Compositional Reachability Analysis”. In *Proceedings of the 18th International Conference on Software Engineering, ICSE 1996, March 25-29, 1996, Berlin, Germany*, pages 144–154. IEEE Computer Society Press, 1996.

[30] S.C. Cheung and J. Kramer. “Checking Safety Properties Using Compositional Reachability Analysis”. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, 1999.

[31] H. Chockler and O. Strichman. “Easier and More Informative Vacuity Checks”. In *Rpoceedings of the 5th ACM & IEEE International Conference on Formal Methods and*

Models for Co-Design, MEMOCODE 2007, May 30 - June 1st, Nice, France, pages 189–198. IEEE, 2007.

[32] A. Cimatti, E.M. Clarke, E. Giunchilia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NUSMV Version 2: An Open Source Tool for Symbolic Model Checking". In *Proceedings of the 14th Internationa Conference on Computer Aided Verification, CAV 2002, Copenhagen, Denmark, July 27-31, 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[33] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement". In *Proceedings of the 12th International Conference on Computer Aided Verification CAV 2000, Chicago, IL, USA, July 15-19, 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[34] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Progress on the State Explosion Problem in Model Checking". In R. Wilhelm, editor, *Informatics. 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001.

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

[36] E. Clarke and H. Schlingloff. "Model Checking". In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2000.

[37] E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, LICS 1989, Pacific Grove, CA, USA, June, 1989*, pages 464–475. IEEE Computer Society, 1989.

[38] E.M. Clarke and E.A. Emerson. "Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic". In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[39] E.M. Clarke, E.A. Emerson, and A.P. Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[40] E.M. Clarke, D.E. Long, and K.L. McMillan. “A Language for Compositional Specification and Verification of Finite State Hardware Controllers”. *Proceedings of the IEEE*, 79(9):1283–1292, 1991.

[41] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. “Breaking Up is Hard to Do: An Investigation of Decomposition for Assume-Guarantee Reasoning”. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 97–108. ACM Press, 2006.

[42] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. “Learning Assumptions for Compositional Verification”. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[43] Y. Dong, B. Sarna-Starosta, C.R. Ramakrishnan, and S. A. Smolka. “Vacuity Checking in the Modal μ-calculus”. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2002.

[44] C. Espinosa-Soto, P. Padilla-Longoria, and E. R. Alvarez-Buylla. “A Gene Regulatory Network Model for Cell-Fate Determination during Arabidopsis thaliana Flower Development That Is Robust and Recovers Experimental Gene Expression Profiles”. *The Plant Cell*, 16:2923–2939, 2004.

[45] A. Farzan, Y.-F. Chen, E.M. Clarke, Y.-K. Tsay, and B.-Y. Wang. “Extending Automated Compositional Verification to the Full Class of Omega-Regular Languages”. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2008.

[46] L. Fix. “Fifteen Years of Formal Property Verification in Intel”. In O. Grumberg and H. Veith, editors, *“25 Years of Model Checking: History, Achievements, Perspectives”*, volume 5000 of *Lecture Notes in Computer Science*, pages 139–144. Springer, 2008.

[47] C. Flanagan, S. N. Freund, and S. Qadeer. “Thread-Modular Verification for Shared-Memory Programs”. In *Proceedings of the 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

[48] C. Flanagan and S. Qadeer. “Thread-Modular Model Checking”. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software, SPIN 2003, Portland, OR, USA, May 9-10, 2003*, pages 213–224, 2003.

[49] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. “Refining Interface Alphabets for Compositional Verification”. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 292–307, 2007.

[50] M. Gheorghiu and A. Gurfinkel. “Tlq: A Query Solver for States“. In *Tools and Posters Session at the 14th International Symposium on Formal Methods, FM 2006, Hamilton,*

Canada, August 21-27, 2006, 2006.

[51] M. Gheorghiu and A. Gurfinkel. "VaqUoT: A Tool for Vacuity Detection". In *Tools and Posters Session at the 14th International Symposium on Formal Methods, FM 2006, Hamilton, Canada, August 21-27, 2006*, 2006.

[52] M. Gheorghiu, A. Gurfinkel, and M. Chechik. "Finding State Solutions to Temporal Logic Queries". In *Proceedings of the 6th International Conference on Integrated Formal Methods, IFM 2007, Oxford, UK, July 2-5, 2007*, volume 273-292 of *Lecture Notes in Computer Science*, page 4591. Springer, 2007.

[53] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. "Assumption Generation for Software Component Verification". In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE 2002, Edinburgh, Scotland, UK, 23-27 September 2002*, pages 3–12. IEEE Computer Society, 2002.

[54] G. Gottlob and C. Koch. "Monadic Queries over Tree-Structures Data". In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science, LICS 2002, 22-25 July 2002, Copenhagen, Denmark*, pages 189–202. IEEE Computer Society, 2002.

[55] O. Grumberg and D.E. Long. "Model Checking and Modular Verification". In *Proceedings of the 2nd International Conference on Concurrency Theory, CONCUR 1991, Amsterdam, The Netherlands, August 26-29, 1991*, volume 527 of *Lecture notes in Computer Science*. Springer, 1991.

[56] A. Gupta, K. L. McMillan, and Z. Fu. "Automated Assumption Generation for Compositional Verification". In *Proceedings of the 19th Internation Conference on Computer Aided Verification, CAV 2007, Berlin, Germany, July 3-7, 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2007.

[57] A. Gurfinkel and M. Chechik. "Multi-Valued Model-Checking via Classical Model-Checking". In *Proceedings of the 14th International Conference on Concurrency Theory,*

CONCUR 2003, Marseille, France, September 3-5, 2003, volume 2761 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2003.

[58] A. Gurfinkel and M. Chechik. "Extending Extended Vacuity". In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004*, volume 3312 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 2004.

[59] A. Gurfinkel and M. Chechik. "How Vacuous Is Vacuous?". In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, 29 - April 2, 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 451–466. Springer, 2004.

[60] A. Gurfinkel, M. Chechik, and B. Devereux. "Temporal Logic Query Checking: A Tool for Model Exploration". *IEEE Transactions on Software Engineering*, 29(10):898–914, October 2003.

[61] T. A. Henzinger, R. Jhala, and R. Majumdar. "Permissive Interfaces". In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 31–40. ACM, 2005.

[62] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. "You Assume, We Guarantee: Methodology and Case Studies". In *Proceedings of the 10th International Conference on Computer-aided Verification, CAV 1998, Vancouver, BC, Canada, June 28 - July 2, 1998*, number 1427 in Lecture Notes in Computer Science, pages 440–451. Springer, 1998.

[63] S. Hornus and Ph. Schnoebelen. "On Solving Temporal Logic Queries". In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology,*

AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, volume 2422 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2002.

[64] R. Jeffords and C. Heitmeyer. "Automatic Generation of State Invariants from Requirements Specifications". In *Proceedings of the 6th International Symposium on Foundations of Software Engineering, FSE 1998, Lake Buena Vista, FL, USA, November 3-5, 1998*, pages 56–69. ACM, 1998.

[65] C. B. Jones. "Specification and Design of (Parallel) Programs". In *Proceedings of the IFIP 9th World Congress on Information Processing, Paris, France, September 19-23, 1983*, pages 321–332. IFIP: North Holland, 1983.

[66] B. Konikowska and W. Penczek. "Reducing Model Checking from Multi-Valued CTL* to CTL*". In *Proceedings of the 13 International Conference on Concurrency Theory, CONCUR 2002, Brno, Czech Republic, August 20-23, 2002*, Lecture Notes in Computer Science, pages 226–239. Springer, 2002.

[67] J.-P. Krimm and L. Mounier. "Compositional State Space Generation from Lotos Programs". In *Proceedings of the 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS 1997, Enschede, The Netherlands, April 2-4, 1997*, volume 1217 of *Lecture Notes in Computer Science*, pages 239–258. Springer, 1997.

[68] O. Kupferman. "Sanity Checks in Formal Verification". In *Proceedings of the 17th International Conference on Concurrency Theory, CONCUR 2006, Bonn, Germany, August 27-30, 2006*, volume 4137 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2006.

[69] O. Kupferman and M. Vardi. "Vacuity Detection in Temporal Model Checking". In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME '99, Bad Herrenalb, Germany,*

September 27-29, 1999, volume 1703 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1999.

[70] O. Kupferman and M. Vardi. “Vacuity Detection in Temporal Model Checking”. *International Journal on Software Tools for Technology Transfer, STTT*, 4(2):224–233, February 2003.

[71] O. Kupferman, M.Y. Vardi, and P. Wolper. “An Automata-Theoretic Approach to Branching-Time Model Checking”. *Journal of the ACM*, 27(2):312–360, March 2000.

[72] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.

[73] P. Maier. “Compositional Circular Assume-Guarantee Rules Cannot Be Sound and Complete”. In *Proceedings of the 6th International Conference on Foundations of Software Science and Computational Structures, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003*, volume 2620 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2003.

[74] Z. Manna and A. Pnueli. “Verification of Concurrent Programs: A Temporal Proof System”. Technical report, Department of Computer Science, Stanford University, 1983.

[75] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[76] G. Miklau and D. Suciu. ”Containment and Equivalence for an XPath fragment”. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Madison, Wisconsin, USA, June 3-5, 2002*, pages 65–76. ACM, 2002.

[77] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[78] J. Misra and K. M. Chandy. “Proofs of Networks of Processes”. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, July 1981.

[79] W. Nam and R. Alur. “Learning-Based Symbolic Assume-Guarantee Reasoning with Automatic Decomposition”. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis, ATVA 2006, Beijing, China, October 23-26, 2006*, volume 4218 of *Lecture Notes in Computer Science*. Springer, 2006.

[80] K. Namjoshi. “An Efficiently Checkable, Proof-Based Formulation of Vacuity in Model Checking”. In *Proceedings of the 16th International Conference on Computer-Aided Verification, CAV 2004, Boston, MA, USA, July 13-17, 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 57–69. Springer, 2004.

[81] C. Pasareanu and D. Giannakopoulou. “Towards a Compositional SPIN”. In *Proceedings of the 13th International SPIN Workshop on Model Checking Software, SPIN 2006, Vienna, Austria, March 30 - April 1, 2006,*, volume 3925 of *Lecture Notes in Computer Science*, pages 234–251. Springer, 2006.

[82] C. S. Pasareanu, D. Giannakopoulou, M. Gheorghiu Bobaru, J. M. Cobleigh, and H. Barringer. “Learning to Divide and Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning”. *Formal Methods in System Design*, 32(3):175–205, 2008.

[83] A. Pnueli. “In Transition from Global to Modular Temporal Reasoning about Programs”. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer Verlag, 1985.

[84] M. Purandare and F. Somenzi. “Vacuum Cleaning CTL Formulae”. In *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002, Copenhagen, Denmark, July 27-31, 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 485–499. Springer, 2002.

[85] J.P. Quielle and J. Sifakis. “Specification and Verification of Concurrent Systems in CESAR”. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[86] T. W. Reps, S. Horwitz, and M. Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, San Francisco, California, January 23-25, 1995*, pages 49–61. ACM Press, 1995.

[87] R. L. Rivest and R. E. Shapire. “Inference of Finite Automata Using Homing Sequences”. *Information and Computation*, 103(2):299–347, April 1993.

[88] M. Samer and H. Veith. “Validity of CTL Queries Revisited”. In *Proceedings of the 17th International Workshop on Computer Science Logic, CSL 2003, Vienna, Austria, August 25-30, 2003*, volume 2803 of *Lecture Notes in Computer Science*, pages 470–483, Vienna, Austria, August 2003. Springer.

[89] M. Samer and H. Veith. “A Syntactic Characterization of Distributive LTL Queries”. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming, ICALP 2004, Turku, Finland, July 12-16, 2004*, volume 3142 of *Lecture Notes in Computer Science*, pages 1099–1110, 2004.

[90] M. Samer and H. Veith. “Parameterized Vacuity”. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004*, volume 3312 of *Lecture Notes in Computer Science*, pages 322–336, Austin, Texas, November 2004. Springer.

[91] N. Sharygina, S. Chaki, E. Clarke, and N. Sinha. “Dynamic Component Substitutability Analysis”. In *Proceedings of the International Symposium of Formal Methods Europe, FM 2005, Newcastle, UK, July 18-22, 2005*, volume 3582 of *Lecture Notes in Computer Science*, pages 512–528. Springer, 2005.

[92] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik. “Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC”. In *Proceedings of the 7th International Conference on Formal Methods in Computer Aided Design, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007*, pages 3–12. IEEE Computer Society, 2007.

[93] N. Sinha and E. M. Clarke. “SAT-Based Compositional Verification Using Lazy Learning”. In *Proceedings of the 19th Internation Conference on Computer Aided Verification, CAV 2007, Berlin, Germany, July 3-7, 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2007.

[94] F. Somenzi. “Binary Decision Diagrams”. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[95] M. Y. Vardi and P. Wolper. “An Automata-Theoretic Approach to Automatic Program Verification”. In *Proceedings of the Symposium on Logic in Computer Science, LICS 1986, Cambridge, Massachusetts, June 16-18, 1986*, pages 322–331, Cambridge MA, 1986. IEEE Computer Society.

[96] D. Zhang and Rance Cleaveland. “Efficient Temporal-Logic Query Checking for Presburger Systems”. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, Long Beach, CA, USA, November 7-11, 2005*, pages 24–33. ACM, 2005.