UNIVERSITY OF CALIFORNIA
SANTA CRUZ

INTERVAL-BASED ABSTRACTION REFINEMENT

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

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Pritam Roy

December 2009

The Dissertation of Pritam Roy
is approved:

Professor Luca de Alfaro, Chair

Professor Natarajan Shankar

Professor Cormac Flanagan

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Interval-based Abstraction Refinement

by

Pritam Roy

The prevailing trend in software and system engineering is towards *component-based design.* In this approach, a number of small design units called *components* compose a complex design. Components are typically open systems that have inputs provided by other components and provide inputs to other components. Designers face a number of design issues to build a complex design from these components. A designed system, expected to perform a set of tasks following its specification, may not behave properly due to the following reasons. Firstly, one or more components may contain bugs and behave in an undesirable fashion. Secondly, components make assumptions on their environment, and expect that the actual environment will meet these assumptions. A number of bug-free components may not work together if their input assumptions are violated. Hence, verification of a complex design can be reduced to the verification of the components and communication among them.

The interaction between components in a design can be modeled via games, and a large body of literature on design and verification shows how games can be used to analyze component compatibility and system correctness. However, while games provide an appropriate, mathematical model for interaction, solving the games is often impractical with current algorithms, due to the large state-space of games representing realistic components, together with the inherent complexity of game-solving techniques. In this thesis, we propose algorithms for the efficient analysis of games with large state spaces.

We present two novel algorithm families in the thesis: (1) Game-based Three Valued

Abstraction (GTVA) for two-player games/transition systems, and (2) Magnifying Lens Abstraction (MLA) for Markov Decision Processes (MDPs). GTVA evaluates the property on the abstract game in three-valued fashion (*yes*, *no*, *maybe*) and refines the abstraction by adding more details to the *maybe* abstract states. However, other approaches construct abstract models; thus verification becomes extremely expensive. We explain how to achieve efficient enumerative and symbolic implementations of the algorithm. MLA partitions the state-space of MDP into regions and then computes upper and lower bounds on the regions, rather than on the concrete states. MLA iterates over the regions to evaluate these limits and considers the concrete states of each region in turn, as if one were moving a magnifying lens across the abstraction and viewing the concrete states corresponding to the current region. The algorithm refines the regions in an adaptive fashion, splitting regions where we need more details, until the difference between the bounds is smaller than a user-given accuracy. We also provide a symbolic version of algorithm MLA (SMLA).

We have implemented the proposed algorithms, and we have applied them to real-life applications, including planning, protocol verification, and interface synthesis for software libraries. The symbolic three-valued algorithms for reachability, safety, compatibility, and refinement properties have been implemented in the tool TICC; case-studies illustrate the accuracy and efficiency benefits of the GTVA algorithms over other approaches. We have implemented the symbolic version of MLA in the tool PRISM. The experimental results demonstrate that MLA can provide accurate answers, with savings in the memory requirements. These algorithms promise to make the analysis of realistic component-based designs possible by pushing the limits of the size of games that can be solved.

To my mother and father,

Acknowledgments

I am extremely lucky to get the guidance of Luca de Alfaro in this long journey. Luca has motivated me to raise the level of my work - both in research and development. I hope I can continue to research following his illustrious footsteps. Moreover, I would like to thank my lab-mates in Design and Verification Lab (DVLAB) at University of California Santa Cruz. I am truly fortunate to have social as well as academic interactions with the lab-mates. I want to thank my mentors Margus Veanes, Chao Wang and Natarajan Shankar for making my summers extremely busy and productive. Some parts of the thesis are results of those summer efforts and further collaborations. I also thank my co-authors David Parker and Gethin Norman for the first implementation of symbolic MLA algorithm into the tool PRISM. I want to thank the dissertation committee members Cormac Flanagan, Natarajan Shankar and Luca de Alfaro for their patience and valuable inputs on the dissertation. My friends have always encouraged me and stood by my side during the hard times. I should also thank my wife Debashri for patiently listening to me when everything else went wrong. Last but not the least, I love to thank my dad, mother, brother and sister to be what I am today.

Chapter 1

Introduction

1.1 Component Based Design

The prevailing trend in software and system engineering is towards *component-based design*. In this approach, a number of small design-units called *components* compose a complex design. Hence the verification of a complex design can be reduced to the verification of its components and the interactions among them. By design, components should work as parts of larger systems. However, components assume constraints over their environment, and the actual environment should meet these constraints. A component is typically an open system which has inputs provided by others components and which provides inputs to other components.

1.1.1 Design Issues

The designers face a number of design issues to build a complex design. A designed system, expected to perform a set of tasks following its specification, may not work due to the following reasons.

Design Bugs in Components : One or more components may contain bugs (design flaws) and behave in an undesirable manner. The bugs in the design cost enormous amount of loss in terms of money, time, and other valuable resources. Hence It is imperative to verify the system-design with respect to its specification by either by static analyses, such as model-checking, or dynamic analyses, such as testing.

Incompatibilty of Components : Since components make assumptions on their environment, a set of bug-free components may not work together when output of one component violates another component's input assumptions. Hence it is necessary to compute the compatibility and refinement of the components. Verification of a complex design is equivalent to verification of the components and communication among them.

1.1.2 Games as System Models

The interaction between components in a design can be modeled via games, and a large body of literature on design and verification shows how games can be used to analyze component compatibility and system correctness. Game-based models can be used to address both aspects of the problem. For example, the interface theories reason about the communication, refinement, and composition of the components using game-based models. On the other hand, game models of the open systems provide elegant formal semantics to these components. Hence the solution of a complex-design verification problem can be reduced to efficient solution of games.

1.1.3 Classification of Games

Games are typically classified with respect to the number of players, and each has different behavior and applications. For example, the internal structure of some software and hardware systems determines the behavior of those systems. *Transition systems* (*1-player games*)

can model this class of systems. A transition system has applications in verification of hardware and software systems. Similarly, *two-player games* model reactive systems. The internal structure and inputs from the environment determine the behavior of reactive systems. Two-player game models have applications in supervisory control [90], sequential hardware synthesis and program synthesis [28, 22, 87], modular verification [5, 8, 46], receptiveness checking [6, 55], interface compatibility checking [42], and schedulability analysis [2]. Systems with probabilistic and non-deterministic behavior can be modeled as Markov Decision Processes (MDPs). MDPs, also known as 1.5 player games, are widely used in probabilistic verification, planning, optimal control, network analysis, and performance analysis.

1.2 Solution of Games

Winning Objectives : A set of specification properties formally define the desirable behavior of the given system. In the game semantics, the specification properties are also known as *winning objectives.* Besides the number of players, the winning objectives can also determine the class of the games. Games with *qualitative objectives* such as reachability, and safety have been widely studied in literature. A reachability objective specifies that the behavior of a system should eventually reach a set of target states. A safety objective specifies that the behavior of the system should never leave a safe subset of states.

Model Checking : Games provide an appropriate, mathematical model for interaction. A game model M with winning objective ϕ can be solved using model checking algorithms. The result of model checking is Boolean for the qualitative systems and a real value $v \in \{0, 1\}$ for the probabilistic systems. The game models can be represented as graphs where the nodes and edges represent state-space and the transition-relation of the system. The model-checking algorithms

can be reduced to graph traversal problems. For example, the model-checking algorithm of a transition system with reachability objective $\Diamond T$ can be reduced to a breadth-first-search starting from the goal set T.

State-space Explosion : Solving the games is often impractical with current algorithms. The main reasons are (1) the large state-space of games representing realistic components, and (2) inherent complexity of game-solving techniques. One challenge for model checking and other algorithmic methods is the state-space explosion problems. The algorithms tend to take more time and space when the system becomes more complex. The capacities of the formal verification and testing tools have not scaled up with the design complexity. So the researchers face increasingly stiff challenges to verify the system-design within limited resources (time and space). Although techniques like symbolic representations, symmetry reduction and partial reduction work well; but these methods have their limitations.

Qualitative Abstraction : The main method used to handle the state-space explosion problems to the solution of games is *abstraction-refinement*. Abstraction is a technique for reduction of a system with large state-spaces (concrete model) to a system with small state-spaces (abstract model) by removing information which is not relevant to the property one would want to verify. The model checking of an abstract model takes less time and space for property verification. However, the result of the model checking over abstract model may not be accurate due to the coarse nature of the abstraction; hence the abstract model needs *refinement* by splitting some abstract states. This iterative framework (known as the *abstraction-refinement* framework) continues "model-check and refinement" steps until the one gets a precise result. In recent times, there has been much research based on this abstraction-refinement framework. Henzinger et. al. [62] have applied counter-example guided abstraction refinement (CEGAR)

[9, 13, 31] approach. These algorithms have *one-sided errors.* Shoham et.al. [96, 98] applied 3 valued abstraction refinement approach for verification. These algorithms construct abstract-models using hyper-transitions [98]; thus the construction of abstract-model and verification become extremely expensive. Unfortunately, these game algorithms either contain one-sided errors or expensive computations of abstract models.

Quantitative Abstraction: The successful application of abstraction techniques in non-probabilistic systems has spurred research into probabilistic systems [32, 67, 81, 70]. The abstraction construction in probabilistic systems is a harder problem due to the presence of probabilities. Like non-probabilistic abstraction algorithms, most of the techniques build abstract models with respect to state transitions in the concrete model (*full abstraction* methods). Most quantitative abstraction techniques are *approximate*; they apply either *simulation* or *abstract interpretation.* Like their qualitative counterpart, these algorithms also contains one-sided errors and/or expensive construction of abstract models.

Interval-based Abstraction Refinement : My dissertation presents a novel framework on interval-based abstraction-refinement. This framework covers a class of abstraction refinement algorithms and this set of algorithms (1) do not involve costly abstract model computation, (2) do not contain one-sided errors. We present two interval-based abstraction refinement algorithms: Three Valued Abstraction (TVA) and Magnifying Lens Abstraction (MLA). We have applied these scalable algorithms into several real-life applications like planning, protocol verification, interface synthesis for software libraries. Our thesis is that the

> verification problem of a complex design can be reduced to the verification of the components and communication among them, and can be solved using game-based models, and the proposed abstraction-refinement algorithms make the analysis of realistic component-based designs possible by pushing the limits of the size of games that can be solved

The remainder of the chapter provides the contributions, related works, and an overview of the organization of the thesis.

1.3 Research Contribution

The research contribution can be divided into two main parts - (1) Application of game-based models, (2) interval-based abstraction refinement frame work and applications.

1.3.1 Application of Game Models

Interface Compatibility Checking of Components: In component-based design, it is crucial to know whether interfaces of two components are *compatible*, and whether a new component can replace an existing component in an integrated system (*refinement*). Interface automata are light-weight representations of the behavior of components of the design. We have designed an extension of interface automata called *Sociable Interface* [39]. Interface theory provides a game-theoretic way of solving the compatibility and refinement problems. We developed symbolic algorithms for compatibility, and refinement properties of interfaces. We implemented the algorithms in the tool TICC[1] where user can specify and verify different properties of a design.

Software Testing: Software testing is another practical application of game models. In the literature, the researchers often view software testing as a game that the tester plays against an *implementation under test* (IUT) [3, 18]. The tester does not know the implementation; instead it knows the model behavior of the IUT. Since both the model and IUT (more precisely a wrapper around the IUT) are examples of open systems, they behave as interface automata [35, 42] and interface theories evaluate the compatibility of these two interfaces. Online testing (also called on-the-fly testing) is a testing procedure that merges test generation and test execution into a single process. Online testing can be modeled as a MDP where the tester has a goal and

the other player (the IUT) is unaware that it is playing and makes random choices following a probability distribution [18]. In online testing, the compatibility checking of IUT with the model is a necessary step. If the goal of the tester is to reach a set of states, then the work [18] solves the problem using value-iteration algorithms. If the goal of the tester is to maximize the coverage for a number of test-runs and number of steps per run, then the work [107] solves the problem using *reinforcement learning* techniques [101]. The large state-space of these problems has prompted state-grouping abstraction [107, 23].

1.3.2 Interval-based Abstraction Refinement Framework

My dissertation presents a new framework on interval-based abstraction-refinement. This framework covers a class of abstraction refinement algorithms that

1. Partition the state-space into a set of abstract states (known as *regions*),
2. Model-check the partitioned state-space and return an interval $[v^-, v^+]$ for each region,
3. Refine adaptively a set of regions where the interval is wider than user-given constant ϵ_{abs}.

These algorithms (1) do not contain one-sided errors, and (2) do not involve costly model-construction algorithms. We present two novel algorithm families in the thesis: (1) Game-based Three Valued Abstraction (GTVA) for two-player games/transition systems, and (2) Magnifying Lens Abstraction (MLA) for Markov Decision Processes (MDPs).

Game-based Three-Valued Abstraction (GTVA): We developed a novel symbolic abstraction algorithm [51] for the solution of transition systems and two-player games with reachability, or safety goals. GTVA evaluates the property on the abstract game in three-valued fashion (*yes*, *no*, *maybe*). If the computation fails to yield a certain yes/no result to the validity of the property on the initial states, our algorithm refines the abstraction by splitting *uncertain* ab-

stract states. Most three-valued approaches construct abstract models using hyper-transitions; thus the abstract model construction and verification become very expensive. Our approach does not build the abstract model explicitly; rather abstract predecessor operators (based on the abstraction function) work on the Binary Decision Diagram (BDD) based symbolic representation of the concrete game structure. We have implemented these three-valued abstraction algorithms in the tool TICC.

Magnifying Lens Abstraction: GTVA algorithms [51] motivated us to investigate a similar set of techniques for the probabilistic counterpart. We have developed a novel abstraction technique which allows the analysis of reachability and safety properties of MDPs with very large state spaces. The technique, called *magnifying-lens abstraction,* (MLA) [50] copes with the state-explosion problem by partitioning the state-space into regions and then computing upper and lower bounds on the regions, rather than on the states. To compute these bounds, MLA iterates over the regions, considering the concrete states of each region in turn, as if one were sliding a magnifying lens across the abstraction which facilitates a closer view of the concrete states. The algorithm adaptively refines the regions, using smaller regions where more detail is needed, until the difference between upper and lower bounds is smaller than a specified accuracy. The experimental results show that MLA can provide accurate answers, with savings in memory requirements. We have prototyped a python implementation to show the space-savings of the of MLA algorithm [50]. We provided a symbolic version of the MLA algorithm that combines symbolic techniques with abstraction techniques to handle the space-explosion problem better. We have implemented a symbolic version of the MLA algorithms (called SMLA [91]) in the probabilistic model checking tool PRISM.

Interfaces for Libraries: Automatic construction of a model in the model based testing (MBT) framework is a relatively new concept. Given a high-level specification of the library behavior and error conditions, an interface (function call sequence) graph can assume the role of a model for the implementation-under-test (IUT). The edges of the interface graph denote the functions and the states denote the valuations of the global variables. Given a set of functions from a library, we compute an interface graph to identify the *safe* (not leading the library to error) calls in the library. We developed and implemented symbolic abstraction-refinement based algorithms by summarizing every function in a purely modular approach. Related work by other groups does not apply purely modular approach to create the interface graphs.

1.4 Related Works

In this section we discuss the related works and compare with my contributions.

1.4.1 Interface Theories and Tools

Previous interface models, such as interface automata [42, 44] and interface modules [43, 25] were based on either actions, or variables, but not both. In sociable interfaces, however, we want to have both: actions to model synchronization, and variables to encode the global and local state of components. In this, sociable interfaces are closely related to the I/O Automata Language (IOA) of [76].However, sociable interfaces diverge from I/O Automata in several ways. Unlike I/O Automata, where every state must be receptive to every possible input event, sociable interfaces allow states to forbid some input events. By not accepting certain inputs, sociable interfaces express the assumption that the environment never generates these inputs: hence, sociable interfaces (like other interface models) model both the output behavior, and the input assumptions, of a component. This approach implies a notion of composition (based on synthe-

sizing the weakest environment assumptions that guarantee compatibility) which is not present in the I/O Automata Model.

Variable-based interface formalisms In variable-based interface formalisms, such as the formalisms of [43, 25], communication is mediated by input and output variables, and the system evolves in synchronous steps. It is well known that synchronous, variable-based models can also encode communication via actions [7]. However, this encoding prevents the modeling of many-to-one and many-to-many communication. In fact, due to the synchronous nature of the formalism, a variable can be modified at most by one module: if two modules modified it, there would be no simple way to determine its updated value. As a consequence, we cannot write modules that can accept inputs from multiple sources: every module must know precisely which other modules can provide inputs to it, so that distinct communication actions can be used. The advance knowledge of the modules involved in communication hampers module re-use.

Action-based interface formalisms Action-based interfaces, such as the models of [42, 44] , enable a natural encoding of asynchronous communication. In previous proposal, however, two interfaces could be composed only if they did not share output actions again ruling out many-to-one communication. Furthermore, previous action-based formalisms lacked a notion of global variables which are visible to all the modules of a system. Such global variables are a very powerful and versatile modeling paradigm, providing a notion of global, shared state. Mimicking global variables in purely action-based models is rather inconvenient: it requires encapsulating every global variable by a module, whose state corresponds to the value of the variable. Read and write accesses to the variable must then be translated to appropriate sequences of input and output actions, leading to cumbersome models. The asynchronous, action-based interface theories of [42, 44] are implemented as part of the Ptolemy tool-set [73]. The tool

CHIC implements synchronous, variable-based interface theories closely modeled after[24]. rich communication schemes, including exclusive, and many-to-many schemes, and differentiates the modules of TICC from other modules with more restrictive communication primitives, such as I/O Automata [76] and Reactive Modules [7].

1.4.2 Application of Game-based Algorithms in Software Testing

The basic idea of online testing has been introduced in the context of labeled transition systems using IOCO theory [20, 102, 104] and implemented in the TorX tool [103]. TGV [68] is another tool frequently used for online or on-the-fly test generation that uses ioco. Ioco theory is a formal testing theory based on labeled transition systems with input actions and output actions. Interface automata [36] are suitable for the game view [25] of online testing and provide the foundation for the conformance relation that we use. Online testing with model programs in the Spec-Explorer tool is discussed in [106]. The algorithm in [106] does not use learning, and as far as we know learning algorithms have not been considered in the context of model based testing. The relation between ioco and refinement of interface automata is briefly discussed in [106]. Specifications given by a guarded command language are used also in [89]. In Black-box testing, some work [84] has been done which uses supervised learning procedures. As far as we know, no previous work has addressed online testing with learning in the context of Model Based Testing. The main intuition behind our algorithm is similar to an anti-ant approach [75] used for test case generation form UML diagrams. From the game point of view, the online testing problem is a $1\frac{1}{2}$-player game. It is known that $1\frac{1}{2}$-player games are Markov Decision Processes [26]. The view of finite explorations of model programs for offline test case generation as negative total reward Markov decision problems with infinite horizon are studied in [19].

1.4.3 Qualitative Abstraction

Counterexample-guided abstraction refinement (CEGAR) [9, 13, 31], the most successful abstraction technique, has been applied in both hardware [31] and software [13, 63] verification. According to this technique, given a system abstraction, we check whether the abstraction satisfies the property. If the answer is affirmative, we are done. Otherwise, the check yields an *abstract counterexample,* encoding a set of "suspect" system behaviors. The abstract counterexample is then further analyzed, either yielding a concrete counterexample (a proof that the property does not hold), or yielding a refined abstraction, in which that particular abstract counterexample is no longer present. The process continues until either a concrete counterexample is found, or until the property can be shown to hold (i.e., no abstract counterexamples are left). The appeal of CEGAR lies in the fact that it is a fully automatic technique, and that the abstraction is refined on-demand, in a property-driven fashion, adding just enough detail as is necessary to perform the analysis. The CEGAR technique has been extended to games in *counterexample-guided control* [62].

In its aim of reducing the number of may-states, our technique is related to the three-valued abstraction refinement schemes proposed for CTL and transition systems in [97, 98]. We avoid the explicit construction of the tree-valued transition relation of the abstraction, relying instead on *may* and *must* versions of the controllable predecessor operators. Our approach provides precision and efficiency benefits. In fact, to retain full precision, the must-transitions of a three-valued model need to be represented as hyper-edges, rather than normal edges [98, 41, 99]; in turn, hyper-edges are computationally expensive both to derive and to represent. The may and must predecessor operators we use provide the same precision as the hyper-edges, without the associated computational penalty. For a similar reason, we show that our three-valued abstraction refinement technique is superior to the CEGAR technique of [62], in the sense that

it can prove a given property with an abstraction that never needs to be finer, and that can often be coarser. Again, the advantage is due to the fact that [62] represents player-1 moves in the abstract model via must-edges, rather than must hyper-edges. A final benefit of avoiding the explicit construction of the abstract model, relying instead on predecessor operators, is that the resulting technique is simpler to present, and simpler to implement. On the other side, we remark that the techniques of [62] extend easily to parity goals, whereas the refinement scheme we propose can be extended, but only at the price of cumbersome bookkeeping.

1.4.4 MLA

For the most part, approaches to MDP abstraction in the literature have followed a conventional route, which we call very broadly the *full abstraction* approach: an abstract model is constructed, and then analyzed on the basis of an abstract transition structure, without further reference to the concrete model. These fully abstract approaches generally rely on clustering states that are similar not only in value, but also in transition structure: in this way, every region of concrete states can be summarized via an abstract state with an associated abstract transition structure. The abstract transition structure may, or may not, be similar to the concrete one. For instance, [70] bases the abstract transition structure on games, rather than MDPs: in this fashion, player 1 can represent the choice of action of the MDP, and player 2 can represent the uncertainty about the concrete state corresponding to the abstract state. This approach enables the computation of lower and upper bounds for properties of interest, similarly to MLA. In a somewhat related spirit, but using entirely different technical means, [58] proposes to abstract Markov chains into *abstract Markov chains* whose transitions are labeled with intervals of probability, representing the uncertainty about the concrete state. Clustering states based on the similarity in their transition probabilities has also been used in [52], which proposes to find the coarsest refinement of an MDP where for each action, states

in the same region have the same probability of going to other regions. An approach for the verification of probabilistic reachability properties via abstraction has been proposed in [32]. The abstraction is built through successive refinements starting from a coarse partition based on the property. Several other approaches also, in fact, rely on constructing MDP abstractions based on simulation or abstract interpretation [67, 81, 80]; all of these approaches rely on clustering states with similar transition structure, and representing these clusters of states, and their transition structures, via compact abstract representations.

Compared with other approaches to MDP abstraction, MLA (and SMLA) has two distinctive features:

1. it clusters states based on value, rather than based on the similarity in their transition function;

2. it updates the valuation of abstract states by considering the concrete states associated with the abstract states, rather than by considering an abstract model only.

The second of the above points underlines how MLA is a semi-abstract, rather than fully abstract, approach to verification: the abstract computation still involves consideration of the concrete states, even though this is done in a way that provides space savings.

The full-abstraction approach outlined above, and the partial value-based approach followed by MLA, each have advantages. The full-abstraction result can handle unbounded, and (depending on the specific approach) even infinite state spaces. In contrast, the space savings afforded by MLA are limited to a square-root factor (a system of size n can be studied in $O(\sqrt{n})$ space), due to the need to consider the concrete states corresponding to each abstract one. Furthermore, the full-abstraction approaches typically need to construct the abstract model only once; in contrast, MLA needs to refer to concrete states (albeit not all of them at once) during the computation.

On the other hand, the ability of MLA to cluster states based on value only, disregarding differences in their transition relation, can lead to compact abstractions for systems where full abstraction provides no benefit. We will give below an example supporting this. Furthermore, in MLA the abstraction is refined dynamically, depending on the required accuracy of the analysis; there is no need to "guess" the right state partition in advance. In our experience, MLA is particularly well-suited to problems where there is a notion of *locality* in the state space, so that it makes sense to cluster states based on variable values — even though their transition relations may not be similar. Many planning and control problems are of this type. MLA instead is not as well-suited to problems where clustering states based on variable values is less effective. Approaches based on predicate abstraction could lend the MLA approach more generality.

In MLA, as long as the value of the property of interest is similar in states in the same interval, abstraction is possible and useful. Indeed, experimentally we noticed that SMLA performs well in many problems with integer-valued state variables, where the properties vary gradually with the value of the state variables. Problems in planning, inventory control, and similar often belong to this category. On the other hand, when it is possible to use symmetry or structural knowledge of an example, and aggregate states of similar transition relation, approaches such as [32, 70, 71] yield superior results.

MLA is reminiscent to methods that represent value functions via ADDs or MTBDDs [30, 11] with an approximation factor used to merge leaves. The similarity, however, is superficial: MLA leads to far more precise results in the analysis; we discuss this in the conclusions, where the appropriate notation will be available.

MLA is also loosely reminiscent of *adaptive mesh refinement* (AMR) methods used in the solution of partial differential equations [14]. There are, however, two important differences between MLA and AMR. In AMR, separate lower and upper bounds are not kept. AMR methods perform computation at the finest mesh sizes only where needed. In MLA, due to the discrete

nature of MDPs, we have no way of computing over a "coarse mesh" only: to update valuations over a region, we need to "magnify" the region to its individual states. Thus, MLA is forced to consider the individual states over the whole system, and it summarizes and returns the results in terms of lower and upper bounds, which are well-suited to answering verification questions.

1.5 Organization of the Thesis

I have organized the thesis into four main parts: Part I (Chapter 2) provides preliminary definitions and algorithms to understand the rest of the chapters. Part II (Chapters 3-4) provides the application of game models in interface compatibility checking and online-testing algorithms. Part III(Chapters 5-6) provides game-based three-valued abstraction-refinement (GTVA) algorithms and applications. Part IV (Chapters 7-8) provides Magnifying Lens Abstraction (MLA) algorithms and their symbolic counterparts SMLA. Finally, Chapter 9 concludes with the summary and future work.

Part I

Background

Chapter 2

Background

2.1 Preliminary Definitions

For a countable set S, a *probability distribution* on S is a function $p : S \mapsto [0,1]$ such that $\sum_{s \in S} p(s) = 1$; we denote the set of probability distributions on S by $D(S)$. A *valuation* over a set S is a function $v : S \mapsto \mathbb{R}$ associating a real number $v(s)$ with every $s \in S$. For valuations v, u over S, we define operators and inequalities in point-wise fashion: for instance, we define $v + u$ by $(v + u)(s) = v(s) + u(s)$ for all $s \in S$, and we write $v \leq u$ if $v(s) \leq u(s)$ at all $s \in S$. For $x \in \mathbb{R}$, we denote by $\mathbf{x}$ the valuation with constant value x; for $T \subseteq S$, we indicate by $[T]$ the valuation having value 1 in T and 0 elsewhere. For two valuations v, u on S, we define $||v - u|| = \sup_{s \in S} |v(s) - u(s)|$.

A *partition* of a set S is a set $R \subseteq 2^S$, such that $\bigcup\{s | s \in R\} = S$, and such that for all $r, r' \in R$, if $r \neq r'$ then $r \cap r' = \emptyset$. We will define abstractions of the state space S simply via partitions of S. For $s \in S$ and a partition R of S, we denote by $[s]_R$ the element $r \in R$ with $s \in r$. We say that a partition R' is *finer* than a partition R if the elements of R can be written as unions of the elements of R'.

2.2 Game Models

Definition 1 Two Player Games : *A two-player game structure* $G = \langle S, \lambda, \delta \rangle$ *consists of:*

- *A state space* S.
- *A turn function* $\lambda : S \to \{1, 2\}$, *associating with each state* $s \in S$ *the player* $\lambda(s)$ *whose turn it is to play at the state. We write* $\sim 1 = 2$, $\sim 2 = 1$, *and we let* $S_1 = \{s \in S \mid \lambda(s) = 1\}$ *and* $S_2 = \{s \in S \mid \lambda(s) = 2\}$.
- *A transition function* $\delta : S \mapsto 2^S \setminus \emptyset$, *associating with every state* $s \in S$ *a non-empty set* $\delta(s) \subseteq S$ *of possible successors.*

Definition 2 Markov Decision Processes (MDPs) : *A* Markov decision process *(MDP)* $M = \langle S, A, \Gamma, p \rangle$ *consists of the following components:*

- *A state space* S.
- *A finite set* A *of actions (moves),*
- *A move assignment* $\Gamma : S \to 2^A \setminus \emptyset$.
- *A probabilistic transition function* $p : S \times A \to D(S)$.

At every state $s \in S$, *the controller can choose an action* $a \in \Gamma(s)$; *the MDP then proceeds to the successor state* t *with probability* $p(s, a, t)$, *for all* $t \in S$.

Transition systems (1-player games) are special cases of two-player games where $S_2 = \emptyset$ and for all $s \in S$, $\lambda(s) = 1$. The game takes place over the state space S, and proceeds in an infinite sequence of rounds. At every round, from the current state $s \in S$, player $\lambda(s) \in \{1, 2\}$ chooses a successor state $s' \in \delta(s)$, and the game proceeds to s'. The infinite sequence of rounds gives rise to a *path* $\overline{s} \in S^\omega$: precisely, a *path* of G is an infinite sequence $\overline{s} = s_0, s_1, s_2, \ldots$ of states in S such that for all $k \geq 0$, we have $s_{k+1} \in \delta(s_k)$. We denote by Ω the set of all paths.

2.3 Objectives

A *game*(G,Φ) consists of a game structure G together with an objective Φ for a player. An *qualitative objective* Φ for a game structure G is a subset $\Phi \subseteq S^\omega$ of the sequences of states of G. A *quantitative* objective is specified as a measurable function $f : \Omega \to \mathbb{R}$.

Given a subset $T \subseteq S$ of states, the *reachability* objective $\Diamond T = \{s_0, s_1, s_2, \cdots \in S^\omega \mid \exists k \geq 0. s_k \in T\}$ consists of all paths that reach T; the *safety* objective $\Box T = \{s_0, s_1, s_2, \cdots \in S^\omega \mid \forall k \geq 0. s_k \in T\}$ consists of all paths that stay in T forever. Games with reachability or safety objectives are called reachability and safety games, respectively.

2.4 Strategies

Strategy in Two-Player Games : A *strategy* for player $i \in \{1,2\}$ in a game $G = \langle S, \lambda, \delta \rangle$ is a mapping $\pi_i : S^* \times S_i \mapsto S$ that associates with every nonempty finite sequence σ of states ending in S_i, representing the past history of the game, a successor state. We require that, for all $\sigma \in S^\omega$ and all $s \in S_i$, we have $\pi_i(\sigma s) \in \delta(s)$. An initial state $s_0 \in S$ and two strategies π_1, π_2 for players 1 and 2 uniquely determine a sequence of states $Outcome(s_0, \pi_1, \pi_2) = s_0, s_1, s_2, \ldots$, where for $k > 0$ we have $s_{k+1} = \pi_1(s_0, \ldots, s_k)$ if $s_k \in S_1$, and $s_{k+1} = \pi_2(s_0, \ldots, s_k)$ if $s_k \in S_2$. Given an initial state s_0 and a winning objective $\Phi \subseteq S^\omega$ for player $i \in \{1,2\}$, we say that state $s \in S$ is *winning* for player i if there is a player-i strategy π_i such that, for all player $\sim i$ strategies $\pi_{\sim i}$, we have $Outcome(s_0, \pi_1, \pi_2) \in \Phi$. We denote by $\langle i \rangle \Phi \subseteq S$ the set of winning states for player i for objective $\Phi \subseteq S^\omega$. A result by [57], as well as the determinacy result of [79], ensures that for all ω-regular goals Φ we have $\langle 1 \rangle \Phi = S \setminus \langle 2 \rangle \neg \Phi$, where $\neg \Phi = S \setminus \Phi$. Given a set $\theta \subseteq S$ of initial states, and a property $\Phi \subseteq S^\omega$, we will present algorithms for deciding whether $\theta \cap \langle i \rangle \Phi \neq \emptyset$ or, equivalently, whether $\theta \subseteq \langle i \rangle \Phi$, for $i \in \{1,2\}$.

Strategy in Markov Decision Processes : We model the choice of actions, on the part of the controller, via a *strategy* (strategies are also variously called *schedulers* [94] or *policies* [54]). A *strategy* is a mapping $\pi : S^+ \mapsto D(A)$: given a past history $\sigma s \in S^+$ for the MDP, a strategy π chooses each action $a \in \Gamma(s)$ with probability $\pi(\sigma s)(a)$; we obviously require $\pi(\sigma s)(b) = 0$ for all $b \in A \setminus \Gamma(s)$.

2.5 Controllable Predecessor Operators

Two-player games with reachability, safety winning conditions are commonly solved using *controllable predecessor operators.* We define the *player-1 controllable predecessor operator* $\mathrm{Cpre}_1 : 2^S \mapsto 2^S$ as follows, for all $X \subseteq S$ and $i \in \{1, 2\}$:

$$\mathrm{Cpre}_i(X) = \{s \in S_i \mid \delta(s) \cap X \neq \emptyset\} \cup \{s \in S_{\sim i} \mid \delta(s) \subseteq X\}. \tag{2.1}$$

Intuitively, for $i \in \{1, 2\}$, the set $\mathrm{Cpre}_i(X)$ consists of the states from which player i can force the game to X in one step.

We will express our algorithms for solving games on the state-space S in μ-calculus notation [57]. Consider a function $\gamma : 2^S \mapsto 2^S$, monotone when 2^S is considered as a lattice with the usual subset ordering. We denote by $\mu Z.\gamma(Z)$ (resp. $\nu Z.\gamma(Z)$) the *least* (resp. *greatest*) *fix-point* of γ, that is, the least (resp. greatest) set $Z \subseteq S$ such that $Z = \gamma(Z)$. As is well known, since S is finite, these fix-points can be computed via Picard iteration: $\mu Z.\gamma(Z) = \lim_{n \to \infty} \gamma^n(\emptyset)$ and $\nu Z.\gamma(Z) = \lim_{n \to \infty} \gamma^n(S)$. In the solution of parity games we will make use of nested fix-point operators, which can be evaluated by nested Picard iteration [57].

Algorithm 1 ValIter$(T, f, g, \varepsilon_{float})$ Value iteration

1. $v := [T]$
2. **repeat**
3. $\quad \hat{v} := v$
4. $\quad$ **for all** $s \in S$ **do** $v(s) := f\left([T](s),\ g\left\{\sum_{s' \in S} p(s,a,s') \cdot \hat{v}(s') \;\middle|\; a \in \Gamma(s)\right\}\right)$
5. **until** $||v - \hat{v}|| \leq \varepsilon_{float}$
6. **return** v

2.6 Optimal Values in Markov Decision Processes

These sets of paths Ω are measurable [105] in Markov Decision Processes, so that given a strategy $\pi \in \Pi$, we can define the probabilities $\Pr_s^\pi(\Diamond T)$, $\Pr_s^\pi(\Box T)$ of following a path in these sets from an initial state $s \in S$ under strategy π. By choosing appropriate strategies, the controller can maximize or minimize these probabilities. Thus, we consider the problem of computing, at all $s \in S$, the quantities:

$$V_{\Box T}^{\max}(s) = \max_{\pi \in \Pi} \Pr_s^\pi(\Box T) \qquad V_{\Diamond T}^{\max}(s) = \max_{\pi \in \Pi} \Pr_s^\pi(\Diamond T)$$

$$V_{\Box T}^{\min}(s) = \min_{\pi \in \Pi} \Pr_s^\pi(\Box T) \qquad V_{\Diamond T}^{\min}(s) = \min_{\pi \in \Pi} \Pr_s^\pi(\Diamond T).$$

The fact that on the right-hand side we have max, min rather than sup, inf is a consequence of the existence of optimal (and memoryless) strategies [54]. Thus, strategies can be both history-dependent, and randomized. We denote by Π the set of all strategies.

2.6.1 Implementation via Value Iteration

Reachability and safety probabilities on an MDP can be computed via a classical value-iteration scheme [54, 15, 49]. The algorithm, depicted as Algorithm 1, is parameterized

Quantity	f	g	convergence
$V^{\max}_{\Box T}$	min	max	from above
$V^{\min}_{\Box T}$	min	min	from above
Quantity	f	g	convergence
$V^{\max}_{\Diamond T}$	max	max	from below
$V^{\min}_{\Diamond T}$	max	min	from below

Table 2.1: Parameters to be used in the call to ValIter(T, f, g) in order to compute reachability and safety properties. The table also indicates whether the computation converges from above, or from below.

by two operators $f, g \in \{\max, \min\}$. The operator f specifies how to merge the valuation of the current state with the expected next-state valuation; we use $f = \max$ for reachability goals, and $f = \min$ for safety ones. The operator g specifies whether to select the action that maximizes, or minimizes, the expected next-state valuation; we use $g = \max$ to compute maximal probabilities, and $g = \min$ to compute minimal probabilities, The algorithm is also parameterized by $\varepsilon_{float} > 0$: this is the threshold below which we consider value iteration to have converged. The following facts are well-known (see, e.g., [54, 33, 34]). For all $\varepsilon_{float} > 0$ and for all $f, g \in \{\min, \max\}$, the call ValIter$(T, f, g, \varepsilon_{float})$ terminates. Moreover, consider any $g \in \{\max, \min\}$ and any $\triangle \in \{\Box, \Diamond\}$, and let $f = \min$ if $\triangle = \Box$, and $f = \max$ if $\triangle = \Diamond$. Then, for all $\delta > 0$, there is an $\varepsilon_{float} > 0$ such that, at all $s \in S$:

$$v(s) - \delta \leq V^{g}_{\triangle T}(s) \leq v(s) + \delta$$

where $v =$ ValIter$(T, f, g, \varepsilon_{float})$. We can replace statement 1 of Algorithm 1 with the following initialization: **if** $f = \max$ **then** $v := \mathbf{0}$ **else** $v := \mathbf{1}$.

Part II

Applications of Games

Chapter 3

Interface Theories in Component Based Design and TICC

Open systems are systems whose behavior is jointly determined by their internal structure, and by the inputs that they receive from their environment. In previous work, it has been argued that games constitute a natural model for open systems [35] We use games to represent the interaction between the behavior originating within a component, and the behavior originating from the components environment. In particular, we model components as Input-Output games: the moves of Input represent the behavior the component can accept from the environment, while the moves of Output represent the behavior the component can generate. Unlike component models based on transition systems, models based on games provide a notion of compatibility [42, 44]. When two components P and Q are composed, we can check whether the output behavior of P satisfies the input requirements of Q, and vice-versa. However, we do not define P and Q to be compatible only if their input requirements are always satisfied. Rather, we recognize that the output behavior of P and Q can still be influenced by their residual interaction with the environment (unless the composition of P and Q is closed). Thus, we define

P and Q to be compatible if there is some environment under which their input assumptions are mutually satisfied, and we associate with their composition P||Q the weakest (most general) assumptions about the environment that guarantee mutual compatibility. In game-theoretic terms, P and Q are compatible if, in their joint model, Input has a strategy to guarantee that all outputs from P to Q can be accepted by Q, and vice-versa; the environment assumption of P||Q is simply the most general such Input strategy. These game-based component models have been called interface theories, and two tools for interface theories predate Ticc. The asynchronous, action-based interface theories of [42, 44] are implemented as part of the Ptolemy tool-set [73]. The tool Chic implements synchronous, variable-based interface theories closely modeled after [24]. Our goal in developing Ticc was to provide an asynchronous model where components have rich communication primitives that facilitate the modeling of software and distributed systems. In Ticc, variables encode both the local state of the components (called modules) and the global state of the system. Modules synchronize on shared actions, and the occurrence of actions can cause variables to be updated. Each global variable can be updated by more than one module, so that it is both read and write-shared; restrictions ensure that variable updates are free from race-conditions. An action can appear in a module both as input and as output. If an action a occurs in a module P as output, but not as input, then P can generate a, but not accept it from other modules. If a occurs in P both as input and as output, then P can both generate a, and accept it from other modules. This enables the encoding of rich communication schemes, including exclusive, and many-to-many schemes, and differentiates the modules of Ticc from other modules with more restrictive communication primitives, such as I/O Automata [76] and Reactive Modules [7]. The theory behind Ticc has been presented in detail in [39]; here, we describe the tool itself.

3.1 Tool

Ticc parses interfaces, called modules, encoded in a guarded-command language, and builds symbolic representations for these interfaces that are used for compatibility checking and composition. Ticc is written in OCaml [74], and the symbolic algorithms rely on the MDD/BDD Glue and Cudd packages [100]. The code of Ticc is freely available and can be downloaded from http://dvlab.cse.ucsc.edu/dvlab/Ticc. This web site is an open Wiki that also contains the documentation for the tool, and several additional examples. We illustrate the modeling language of Ticc by means of a simple example: a fire detection system. The system is composed of a control unit and several smoke detectors. When a detector senses smoke (action smoke), it reports it by emitting the action fire. When the control unit receives action fire from any of the detectors, it emits the action call fd , corresponding to a call to the fire department. Additionally, an input disable disables both the control unit and the detectors, so that the smoke sensors can be tested without triggering an alarm. We provide the code for the control unit module (ControlUnit), for one of the (several) fire detectors (FireDetector1), as well as for a faulty detector that ignores the disable messages (Faulty FireDetector2): The body of each module starts with the list of its local variables; Ticc supports Boolean and integral range variables. The transitions are specified using guarded commands guard) command, where guard and command are boolean expressions over the local and global variables; as usual, primed variables refer to the values after a transition is taken. For instance, the output transition fire in module FireDetector1 can be taken only when s has value 1; the transition leads to a state where $s = 2$.

```
module ControlUnit:
var s: [0..3] // 0=waiting, 1=alarm raised, 2=fd called, 3=disabled

input fire: { local: s = 0 | s = 1 ==> s := 1
else s = 2 ==> }
```

```
input disable: { local: true ==> s := 3 }
output call_fd: { s = 1 ==> s = 2 }
endmodule

module FireDetector1:
var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive

input smoke1: { local: s = 0 | s = 1 ==> s := 1
else s = 2 ==> } // do nothing if inactive

output fire: { s = 1 ==> s = 2 }

input fire: { } // accepts (and ignores) fire inputs

input disable: { local: true ==> s := 2 }
endmodule

module Faulty_FireDetector2:
var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive

input smoke2: { local: s = 0 | s = 1 ==> s := 1
else s = 2 ==> } // do nothing if inactive

output fire: { s = 1 ==> s = 2 }

input fire: { } // accepts (and ignores) fire inputs
  // does not listen to disable action
endmodule
```

When modules ControlUnit and FireDetector1 are composed, they synchronize on the shared actions fire and disable. First, input transitions in a module synchronize with the corresponding output transitions in the other module. Thus, the output transition labeled with fire in FireDetector1 synchronizes with the input transitions labeled with fire in ControlUnit. Moreover, input transitions associated to a shared action in different modules also synchronize. For instance, the input transitions associated with fire in FireDetector1 and ControlUnit synchronize, so that the composition FireDetector1||ControlUnit can also accept fire as input, and can therefore be composed with other fire detectors. The composition of ControlUnit and Faulty FireDetector2 goes less smoothly. When the composition receives a disable action, the control unit shuts down ($s = 3$), while the faulty detector remains in operation. When the

faulty detector senses smoke (input smoke2), it will emit fire: if the control unit has been disabled by the disable action, this causes an incompatibility. Ticc diagnoses this incompatibility by synthesizing the following input restrictions:

- A restriction preventing the input disable if the faulty detector is in state $s = 1$, that is, it has detected smoke and is about to issue fire.
- A restriction preventing the input smoke2 when Control-Unit is at $s = 3$ (disabled).

Since the actions disable and smoke2 should be acceptable at any time, the new input restrictions for these actions are a strong indication that the composition Control-Unit || Faulty Fire-Detector2 does not work properly.

3.2 Using TICC

Ticc is implemented as a set of functions that extends the capabilities of the OCaml command-line. The incompatibility mentioned in the previous section is exposed by the following series of OCaml commands:

```
# open Ticc;;
# parse "fire-detector-disable.si";;
# let controlunit = mk_sym "ControlUnit";;
# let wfire2 = mk_sym "Faulty_FireDetector2";;
# print_input_restriction (compose controlunit wfire2) "disable";;
# print_input_restriction (compose controlunit wfire2) "smoke2";;
```

The mk sym function builds a symbolic representation of a module, given the module name. The last two lines print how the input actions have been restricted in the composition.

Chapter 4

Online Testing with Learning

4.1 Introduction

Many software systems are reactive. The behavior of a reactive system, especially when distributed or multi-threaded, can be nondeterministic. For example, systems may produce spontaneous outputs like asynchronous events. Factors such as thread scheduling are not entirely under the control of the tester but may still affect the behavior observed. In these cases, a test suite generated offline may be infeasible, since all of the observable behaviors would have to be encoded a priori as a decision tree, and the size of such a decision tree can be very large.

Online testing (also called on-the-fly testing) can be more appropriate than offline tests for reactive systems. The reason is that with online testing the tests may be dynamically adapted at runtime, effectively pruning the search space to include only those behaviors actually observed instead of all possible behaviors. The interaction between tester and implementation under test (IUT) is seen as a game [4] where the tester chooses moves based on the observed behavior of the implementation under test. Only the tester is assumed to have a goal; the other player (the IUT) is unaware that it is playing. This kind of game is known in the literature as

a $1\frac{1}{2}$-player game [26].

Online testing is a form of *model-based testing (MBT)*, where the tester uses a specification (or *model*) of the system's behavior to guide the testing and to detect the discrepancies between the IUT and the model. It is an established technique, supported in tools like TorX [103] and Spec Explorer [106]. We express the model as a set of guarded update rules that operate on an abstract state. This formulation is called a *model program.* Both the IUT and the model are viewed as *interface automata* [47] in order to establish a a formal conformance relation between them.

We distinguish between moves of the tester and moves of the IUT. The actions available to the tester are called *controllable* actions. The IUT's responses are *observable* actions. A *conformance failure* occurs when the IUT rejects a controllable action produced by the model or when the model rejects an observable action produced by the IUT.

A principal concern of online testing is the *strategy* used to choose test actions. A poor strategy may fail to provoke behaviors of interest or may take an infeasible amount to time to achieve good coverage. One can think of strategy in economic terms. The cost of testing increases with the number of test runs and the number of steps per run. We want to minimize the number of steps taken to achieve a given level of coverage for the possible behaviors. Exhaustive coverage is often infeasible. Instead, we strive for the best coverage possible within fixed resource constraints. The main challenge is to choose actions that minimize backtracking, since resetting the IUT to its initial state can be an expensive operation.

A purely random strategy for selecting test actions can be wasteful in this regard, since the tester may repeat actions that have already been tested or fail to systematically explore the reachable model states. A random strategy cannot benefit from remembering actions chosen in previous runs.

In this chapter we propose an algorithm for online testing, using the ideas from *Rein-*

forcement Learning (RL) [101, 69]. RL techniques address some of the drawbacks of random action selection. Our algorithm is related to the anti-ant algorithm introduced in [75], which avoids the generation of redundant test cases from UML diagrams.

RL refers to a collection of techniques in which an *agent* makes moves (called *actions*) with respect to the *state* of an environment. Actions are associated with *rewards* or *costs* in each state. The agent's goal is to choose a sequence of actions to maximize expected reward or, equivalently, to minimize expected cost.

The history needed to compute the strategy is encoded in a data structure called a "Test-Trace Graph (TTG)". We compare several such strategies below. The results show that a greedy strategy (*Least-Cost*) has a suboptimal solution. The probability of reaching a failure state does not change with a purely randomized strategy (*Random*), though the probability reduces monotonically in a randomized greedy strategy (*RandomizedLeastCost*). This is because the probability in the latter case is negatively reinforced by the number of times a failure state has been visited, whereas it remains same in the former case.

The contributions of this chapter are the following:

- We transform the online testing problem into a special case of reinforcement learning where the frequencies of various abstract behaviors are recorded. This allows us to better choose controllable actions.
- We show with benchmarks that an RL-based approach can significantly outperform random action selection.

4.2 Testing Theory

In model-based testing a tester uses a specification for two purposes. One is *conformance checking*: to decide if the IUT behaves as expected or specified. The other is *scenario*

control: which actions should be taken in which order and pattern. Model-based testing is currently a growing practice in industry. In many respects the second purpose is the main use of models to drive tests and relates closely to test scenarios is traditional testing. However, with a growing complexity and need for protocol level testing and interaction testing, the first purpose is gaining importance.

Formally, model programs are mapped (unwound) to interface automata in order to do conformance checking. The conformance relation that is used can be defined as a form of alternating refinement. This form of testing is provided by the Spec Explorer tool, see e.g. [106].

4.2.1 Model Programs as Specifications

States are *memories* that are finite mappings from (memory) locations to a fixed universe of values. By an update rule we mean here a finite representation of a function that given a memory (state) produces an updated memory (state). A update rule p may be parameterized with respect to a sequence of *formal input parameters* $\bar{x}$, denoted by $p[\bar{x}]$. The instantiation of $p[\bar{x}]$ with input values $\bar{v}$ of appropriate type, is denoted by $p[\bar{v}]$. In general, an update rule may be *nondeterministic*, in which case it may yield several states from a given state and given inputs. Thus, an *update rule* $p[x_1, \ldots, x_n]$ denotes a relation $[\![p]\!] \subseteq \mathit{States} \times \mathit{Values}^n \times \mathit{States}$. When p is deterministic, we consider $[\![p]\!]$ as a function $[\![p]\!] : \mathit{States} \times \mathit{Values}^n \rightarrow \mathit{States}$ and we say that the *invocation* (or *execution*) of $p[\bar{v}]$ from state s yields the state $[\![p]\!](s, \bar{v})$.

A *guard* φ is a state dependent formula that may contain free logic variables $\bar{x} = x_1, \ldots, x_n$, denoted by $\varphi[\bar{x}]$; φ is *closed* if it contains no free variables. Given values $\bar{v} = v_1 \ldots, v_n$ we write $\varphi[\bar{v}]$ for the replacement of x_i in φ by v_i for $1 \leq i \leq n$. A closed formula φ has the standard truth interpretation $s \models \varphi$ in a state s. A *guarded update rule* is a pair (φ, p) containing a guard $\varphi[\bar{x}]$ and an update rule $p[\bar{x}]$; intuitively (φ, p) limits the execution of p to those states and arguments $\bar{v}$ where $\varphi[\bar{v}]$ holds.

Definition 3 A *model program* P has the following components.

- A state space *States*.
- A value space *Values*.
- An *initial state* $s_0 \in States$,
- A finite vocabulary Σ of *action symbols* partitioned into two disjoint sets
  - Σ^c of *controllable* action symbols, and
  - Σ^o of *observable* action symbols.
- A *reset* action symbol $Reset \in \Sigma^c$.
- A family $(\varphi_f, p_f)_{f \in \Sigma}$ of guarded update rules.
  - The *arity* of f is the number of input parameters of p_f.
  - The arity of *Reset* is 0 and $[\![p_{Reset}]\!](s) = s_0$ for all $s \models \varphi_{Reset}$.

P is *deterministic* if, for all action symbols $f \in \Sigma$, p_f is deterministic.

An n-ary action symbol has logically the term interpretation, i.e. two ground terms whose function symbols are action symbols are equal if and only if the action symbols are identical and their corresponding arguments are equal. An *action* has the form $f(v_1, \ldots, v_n)$ where f is an n-ary action symbol and each v_i is a value that matches the required type of the corresponding input parameter of p_f. We say that an action $f(\bar{v})$ is *enabled* is a state s if $s \models \varphi(\bar{v})$. Notice the two special cases regarding reset: one when reset is always disabled ($\varphi_{Reset} = false$), in which case the definition of p_{Reset} is irrelevant, and the other one when reset is always enabled ($\varphi_{Reset} = true$), in which case p_{Reset} must be able to reestablish the initial state from any other program state.

We sometimes use *action* to mean an action symbol, when this is clear from the context or when the action symbol is nullary in which case there is no distinction between the two.

4.2.2 Example: *Recycling Robot*

We show a model program of a collection of *recycling robots* written in C# in Figure 4.1. A robot is a movable recycle-bin, it can either

1. move and *search* for a can if its power level (measured in percentage) is above the given threshold 30%, or
2. remain stationary and *wait* for people to dispose of a can if its power level is below the given threshold 50%.

A robot gets a reward by collecting cans. The reward is bigger when searching than while waiting, but each search reduces the power level of the robot by 30%. A robot can be *recharged* when it is not fully charged, i.e when the power level is less than 100%. New robots can be *started* dynamically provided that the total number of robots does not exceed a limit (if such a limit is given).

Actions In this example, the action symbols are `Start`, `Search`, `Wait` and `Recharge`, where the first three symbols are classified as being controllable and the last one is classified as being observable. All of the symbols are unary (i.e., they take one input). All actions have the form $f(i)$ where f is one of the four action symbols and i is a non-negative integer representing the id of a robot. The reset action is in this example implicit, and is assumed to be enabled in all states.

States The state signature has three state variables, a map `Robot.Instances` from object ids (natural numbers) to robots (objects of type `Robot`), and two field value maps `power` and `reward`

```
class Robot : EnumeratedInstance // The base class keeps track of created robot instances
{
    int power  = 0;
    int reward = 0;
}

class RobotModel
{
    static int maxNoOfRobots = ...;

    [Action]
    static void Start(int robotId)
    {
        Assume.IsTrue(Robot.Instances.Count < maxNoOfRobots &&
                      ¬ Robot.Instances.Count == robotId));
        new Robot(robotId);
    }

    [Action]
    static void Search(int robotId)
    {
        Assume.IsTrue(robotId ∈ Robot.Instances);
        Robot robot = Robot.Instances[robotId];
        Assume.IsTrue(robot.power > 30);

        robot.power  = robot.power - 30;
        robot.reward = robot.reward + 2;
    }

    [Action]
    static void Wait(int robotId)
    {
        Assume.IsTrue(robotId ∈ Robot.Instances);
        Robot robot = Robot.Instances[robotId];
        Assume.IsTrue(robot.power <= 50);

        robot.reward = robot.reward + 1;
    }

    [Action(Kind = Observable)]
    static void Recharge(int robotId)
    {
        Assume.IsTrue(robotId ∈ Robot.Instances);
        Robot robot = Robot.Instances[robotId];
        Assume.IsTrue(robot.power < 100);

        robot.power = 100;
    }
}
```

Figure 4.1: Model Program of the *Recycling Robot* example.

that map robots to their corresponding power and reward values. The initial state is the state where all those maps are empty.

Guarded update rules For each of the four actions f the guarded update rule (φ_f, p_f) is defined by the corresponding static method f of the `RobotModel` class. Given a robot id i and a state s, the guard $\varphi_f(i)$ is true in s, if all the `Assume.IsTrue` statements evaluate to *true* in s. Execution of $p_f[i]$ corresponds to the method invocation of $f(i)$. For example, in the initial state, say s_0, of the robot model, the single enabled action is `Start(0)`. In the resulting state

$[\![p_{\texttt{Start}}]\!](s_0, 0)$ a new robot with id 0 has been created whose reward and power are 0.

4.2.3 Deterministic model programs as interface automata

We use the notion of interface automata [47, 36] following the exposition in [36]. The view of a model program as an interface automaton is important for formalizing the conformance relation. In this chapter, we use the terms "controllable" and "observable" here instead of the terms "input" and "output" used in [36].

Definition 4 An *interface automaton* M has the following components:

- A set S of *states.*
- A nonempty subset S^{init} of S called the *initial states.*
- Mutually disjoint sets of *controllable actions* A^{c} and *observable actions* A^{o}.
- *Enabling functions* Γ^{c} and Γ^{o} from S to subsets of A^{c} and A^{o}, respectively.
- A *transition function* δ that maps a source state and an action enabled in the source state to a target state.

In order to identify a component of an interface automaton M, we index that component by M, unless M is clear from the context. Let P be a deterministic model program ($States$, $Values$, s_0, Σ, Σ^{c}, Σ^{o}, $Reset$, $(\varphi_f, p_f)_{f \in \Sigma}$). P has the following straightforward denotation $[\![P]\!]$ as an interface automaton:

$$S_{[\![P]\!]} = \mathit{States}$$

$$S^{\text{init}}_{[\![P]\!]} = \{s_0\}$$

$$A^{\text{c}}_{[\![P]\!]} = \{f(\bar{v}) \mid f \in \Sigma^{\text{c}}, \bar{v} \subseteq \mathit{Values}\}$$

$$A^{\text{o}}_{[\![P]\!]} = \{f(\bar{v}) \mid f \in \Sigma^{\text{o}}, \bar{v} \subseteq \mathit{Values}\}$$

$$\Gamma^{\text{c}}_{[\![P]\!]}(s) = \{f(\bar{v}) \in A^{\text{c}}_{[\![P]\!]} \mid s \models \varphi_f(\bar{v})\}$$

$$\Gamma^{\text{o}}_{[\![P]\!]}(s) = \{f(\bar{v}) \in A^{\text{o}}_{[\![P]\!]} \mid s \models \varphi_f(\bar{v})\}$$

$$\delta_{[\![P]\!]}(s, f(\bar{v})) = [\![P_f]\!](s, \bar{v}) \quad (\text{for } f \in \Sigma,\ s \in \mathit{States},\ s \models \varphi_f(\bar{v}))$$

Note that $\delta_{[\![P]\!]}$ is well-defined, since P is deterministic. In light of the above definition we occasionally drop the distinction between P and the interface automaton $[\![P]\!]$ it denotes.

4.2.4 Implementing a Model Program as an Interface Automaton

A model program P exposes itself as an interface automaton through a *stepper* that provides a particular "walk" through the interface automaton one transition at a time. A stepper of P is implemented through the `IStepper` interface defined below. A stepper has an implicit *current state* that is initially the initial state of P. In the current state s of a stepper, the enabled actions are given by $\Gamma_{[\![P]\!]}(s)$. Doing a step in the current state s of the stepper according to a given action a corresponds to setting the current state of the stepper to $\delta_{[\![P]\!]}(s, a)$. The *Reset* action is handled separately and is not included in the set of currently enabled actions `EnabledControllables`.

```
interface IStepper
{
    Sequence<Action> EnabledControllables { get; }
    Sequence<Action> EnabledObservables { get; }
    void DoStep(Action action);

    void Reset();
```

```
    bool ResetEnabled { get; }
}
```

For conformance testing, an implementation is also assumed to be an interface automaton that is exposed through a stepper. If both the model and the IUT are interface automata with a common action signature, we test the conformance of the two automata using the refinement relation between interface automata as defined in [36].

4.3 Online Testing Algorithm

In this section we describe an algorithm that uses reinforcement learning to choose controllable actions during conformance testing of an implementation I against a model (specification) M. Both M and I are assumed to be given as model programs that expose an `IStepper` interface to the algorithm. In addition, the model exposes an interface that provides an abstract value of the current state of the model and an abstract value of any action enabled in a given state. It is convenient to view this interface as an extension `IModelStepper` of the IStepper interface:

```
interface IModelStepper : IStepper
{
    IComparable GetAbstractState(Action action);
    IComparable GetAbstractAction(Action action);
}
```

The main motivation for these functions is to divide the state space and the action space into equivalence classes that reflect "interesting" groups of states and actions for the purposes of coverage.

Example 1 *Consider the Robot model. We could define the abstract states and abstract actions to be the concrete states and the concrete actions as follows. In other words, there is no grouping of either states or actions in this case.*

```
class RobotModel : IModelStepper
{
    Sequence<Pair<int,int>> GetAbstractState(Action action)
```

```
    {
        return [(r.power, r.reward) | r in Robot.Instances]
    }
    Action GetAbstractAction(Action action);
    {
        return action;
    }
}
```

Example 2 *A more interesting case is if we abstract away the id of the robot and project the state to the state of the robot doing the action, or a default value if the robot has not been started yet. This is reasonable because the robots do not interact with each other.*

```
class RobotModel : IModelStepper
{
    Pair<int,int> GetAbstractState(Action action)
    {
        if (action.Name == "Start") return (-1, -1);
        Robot r = Robot.Instances[action.Argument(0)];
        return (r.power, r.reward);
    }
    string GetAbstractAction(Action action);
    {
        return action.Name;
    }
}
```

We use pseudo code that is similar to the original implementation code written in C# to describe the algorithm. We consider two controllable action selection *strategies* `Lct` and `Rlc` that are explained below, in addition to a memoryless purely randomized strategy `Rnd`.

```
enum Strategy {Rnd, Lct, Rlc}
```

The algorithm uses also an "oracle" to ask advice about whether to observe an observable action from the implementation, to call a controllable action, or to end a particular test run, during a single step of the algorithm. The oracle makes a random choice between controlling an observing when an observable action is enabled in the implementation at the same time as a controllable action is enabled in the model. If there are no observable actions enabled in the implementation and no controllable actions enabled in the model then the only meaningful advice the oracle can give is to end the current test run.

```
enum Advice {Control, Observe, End}

class Oracle
{
    IStepper M;
    IStepper I;

    Advice Advise()
    {
```

```
        bool noCtlrs = M.EnabledControllables.IsEmpty;
        bool noObs = I.EnabledObservables.IsEmpty;

        if (noCtlrs ∧ noObs) return Advice.End;
        if (noCtlrs) return Advice.Observe;
        if (noObs) return Advice.Control;
        return new Choose(Advice.Control, Advice.Observe);
    }
}
```

4.3.1 Top level loop

The top level loop of the algorithm is described by the following pseudo code.

```
class OnlineTesting
{
    IModelStepper M;
    IStepper I;
    int maxRun;
    int maxStep;
    Strategy h;
    Oracle oracle;

    bool ResetEnabled {get return M.ResetEnabled ∧ I.ResetEnabled;}

    void Run()
    {
        int run = 0;
        while (run < maxRun)
        {
            RunTestCase();              // The core algorithm
            if (¬ResetEnabled) return; // Cannot continue, must abort
            Reset();
            run += 1;
        }
    }
}
```

The inputs to the algorithm are a model program `M` that provides the `IModelStepper` interface and is the specification, a model program `I` that provides the `IStepper` interface an is the implementation under test, an upper bound `maxRun` on the total number of runs, an upper bound `maxStep` on the total number of steps (state transitions) per one run, a strategy `h`, and an oracle `oracle` as explained above.

4.3.2 The Core Algorithm

The algorithm keeps track of the *weights* of *abstract transitions* that have occurred during the test runs. An abstract transition is a pair (s, a) where s is an abstract state and a is an abstract action. The weight of an abstract transition is total number of times it has occurred plus one, since the algorithm was started. The abstract state and action values are calculated

using the `IModelStepper` interface introduced above. This weight information is stored in a *test trace graph* that is updated dynamically and is initially empty.

```
class TestTraceGraph
{
    Map<AbstractTransition, int> F = ∅;  // Frequencies of explored abstract transitions
    IModelStepper M;

    int W(Action a)                      // Weights are positive
    {
        AbstractState s = M.GetAbstractState(a);
        AbstractAction b = M.GetAbstractAction(a);
        if ((s,b) ∈ F) return F[(s,b)]; else return 1;
    }

    void Update(Action a, int w)
    {
        AbstractState s = M.GetAbstractState(a);
        AbstractAction b = M.GetAbstractAction(a);
        F[(s,b)] = W(a) + w;
    }
}
```

The next controllable action is chosen by the algorithm from a nonempty set of controllable actions that are currently enabled, using the given strategy.

```
class TestTraceGraph
{
    Action ChooseAction(Sequence<Action> acts, Strategy h)
    {
        switch (h)
         {
            case Strategy.Lct:
                Action a = acts.Head;
                Pair<Set<Action>,int> lct =
                    acts.Tail.Reduce(Reducer,({acts.Head},W(acts.Head)));
                return lct.First.Choose();

            case Strategy.Rlc:
                Sequence<int> costs = [W(a) | a ∈ acts];
                int prod = ...; // Compute an approximate common multiple of costs
                Sequence<int> occurs = [prod/x | x ∈ costs];
                Bag<Action> bg = {|(acts[i], occurs[i]) | i < acts.Count|};
                return bg.Choose();

            default:
                return acts.Choose();
         }
     }
    Pair<Set<Action>,int> Reducer(Action a, Pair<Set<Action>,int> lct)
    {
        if (W(a) < lct.Second) return ({a}, w);
        else if (W(a) == lct.Second) return (lct.First ∪ {a}, w);
        else return lct;
    }
}
```

`Lct`: Choose an action that has the "least cost". Here *cost* of an action a is measured as the current weight of the abstract transition (s, b), where s is the abstract state computed in the current model state with respect to a, and b is the abstract action corresponding to a, computed in the current model state. If several actions have the same least cost, one is chosen randomly from among those.

`Rlc`: Choose an action with a likelihood that is inversely proportional to its current cost, with cost having the same meaning as above. Intuitively this means that the least frequent actions are the most favored ones. In other words, if the candidate actions are $(a_i)_{i<k}$ for some k, having costs $(c_i)_{i<k}$, then the probability of selecting the action a_i is $c_i^{-1} / \sum_{j \neq i} c_j^{-1}$. The implementation uses a built-in bag construct to make such a choice.

`Rnd`: Make a random choice.

The algorithm runs one test case until, either a conformance failure occurs (in form of a violation of the refinement relation between $[\![\mathtt{M}]\!]$ and $[\![\mathtt{I}]\!]$), or until the given maximum number of steps has been reached.

```
class OnlineTesting
{
    TestTraceGraph ttg = new TestTraceGraph(M);

    bool RunTestCase()
    {
        int step = 0;
        while (step < maxStep)
        {
            Advice advice = oracle.Advise();

            if (advice == Advice.Control)
            {
                Sequence<Action> cs = M.EnabledControllables;
                Action c = ttg.ChooseAction(cs, h);
                ttg.Update(c, 1);           // Increase the weight by 1
                M.DoStep(c);                // Do the step in M

                if (c ∈ I.EnabledControllables)
                    I.DoStep(c);            // Do the corresponding step in I
                else
                    return false;           // Conformance failure occurred
            }
            else if (advice == Advice.Observe)
            {
                Sequence<Action> os = I.EnabledObservables;
                // This is an abstract view of the execution of the implementation, in reality
                // the implementation performs the choice itself and notifies the test harness
                Action o = os.Choose();
                I.DoStep(o);

                if (o ∈ M.EnabledObservables)
                {
                    ttg.Update(o, 1);       // Increase the weight by 1
                    M.DoStep(o);            // Do the corresponding step in M
                }
                else
                    return false;           // Conformance failure occurred

                #endregion
            }
            else
                return true;                // No more steps can be performed
            step += 1;
        }
        return true;                        // The test case succeeds
    }
}
```

The *Lct* strategy is a greedy approach; it is very simple and relatively cheap to compute. However, it favors actions that have been used less frequently, and thus may systematically avoid long sequences of the same action, as is illustrated next.

Example 3 *Consider a bounded stack of size n. The stack has two controllable actions,* top *and* push*, enabled in every state. The greedy strategy will alternate between these two actions until the stack is full. If we want to test the behavior of* push *when the stack is full, we need to continue testing for at least* $2n$ *steps (so that* push *is executed n times).*

In the given algorithm, the weight increase is always 1. This value can be made domain specific and can vary depending both on the action and the current state, for example by extending the `IModelStepper` interface with a function that provides the wait increase for the given action in the current state and using that function instead of 1.

By using `Rlc`, the probability of selecting an action is inversely proportional to its frequency. Thus, the more an action has been selected the less likely it is that it will be selected again. So the potential problem shown in Example 3 is still there but ameliorated, since no enabled action is excluded from the choice.

4.4 Experiments

We used the Robot model to conduct a few experiments with the algorithm in order to evaluate and compare the different strategies. The main purpose was to see if the two proposed strategies `Lct` or `Rlc` are useful by providing better or at least as good coverage of the state space as the purely random approach. Since we are interested in state and transition coverage only, we ran the algorithm against a correct implementation. We ran the algorithm with a different maximum number of robots, different abstraction functions introduced in the examples above, and different limits on the total number of runs and the total number of steps per run. The

experiments are summarized in Tables 4.1 and 4.2. We ran each case independently 50 times, the entries in the tables are shown on the form $m \pm \sigma$ where m is the mean of the obtained results and σ is the standard deviation. The absolute running times are shown only for comparison, the concrete machine was a 3GHz Pentium 4.

If states and actions are not grouped at all, by assuming the definitions given in Example 1, the majority of abstract transitions will occur only a single time and the strategies perform more or less as the random case, which is shown in Table 4.1. One can see that `Lct` performs marginally better than `Rnd` when the number of robots and the number of runs increases.

Table 4.1: Execution of the online algorithm on the Robot model without grouping.

Parameters			Lct		Rlc		Rnd	
Robots	Runs	Steps	#States	t(ms)	#States	t(ms)	#States	t(ms)
1	1	100	100 ± 0	3	100 ± 0	1	100 ± 0	1
1	10	100	420 ± 11	20	415 ± 8	19	414 ± 9	15
1	100	100	503 ± 3	275	503 ± 3	241	502 ± 2	172
1	100	500	2485 ± 5	1303	2485 ± 5	1292	2485 ± 6	968
2	1	100	100 ± 0	3	100 ± 0	1	100 ± 0	2
2	10	100	951 ± 8	24	941 ± 10	22	938 ± 12	14
2	100	100	7449 ± 83	286	7085 ± 110	284	7055 ± 114	201
2	100	500	44119 ± 225	1548	42437 ± 339	1479	42364 ± 289	1040
5	1	100	100 ± 0	5	100 ± 0	3	100 ± 0	1
5	10	100	972 ± 3	42	971 ± 3	37	969 ± 4	18
5	100	100	9368 ± 17	516	9328 ± 22	468	9322 ± 24	297
5	100	500	49364 ± 19	2794	49330 ± 25	2541	49320 ± 19	1587

When the states and the actions are mapped to abstract values, as defined in Example 2, then `Lct` starts finding many more abstract states than `Rnd` as the number of robots grows. The robot id is ignored by the abstraction and thus concrete transitions of different robots that differ only by the id are mapped to the same abstract transition. Overall this will have the effect that the `Lct` approach will favor actions that transition to new abstract states. The same is true for

the `Rlc` case but the increase in coverage is smaller.

Table 4.2: Execution of the online algorithm on the Robot model with state grouping and action grouping.

Parameters			Lct		Rlc		Rnd	
Robots	Runs	Steps	#States	t(ms)	#States	t(ms)	#States	t(ms)
1	1	100	100 ± 0	3	100 ± 0	<1	100 ± 0	<1
1	10	100	417 ± 9	9	413 ± 8	7	416 ± 8	4
1	100	100	502 ± 2	100	503 ± 3	88	502 ± 2	44
1	100	500	2486 ± 5	508	2486 ± 6	417	2484 ± 6	234
2	1	100	100 ± 0	1	90 ± 3	<1	93 ± 5	<1
2	10	100	419 ± 7	10	284 ± 21	9	237 ± 8	4
2	100	100	502 ± 3	106	437 ± 12	96	293 ± 6	46
2	100	500	2485 ± 5	561	1602 ± 33	506	1324 ± 15	241
5	1	100	100 ± 0	<1	66 ± 4	1	61 ± 2	<1
5	10	100	418 ± 10	10	279 ± 30	11	117 ± 5	5
5	100	100	503 ± 3	115	472 ± 7	116	155 ± 7	50
5	100	500	2484 ± 5	561	1696 ± 96	657	582 ± 10	247
5	100	1000	4949 ± 8	1200	2467 ± 95	1388	1088 ± 13	540
10	10	100	418 ± 9	10	293 ± 25	12	91 ± 6	5
10	100	100	502 ± 3	103	473 ± 6	137	128 ± 6	59
10	100	1000	4951 ± 11	1131	3541 ± 198	1718	602 ± 10	578
10	1000	1000	4985 ± 8	12521	4352 ± 66	18043	654 ± 9	5953

The Robot case study is representative for models that deal with multiple agents at the same time, which is a typical case in testing of multi-threaded software [106]. Often the threads are mostly independent, an abstraction technique that can be used in this context is to look at the part of the state that belongs to the agent doing the action. This is an instance of so-called multiple state-grouping approach that is also used as an exploration technique for FSM generation [23]. This is exactly what is done in Example 2. It seems that `Lct` is a promising heuristic for online testing of these kinds of models. One can note that, the coverage provided by the random approach degrades almost by half as the number of robots is doubled (for example from 5 to 10).

4.5 Related Work

The basic idea of online testing has been introduced in the context of labeled transition systems using ioco theory [20, 102, 104] and implemented in the TorX tool [103]. TGV [68] is another tool frequently used for online or on-the-fly test generation that uses ioco. Ioco theory is a formal testing theory based on labeled transition systems with input actions and output actions. Interface automata [36] are suitable for the game view [25] of online testing and provide the foundation for the conformance relation that we use. Online testing with model programs in the Spec-Explorer tool is discussed in in [106]. The algorithm in [106] does not use learning, and as far as we know learning algorithms have not been considered in the context of model based testing. The relation between ioco and refinement of interface automata is briefly discussed in [106]. Specifications given by a guarded command language are used also in [89].

In Black-box testing, some work [84] has been done which uses supervised learning procedures. As far as we know, no previous work has addressed online testing with learning in the context of Model Based Testing. The main intuition behind our algorithm is similar to an anti-ant approach [75] used for test case generation form UML diagrams. From the game point of view, the online testing problem is a $1\frac{1}{2}$-player game. It is known that $1\frac{1}{2}$-player games are Markov Decision Processes [26]. The view of finite explorations of model programs for offline test case generation as negative total reward Markov decision problems with infinite horizon are studied in [19].

4.6 Open Problems

One of the interesting areas that is also practically very relevant is to gain better understating of approaches for online testing that learn from model-coverage that uses abstractions. The experiments reported in Section 4.4 exploited that idea to a certain extent by using state

and action abstraction through the `IModelStepper` interface, but the general technique and theory need to be developed further. Such abstraction functions can either be user-provided [61, 23] or automatically generated from program text similar to iterative refinement [89].

Currently we have an implementation of the presented algorithm using a modeling library developed in C#. As a short-term goal, we are working on a more detailed report where we are considering larger case studies.

The algorithm can also be adapted to run without a model, just as a semi-random (stress) testing tool of implementations. In that case the history of used actions is kept solely based on the test runs of the implementation. In this case, erroneous behaviors would for example manifest themselves through unexpected exceptions thrown by the implementation, rather than trough conformance violations.

Part III

Qualitative Abstraction

Chapter 5

Game-Based Three Valued Abstraction

5.1 Introduction

Games provide a computational model that is widely used in applications ranging from controller design, to modular verification, to system design and analysis. The main obstacle to the practical application of games to design and control problems lies in very large state space of games modeling real-life problems. In system verification, one of the main methods for coping with large-size problems is *abstraction*. An abstraction is a simplification of the original system model. To be useful, an abstraction should contain sufficient detail to enable the derivation of the desired system properties, while being succinct enough to allow for efficient analysis. Finding an abstraction that is simultaneously informative and succinct is a difficult task, and the most successful approaches rely on the automated construction, and gradual refinement, of abstractions. Given a system and the property, a coarse initial abstraction is constructed: this initial abstraction typically preserves only the information about the system that is most immediately involved in the property, such as the values of the state variables mentioned in the property. This initial abstraction is then gradually, and automatically, refined, until the

property can be proved or disproved, in the case of a verification problem, or until the property can be analyzed to the desired level of accuracy, in case of a quantitative problem.

One of the most successful techniques for automated abstraction refinement is the technique of *counterexample-guided refinement,* or CEGAR [9, 31, 13]. According to this technique, given a system abstraction, we check whether the abstraction satisfies the property. If the answer is affirmative, we are done. Otherwise, the check yields an *abstract counterexample,* encoding a set of "suspect" system behaviors. The abstract counterexample is then further analyzed, either yielding a concrete counterexample (a proof that the property does not hold), or yielding a refined abstraction, in which that particular abstract counterexample is no longer present. The process continues until either a concrete counterexample is found, or until the property can be shown to hold (i.e., no abstract counterexamples are left). The appeal of CEGAR lies in the fact that it is a fully automatic technique, and that the abstraction is refined on-demand, in a property-driven fashion, adding just enough detail as is necessary to perform the analysis. The CEGAR technique has been extended to games in *counterexample-guided control* [62].

We propose here an alternative technique to CEGAR for refining game abstractions: namely, we propose to use *three-valued* analysis [97, 98, 41] in order to guide abstraction refinement for games. The technique is suited to *reachability games,* where the goal is to reach a set of target states, and to *safety properties,* where the goal is to stay always in a set of "safe" states. The technique works as follows. Given a game abstraction, we analyze it in three-valued fashion, computing the set of *must-win* states, which are known to satisfy the reachability or safety property, and the set of *never-win* states, which are known not to satisfy the property; the remaining states, for which the satisfaction is unknown, are called the *may-win* states. If this three-valued analysis yields the desired information (for example, showing the existence of an initial state with a given property), the analysis terminates. Otherwise, we refine the abstraction in a way that reduces the number of *may-win* states. The abstraction refinement proceeds in a

property-dependent way. For reachability properties, we refine the abstraction at the may-must border, splitting a may-win abstract state into two parts, one of which is known to satisfy the property (and that will become a must-win state). For the dual case of safety properties, the refinement occurs at the may-never border.

Our proposed three-valued abstraction refinement technique can be implemented in fully symbolic fashion, and it can be applied to games with both finite and infinite state spaces. The technique terminates whenever the game has a finite *region algebra* (a partition of the state space) that is closed with respect to Boolean and controllable-predecessor operators [45]: this is the case for many important classes of games, among which timed games [77, 40]. Furthermore, we show that the technique never performs unnecessary refinements: the final abstraction is never finer than a region algebra that suffices for proving the property.

In its aim of reducing the number of may-states, our technique is related to the three-valued abstraction refinement schemes proposed for CTL and transition systems in [97, 98]. Differently from these approaches, however, we avoid the explicit construction of the tree-valued transition relation of the abstraction, relying instead on *may* and *must* versions of the controllable predecessor operators. Our approach provides precision and efficiency benefits. In fact, to retain full precision, the must-transitions of a three-valued model need to be represented as hyper-edges, rather than normal edges [98, 41, 99]; in turn, hyper-edges are computationally expensive both to derive and to represent. The may and must predecessor operators we use provide the same precision as the hyper-edges, without the associated computational penalty. For a similar reason, we show that our three-valued abstraction refinement technique is superior to the CEGAR technique of [62], in the sense that it can prove a given property with an abstraction that never needs to be finer, and that can often be coarser. Again, the advantage is due to the fact that [62] represents player-1 moves in the abstract model via must-edges, rather than must hyper-edges. A final benefit of avoiding the explicit construction of the abstract model,

relying instead on predecessor operators, is that the resulting technique is simpler to present, and simpler to implement. On the other side, we remark that the techniques of [62] extend easily to parity goals, whereas the refinement scheme we propose can be extended, but only at the price of cumbersome bookkeeping.

While we present the technique for games, the technique also yields a three-valued abstraction refinement scheme for the verification of safety and reachability properties of transition systems.

5.2 Definitions

5.2.1 Game Abstractions

An *abstraction* V of a game structure $G = \langle S, \lambda, \delta \rangle$ consists of a set $V \subseteq 2^{2^S \setminus \emptyset}$ of *abstract states:* each abstract state $v \in V$ is a non-empty subset $v \subseteq S$ of concrete states. We require $\bigcup V = S$. For subsets $T \subseteq S$ and $U \subseteq V$, we write:

$$U\downarrow = \bigcup_{u \in U} u \qquad T\uparrow_V^m = \{v \in V \mid v \cap T \neq \emptyset\} \qquad T\uparrow_V^M = \{v \in V \mid v \subseteq T\} \tag{5.1}$$

Thus, for a set $U \subseteq V$ of abstract states, $U\downarrow$ is the corresponding set of concrete states. For a set $T \subseteq S$ of concrete states, $T\uparrow_V^m$ and $T\uparrow_V^M$ are the set of abstract states that constitute over and under-approximations of the concrete set T. The following result follows immediately from the definitions (5.1).

Lemma 1 *For all sets $T \subseteq S$, we have:*

$$T\uparrow_V^M \subseteq T\uparrow_V^m, \qquad (T\uparrow_V^M)\downarrow \subseteq T \subseteq (T\uparrow_V^m)\downarrow.$$

We say that the abstraction V of a state-space S is *precise* for a set $T \subseteq S$ of states if $T\uparrow_V^m = T\uparrow_V^M$.

5.2.2 Abstract Controllable Predecessor Operators

In order to allow the solution of games on the abstract state space V, we introduce abstract versions of $\mathrm{Cpre}_.$. As multiple concrete states may correspond to the same abstract state, we cannot compute, on the abstract state space, a precise analogous of $\mathrm{Cpre}_.$. Thus, for player $i \in \{1,2\}$, we define two abstract operators: the *may* operator $\mathrm{Cpre}_i^{V,m} : 2^V \mapsto 2^V$, which constitutes an over-approximation of Cpre_i, and the *must* operator $\mathrm{Cpre}_i^{V,M} : 2^V \mapsto 2^V$, which constitutes an under-approximation of Cpre_i [41]. We let, for $U \subseteq V$ and $i \in \{1,2\}$:

$$\mathrm{Cpre}_i^{V,m}(U) = \mathrm{Cpre}_i(U\downarrow)\uparrow_V^m \qquad \mathrm{Cpre}_i^{V,M}(U) = \mathrm{Cpre}_i(U\downarrow)\uparrow_V^M. \tag{5.2}$$

By the results of [41], we have the duality

$$\mathrm{Cpre}_i^{V,M}(U) = V \setminus \mathrm{Cpre}_{\sim i}^{V,m}(V \setminus U). \tag{5.3}$$

The fact that $\mathrm{Cpre}_.^{V,m}$ and $\mathrm{Cpre}_.^{V,M}$ are over and under-approximations of the concrete predecessor operator is made precise by the following observation, which follows directly from Lemma 1: for all $U \subseteq V$ and $i \in \{1,2\}$, we have

$$\mathrm{Cpre}_i^{V,M}(U)\downarrow \subseteq \mathrm{Cpre}^i(U\downarrow) \subseteq \mathrm{Cpre}_i^{V,m}(U)\downarrow . \tag{5.4}$$

5.3 Reachability and Safety Games

We present our three-valued abstraction refinement technique by applying it first to the simplest games: reachability and safety games. It is convenient to present the arguments first for reachability games; the results for safety games are then obtained by duality.

5.3.1 Reachability Games

Our three-valued abstraction-refinement scheme for reachability proceeds as follows. We assume we are given a game $G = \langle S, \lambda, \delta \rangle$, together with an initial set $\theta \subseteq S$ and a final set

$T \subseteq S$, and an abstraction V for G that is precise for θ and T. The question to be decided is: $\theta \cap \langle 1 \rangle \Diamond T = \emptyset$?

The algorithm proceeds as follows. Using the may and must predecessor operators, we compute respectively the set W_1^m of *may-winning* abstract states, and the set W_1^M of *must-winning* abstract states. If $W_1^m \cap \theta \uparrow_V^m = \emptyset$, then the algorithm answers the question No; if $W_1^M \cap \theta \uparrow_V^M \neq \emptyset$, then the algorithm answers the question Yes. Otherwise, the algorithm picks an abstract state v such that

$$v \in (W_1^m \setminus W_1^M) \cap \mathrm{Cpre}_1^{V,m}(W_1^M). \tag{5.5}$$

Such a state lies at the border between W_1^M and W_1^m. The state v is split into two abstract states v_1 and v_2, where:

$$v_1 = v \cap \mathrm{Cpre}_1(W_1^M \downarrow) \qquad\qquad v_2 = v \setminus \mathrm{Cpre}_1(W_1^M \downarrow).$$

As a consequence of (5.5), we have that $v_1, v_2 \neq \emptyset$. The algorithm is given in detail as Algorithm 2. We first state the partial correctness of the algorithm, postponing the analysis of its termination to Section 5.3.4.

Lemma 2 *After Step 3 of Algorithm 2, we have* $W_1^M \downarrow \subseteq \langle 1 \rangle \Diamond T \subseteq W_1^m \downarrow$.

Proof: The result follows from (5.4), and from the monotonicity of the μ-calculus operators appearing in Steps 2 and 3 of Algorithm 2. ∎

Lemma 3 *If Step 7 of Algorithm 2 is reached, there is at least one region* $v \in (W_1^m \setminus W_1^M) \cap Cpre_1^{V,m}(W_1^M)$.

Proof: First, notice that since the algorithm did not terminate at Step 4 or Step 5, it must be $W_1^m \cap \theta \uparrow_V^m \neq \emptyset$ and $W_1^M \cap \theta \uparrow_V^M = \emptyset$, which by the previous lemma implies $W_1^M \subsetneq W_1^m$.

Algorithm 2 3-valued Abstraction Refinement for Reachability Games

Input: A concrete game structure $G = \langle S, \lambda, \delta \rangle$, a set of initial states $\theta \subseteq S$, a set of target states $T \subseteq S$, and an abstraction $V \subseteq 2^{2^S \setminus \emptyset}$ that is precise for θ and T.

Output: Yes if $\theta \cap \langle\langle 1 \rangle\rangle \Diamond T \neq \emptyset$, and No otherwise.

1. **while** true **do**
2. $\quad W_1^M := \mu Y.(T\uparrow_V^M \cup \mathrm{Cpre}_1^{V,M}(Y))$
3. $\quad W_1^m := \mu Y.(T\uparrow_V^m \cup \mathrm{Cpre}_1^{V,m}(Y))$
4. $\quad$ **if** $W_1^m \cap \theta\uparrow_V^m = \emptyset$ **then return** No
5. $\quad$ **else if** $W_1^M \cap \theta\uparrow_V^M \neq \emptyset$ **then return** Yes
6. $\quad$ **else**
7. $\quad\quad$ choose $v \in (W_1^m \setminus W_1^M) \cap \mathrm{Cpre}_1^{V,m}(W_1^M)$
8. $\quad\quad$ let $v_1 := v \cap \mathrm{Cpre}_1(W_1^M\downarrow)$ and $v_2 := v \setminus v_1$
9. $\quad\quad V := (V \setminus \{v\}) \cup \{v_1, v_2\}$
10. $\quad$ **end if**
11. **end while**

From the fact that W_1^m is a least fix-point, we have $W_1^m = \mu Y.(W_1^M \cup \mathrm{Cpre}_1^{V,m}(Y))$. Thus, there must be some $v \in W_1^m \setminus W_1^M$ with $v \in \mathrm{Cpre}_1^{V,m}(W_1^M)$. ∎

Lemma 4 *The sets v_1 and v_2 computed at Step 8 of Algorithm 2 are both non-empty.*

Proof: Consider $v \in W_1^m \setminus W_1^M$ with $v \in \mathrm{Cpre}_1^{V,m}(W_1^M)$. For $v_1 = v \cap \mathrm{Cpre}_1(W_1^M\downarrow)$, we have $v_1 \neq \emptyset$, for otherwise $v_1 \notin \mathrm{Cpre}_1^{V,m}(W_1^M)$. Furthermore, we have $v_1 \subsetneq v$, for else we would have $v \in \mathrm{Cpre}_1^{V,M}(W_1^M)$, contradicting the fact that W_1^M is the fix-point computed at Step 2. ∎

Theorem 1 (partial correctness) *Algorithm 2 can be executed without errors. Moreover:*

1. *if the algorithm terminates with answer Yes, then* $\theta \cap \langle 1 \rangle \Diamond T \neq \emptyset$;

2. *if the algorithm terminates with answer No, then* $\theta \cap \langle 1 \rangle \Diamond T = \emptyset$.

Proof: The only statement that could result in an error in the execution of Algorithm 2 is the choice of v at Step 7; Lemma 3 ensures that the error never arises. The fact that if the algorithm terminates, it returns a correct answer, is a consequence of Lemma 2. ∎

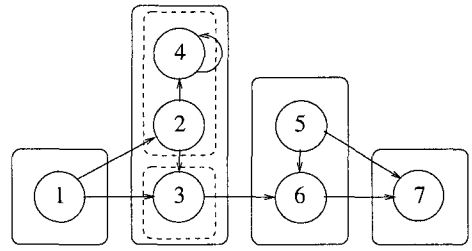


Figure 5.1: Three-Valued Abstraction Refinement in Reachability Game

Sufficient conditions for the termination of the algorithm are presented later, in Section 5.3.4

Example. As an example, consider the game G illustrated in Figure 5.1. The state space of the game is $S = \{1, 2, 3, 4, 5, 6, 7\}$, and the abstract state space is $V = \{v_a, v_b, v_c, v_d\}$, as indicated in the figure; the player-2 states are $S_2 = \{2, 3, 4\}$. We consider $\theta = \{1\}$ and $T = \{7\}$. After Steps 2 and 3 of Algorithm 2, we have $W_1^m = \{v_a, v_b, v_c, v_d\}$, and $W_1^M = \{v_c, v_d\}$. Therefore, the algorithm can answer neither No in Steps 4, nor Yes in Step 5, and proceeds to refine the abstraction. In Step 7, the only candidate for splitting is $v = v_b$, which is split into $v_1 = v_b \cap \mathrm{Cpre}_1(W_1^M\downarrow) = \{3\}$, and $v_2 = v_b \setminus v_1 = \{2, 4\}$. It is easy to see that at the next iteration of the analysis, v_1 and v_a are added to W_1^M, and the algorithm returns the answer Yes.

5.3.2 An Improved Algorithm for Reachability

Algorithm 2 can be improved by avoiding the full re-computation of the sets W_1^M and W_1^m at each abstraction refinement. Once we obtain v_1 and v_2 as in Step 8, we can set $W := W_1^M \cup \{v_1\}$, and we can compute for the next iteration:

$$W_1^M := \mu Y.(W \cup \mathrm{Cpre}_1^{V,M}(Y)) W_1^m := \mu Y.(W \cup \mathrm{Cpre}_1^{V,m}(Y))$$

The resulting algorithm is presented as Algorithm 3.

5.3.3 Safety Games

We next consider a safety game specified by a target $T \subseteq S$, together with an initial condition $\theta \subseteq S$. Given an abstraction V that is precise for T and θ, the goal is to answer the question of whether $\theta \cap \langle 1 \rangle \Box T = \emptyset$. As for reachability games, we begin by computing the set W_1^m of may-winning states, and the set W_1^M of must-winning states. Again, if $W_1^m \cap \theta \uparrow_V^m = \emptyset$, we answer No, and if $W_1^M \cap \theta \uparrow_V^M \neq \emptyset$, we answer Yes. In safety games, unlike in reachability games, we cannot split abstract states at the may-must boundary. For reachability games, a may-state can only win by reaching the goal T, which is contained in $W_1^M \downarrow$: hence, we refine the may-must border. In a safety game with objective $\Box T$, on the other hand, we have $W_1^m \downarrow \subseteq T$, and a state in $W_1^m \downarrow$ can be winning even if it never reaches $W_1^M \downarrow$ (which indeed can be empty if the abstraction is too coarse). Thus, to solve safety games, we split abstract states at the may-losing boundary, that is, at the boundary between W_1^m and its complement. This can be explained by the fact that $\langle 1 \rangle \Box T = S \setminus \langle 2 \rangle \Diamond \neg T$: the objectives $\Box T$ and $\Diamond \neg T$ are dual. Therefore, we adopt for $\Box T$ the same refinement method we would adopt for $\Diamond \neg T$, and the may-must boundary for $\langle 2 \rangle \Diamond \neg T$ is the may-losing boundary for $\langle 1 \rangle \Box T$. This yields Algorithm 4.

Theorem 2 (partial correctness) *Algorithm 4 can be executed without errors. Moreover:*

1. *if the algorithm terminates with answer Yes, then* $\theta \cap \langle 1 \rangle \Box T \neq \emptyset$;

2. *if the algorithm terminates with answer No, then* $\theta \cap \langle 1 \rangle \Box T = \emptyset$.

Proof : The theorem can be proved by noting that the goals $\Box T$ and $\Diamond \neg T$ are dual, and by noting that from (5.3) we have:

$$\nu Y.(T \uparrow_V^M \cap \mathrm{Cpre}_1^{V,M}(Y)) = V \setminus \mu Y.((S \setminus T) \uparrow_V^M \cup \mathrm{Cpre}_1^{V,M}(Y))$$

$$\nu Y.(T \uparrow_V^m \cap \mathrm{Cpre}_1^{V,m}(Y)) = V \setminus \mu Y.((S \setminus T) \uparrow_V^m \cup \mathrm{Cpre}_1^{V,m}(Y)) \; .$$

Thus, the Algorithm 4 is the dual of Algorithm 2, and its correctness can be proved in analogous fashion. ∎

We note that it is possible to obtain a more efficient version of Algorithm 4 by performing a dual transformation to the one that yielded Algorithm 3. Precisely, before Step 1, we let $W := (T \uparrow_V^m)$; the fix-points at Steps 2 and 3 are computed via $W_1^M := \nu Y.(W \cap \mathrm{Cpre}_1^{V,M}(Y))$ and $W_1^m := \nu Y.(W \cap \mathrm{Cpre}_1^{V,m}(Y))$; and after Step 8 we set $W := W_1^m \setminus \{v_1\}$.

5.3.4 Termination

We present a condition that ensures termination of Algorithms 2 and 4 (and thus also Algorithm 3). The condition states that, if there is a finite algebra of regions (sets of concrete states) that is closed under Boolean operations and controllable predecessor operators, and that is precise for the sets of initial and target states, then (i) Algorithms 2 and 4 terminate, and (ii) the algorithms never produce abstract states that are finer than the regions of the algebra (guaranteeing that the algorithms do not perform unnecessary work). Formally, a *region algebra* for a game $G = \langle S, \lambda, \delta \rangle$ is an abstraction U such that:

- U is closed under Boolean operations: for all $u_1, u_2 \in U$, we have $u_1 \cup u_2 \in U$ and $S \setminus u_1 \in U$.

- U is closed under controllable predecessor operators: for all $u \in U$, we have $\mathrm{Cpre}_1(u) \in U$ and $\mathrm{Cpre}_2(u) \in U$.

Theorem 3 (termination) *Consider a game G with a finite region algebra U. Assume that Algorithm 2 or 4 are called with arguments G, θ, T, with $\theta, T \in U$, and with an initial abstraction $V \subseteq U$. Then, the following assertions hold for both algorithms:*

1. *The algorithms, during their executions, produce abstract states that are all members of the algebra U.*

2. *The algorithms terminate.*

Proof : Let us prove the theorem for the reachability game. The proof for safety game can be easily obtained by duality.

First, note that due to the closure properties of the region algebra U, the algorithm computes entirely with regions in U: precisely, variables are only assigned regions of U. This yields the first assertion of the theorem.

The termination of Algorithm 2 can be proved by the following argument. At each refinement loop, the algorithm decreases the size of the uncertainty region $W_1^m \setminus W_1^M$, since the set v_1 computed in Step 8 will belong to W_1^M in the following iteration. As the region algebra U is finite, within a finite number of refinements the uncertainty region will be empty, and the algorithm will return either Yes or No. ∎

Many games, including timed games, have the finite region algebras mentioned in the above theorem [77, 45, 40].

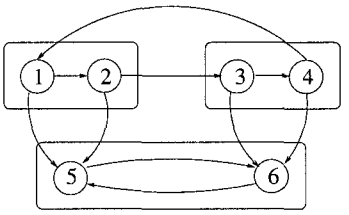


Figure 5.2: Safety game, with objective $\Box T$ for $T = \{1, 2, 3, 4\}$.

5.3.5 Comparison with Counterexample-Guided Control

It is instructive to compare our three-valued refinement approach with the *counterexample-guided control* approach of [62]. In [62], an abstract game structure is constructed and analyzed. The abstract game contains *must* transitions for player 1, and *may* transitions for player 2. Every counterexample to the property (spoiling strategy for player 2) found in the abstract game is analyzed in the concrete game. If the counterexample is real, the property is disproved; If the counterexample is spurious, it is ruled out by refining the abstraction. The process continues until either the property is disproved, or no abstract counterexamples is found, proving the property.

The main advantage of our proposed three-valued approach over counterexample-guided control is, somewhat paradoxically, that we do not explicitly construct the abstract game. It was shown in [98, 41] that, for a game abstraction to be fully precise, the *must* transitions should be represented as hyper-edges (an expensive representation, space-wise). In the counterexample-guided approach, instead, normal *must* edges are used: the abstract game representation incurs a loss of precision, and more abstraction refinement steps may be needed than with our proposed three-valued approach. This is best illustrated with an example.

Example. Consider the game structure depicted in Figure 5.2. The state space is $S = \{1, 2, 3, 4, 5, 6\}$, with $S_1 = \{1, 2, 3, 4\}$ and $S_2 = \{5, 6\}$; the initial states are $\theta = \{1, 2\}$.

We consider the safety objective $\Box T$ for $T = \{1, 2, 3, 4\}$. We construct the abstraction $V = \{v_a, v_b, v_c\}$ precise for θ and T, as depicted. In the counterexample-guided control approach of [62], hyper-must transitions are not considered in the construction of the abstract model, and the transitions between v_a and v_b are lost: the only transitions from v_a and v_b lead to v_c. Therefore, there is a spurious abstract counterexample tree $v_a \rightarrow v_c$; ruling it out requires splitting v_a into its constituent states 1 and 2. Once this is done, there is another spurious abstract counterexample $2 \rightarrow v_b \rightarrow v_c$; ruling it out requires splitting v_b in its constituent states. In contrast, in our approach we have immediately $W_1^M = \{v_a, v_b\}$ and $v_a, v_b \in \mathrm{Cpre}_1^{V,M}(\{v_a, v_b\})$, so that no abstraction refinement is required. ∎

The above example illustrates the fact that the counterexample-guided control approach of [62] may require a finer abstraction than our three-valued refinement approach, to prove a given property. On the other hand, it is easy to see that if an abstraction suffices to prove a property in the counterexample-guided control approach, it also suffices in our three-valued approach: the absence of abstract counterexamples translates directly in the fact that the states of interest are must-winning.

5.4 Symbolic Implementation

We now present a concrete symbolic implementation of our abstraction scheme. We chose a simple symbolic representation for two-player games; while the symbolic game representations encountered in real verification systems (see, e.g.,[38, 39]) are usually more complex, the same principles apply.

5.4.1 Approximate Abstraction Refinement Schemes

While the abstraction refinement scheme above is fairly general, it makes two assumptions that may not hold in a practical implementation:

- it assumes that we can compute $\mathrm{Cpre}_*^{V,m}$ and $\mathrm{Cpre}_*^{V,M}$ of (5.2) precisely;
- it assumes that, once we pick an abstract state v to split, we can split it into v_1 and v_2 precisely, as outlined in Algorithms 2 and 4.

In fact, both assumptions can be related, yielding a more widely applicable abstraction refinement algorithm for two-player games. We present the modified algorithm for the reachability case only; the results can be easily extended to the dual case of safety objectives. Our starting point consists in approximate versions $\mathrm{Cpre}_i^{V,m+}, \mathrm{Cpre}_i^{V,M-} : 2^V \mapsto 2^V$ of the operators $\mathrm{Cpre}_i^{V,m}$, $\mathrm{Cpre}_i^{V,M}$, for $i \in \{1,2\}$. We require that, for all $U \subseteq V$ and $i \in \{1,2\}$, we have:

$$\mathrm{Cpre}_i^{V,m}(U) \subseteq \mathrm{Cpre}_i^{V,m+}(U) \qquad \mathrm{Cpre}_i^{V,M-}(U) \subseteq \mathrm{Cpre}_i^{V,M}(U)\ . \tag{5.6}$$

With these operators, we can phrase a new, approximate abstraction scheme for reachability, given in Algorithm 5. The use of the approximate operators means that, in Step 8, we can be no longer sure that both $v_1 \neq \emptyset$ and $v \setminus v_1 \neq \emptyset$. If the "precise" split of Step 8 fails, we resort instead to an arbitrary split (Step 10). The following theorem states that the algorithm essentially enjoys the same properties of the "precise" Algorithms 2 and 4.

Lemma 5 *At Step 4 of Algorithm 5, we have* $W_1^{M-}\downarrow \subseteq \langle 1\rangle\Diamond T \subseteq W_1^{m+}\downarrow$.

Proof: From Equation 5.6, we have : $\mathrm{Cpre}_i^{V,m}(U) \subseteq \mathrm{Cpre}_i^{V,m+}(U)$ and $\mathrm{Cpre}_i^{V,M-}(U) \subseteq \mathrm{Cpre}_i^{V,M}(U)$ for all $U \subseteq V$ and $i \in \{1,2\}$. For reachability game the approximate and accurate must winning set is obtained by the respective fix-point formulas $W_1^{M-} = \mu Y.(T\uparrow_V^M \cup \mathrm{Cpre}_1^{V,M-}(Y)$ and $W_1^M = \mu Y.(T\uparrow_V^M \cup \mathrm{Cpre}_1^{V,M}(Y)$. Since we start with the same initial set and

in each iteration $\mathrm{Cpre}_i^{V,M-}(Y) \subseteq \mathrm{Cpre}_i^{V,M}(Y)$ holds, it is obvious that $W_1^{M-} \subseteq W_1^M$. Similar arguments will prove that $W_1^m \subseteq W_1^{m+}$. After we combine these two results with Lemma 2, we obtain $W_1^{M-}\downarrow \subseteq \langle 1\rangle\Diamond T \subseteq W_1^{m+}\downarrow$. ∎

Theorem 4 *The following assertions hold.*

1. Correctness. *If Algorithm 5 terminates, it returns the correct answer.*

2. Termination. *Assume that Algorithm 5 is given as input a game G with a finite region algebra U, and arguments $\theta, T \in U$, as well as with an initial abstraction $V \subseteq U$. Assume also that the region algebra U is closed with respect to the operators $Cpre_i^{V,M-}$ and $Cpre_i^{V,m+}$, for $i \in \{1,2\}$, and that Step 10 of Algorithm 5 splits the abstract states in regions in U. Then, (a) Algorithm 5 produces only abstract states in U in the course of its execution and (b) it terminates within finite number of refinements.*

Proof: (1) The correctness can be proved as Theorem 1 using Lemma 5. (2)(a) The fact that Algorithm 5 produces only regions in U follows from the closure of U, and by inspection of the operations performed by the algorithm. (b) The termination of Algorithm 5 follows again from the finiteness of U, and from the fact that at each iteration, the uncertainty region shrinks. ∎

5.4.2 Symbolic Game Structures

To simplify the presentation, we assume that all variables are Boolean. For a set X of Boolean variables, we denote by $\{(X)$ the set of propositional formulas constructed from the variables in X, the constants *true* and *false*, and the propositional connectives $\neg, \wedge, \vee, \rightarrow$. We denote with $\phi[\psi/x]$ the result of replacing all occurrences of the variable x in ϕ with a formula ψ. For $\phi \in \{(X)$ and $x \in X$, we write $\{^{\forall}_{\exists}\}x.\phi$ for $\phi[true/x]\{^{\wedge}_{\vee}\}\phi[false/x]$. We extend this notation

to sets $Y = \{y_1, y_2, \ldots, y_n\}$ of variables, writing $\forall Y.\phi$ for $\forall y_1.\forall y_2.\cdots\forall y_n.\phi$, and similarly for $\exists Y.\phi$. For a set X of variables, we also denote by $X' = \{x' \mid x \in X\}$ the corresponding set of *primed* variables; for $\phi \in \{(X)$, we denote ϕ' the formula obtained by replacing every $x \in X$ with x'.

A *state* s over a set X of variables is a truth-assignment $s : X \mapsto \{\mathrm{T}, \mathrm{F}\}$ for the variables in X; we denote with $S[X]$ the set of all such truth assignments. Given $\phi \in \{(X)$ and $s \in S[X]$, we write $s \models \phi$ if ϕ holds when the variables in X are interpreted as prescribed by s, and we let $[\![\phi]\!]_X = \{s \in S[X] \mid s \models \phi\}$. Given $\phi \in \{(X \cup X')$ and $s, t \in S[X]$, we write $(s,t) \models \phi$ if ϕ holds when $x \in X$ has value $s(x)$, and $x' \in X'$ has value $t(x)$. When X, and thus the state space $S[X]$, are clear from the context, we equate informally formulas and sets of states. These formulas, or sets of states, can be manipulated with the help of symbolic representations such as BDDs [21]. A *symbolic game structure* $G_S = \langle X, \Lambda_1, \Delta\rangle$ consists of the following components:

- A set of Boolean variables X.
- A predicate $\Lambda_1 \in \{(X)$ defining when it is player 1's turn to play. We define $\Lambda_2 = \neg\Lambda_1$.
- A transition function $\Delta \in \{(X \cup X')$, such that for all $s \in S[X]$, there is some $t \in S[X]$ such that $(s,t) \models \Delta$.

A symbolic game structure $G_S = \langle X, \Lambda_1, \Delta\rangle$ induces a (concrete) game structure $G = \langle S, \lambda, \delta\rangle$ via $S = S[X]$, and for $s, t \in S$, $\lambda(s) = 1$ iff $s \models \Lambda_1$, and $t \in \delta(s)$ iff $(s,t) \models \Delta$. Given a formula $\phi \in \{(X)$, we have

$$\mathrm{Cpre}_1([\![\phi]\!]_X) = [\![\Big(\Lambda_1 \wedge \exists X'.(\Delta \wedge \phi')\Big) \vee \Big(\neg\Lambda_1 \wedge \forall X'.(\Delta \rightarrow \phi')\Big)]\!]_X.$$

5.4.3 Symbolic Abstractions

We specify an abstraction for a symbolic game structure $G_S = \langle X, \Lambda_1, \Delta\rangle$ via a subset $X^a \subseteq X$ of its variables: the idea is that the abstraction keeps track only of the values of

the variables in X^a; we denote by $X^c = X \setminus X^a$ the concrete-only variables. We assume that $\Lambda_1 \in \{(X^a)$, so that in each abstract state, only one of the two players can move (in other words, we consider *turn-preserving* abstractions [41]). With slight abuse of notation, we identify the abstract state space V with $S[X^a]$, where, for $s \in S[X]$ and $v \in V$, we let $s \in v$ iff $s(x) = v(x)$ for all $x \in X^a$. On this abstract state space, the operators $\mathrm{Cpre}_1^{V,m}$ and $\mathrm{Cpre}_1^{V,M}$ can be computed symbolically via the corresponding operators $\mathrm{SCpre}_1^{V,m}$ and $\mathrm{SCpre}_1^{V,M}$, defined as follows. For $\phi \in \{(X^a)$,

$$\mathrm{SCpre}_1^{V,m}(\phi) = \exists X^c.\left(\left(\Lambda_1 \wedge \exists X'.(\Delta \wedge \phi')\right) \vee \left(\Lambda_2 \wedge \forall X'.(\Delta \rightarrow \phi')\right)\right) \tag{5.7}$$

$$\mathrm{SCpre}_1^{V,M}(\phi) = \forall X^c.\left(\left(\Lambda_1 \wedge \exists X'.(\Delta \wedge \phi')\right) \vee \left(\Lambda_2 \wedge \forall X'.(\Delta \rightarrow \phi')\right)\right) \tag{5.8}$$

The above operators correspond exactly to (5.2). Alternatively, we can abstract the transition formula Δ, defining:

$$\Delta_{X^a}^m = \exists X^c.\exists X^{c\prime}.\Delta \qquad \Delta_{X^a}^M = \forall X^c.\exists X^{c\prime}.\Delta\ .$$

These abstract transition relations can be used to compute approximate versions $\mathrm{SCpre}_1^{V,m+}$ and $\mathrm{SCpre}_1^{V,M-}$ of the controllable predecessor operators of (5.7), (5.8):

$$\mathrm{SCpre}_1^{V,m+}(\phi) = \left(\left(\Lambda_1 \wedge \exists X^{a\prime}.(\Delta_{X^a}^m \wedge \phi')\right) \vee \left(\Lambda_2 \wedge \forall X^{a\prime}.(\Delta_{X^a}^m \rightarrow \phi')\right)\right)$$

$$\mathrm{SCpre}_1^{V,M-}(\phi) = \left(\left(\Lambda_1 \wedge \exists X^{a\prime}.(\Delta_{X^a}^M \wedge \phi')\right) \vee \left(\Lambda_2 \wedge \forall X^{a\prime}.(\Delta_{X^a}^M \rightarrow \phi')\right)\right)$$

These operators, while approximate, satisfy the conditions (5.6), and can thus be used to implement symbolically Algorithm 5.

5.4.4 Symbolic Abstraction Refinement

We replace the abstraction refinement step of Algorithms 2, 4, and 5 with a step that adds a variable $x \in X^c$ to the set X^a of variables present in the abstraction. The challenge is to

choose a variable x that increases the precision of the abstraction in a useful way. To this end, we follow an approach inspired directly by [31].

Denote by $v \in S[X^a]$ the abstract state that Algorithms 5 chooses for splitting at Step 7, and let $\psi_1^{M-} \in \{(X^a)$ be the formula defining the set W_1^{M-} in the same algorithm. We choose $x \in X^c$ so that there are at least two states $s_1, s_2 \in v$ that differ only for the value of x, and such that $s_1 \models \mathrm{SCpre}_1^{V,m+}(\psi_1^{M-})$ and $s_2 \not\models \mathrm{SCpre}_1^{V,m+}(\psi_1^{M-})$. Thus, the symbolic abstraction refinement algorithm first searches for a variable $x \in X^c$ for which the following formula is true:

$$\exists (X^c \setminus x).\left(\left(\chi_v \rightarrow \left(x \equiv \mathrm{SCpre}_1^{V,m+}(\psi_1^{M-})\right)\right) \vee \left(\chi_v \rightarrow \left(x \not\equiv \mathrm{SCpre}_1^{V,m+}(\psi_1^{M-})\right)\right)\right),$$

where χ_v is the *characteristic formula* of v:

$$\chi_v = \bigwedge\left\{x \mid x \in X^a.v(x) = \mathrm{T}\right\} \wedge \bigwedge\left\{\neg x \mid x \in X^a.v(x) = \mathrm{F}\right\}.$$

If no such variable can be found, due to the approximate computation of $\mathrm{SCpre}_1^{V,m+}$ and $\mathrm{SCpre}_1^{V,M-}$, then $x \in X^c$ is chosen arbitrarily. The choice of variable for Algorithm 4 can be obtained by reasoning in dual fashion.

5.5 Conclusion

We have presented a technique for the verification of game properties based on the construction, three-valued analysis, and refinement of game abstractions. The approach is suitable for symbolic implementation, and can be implemented in a relatively straightforward manner. The key insight of the approach consists on relying on three-valued versions of the usual predecessor operators to analyze a system, avoiding the construction of a three-valued transition relation, which would require an exponential blow-up in the size of the abstract system to achieve comparable precision. The method, presented here for games, is equally suited to transition systems,

where it constitutes an alternative to the classical counterexample-guided refinement technique of CEGAR [9, 31, 13].

Algorithm 3 Improved 3-valued Abstraction Refinement for Reachability Games

Input: A concrete game structure $G = \langle S, \lambda, \delta \rangle$, a set of initial states $\theta \subseteq S$, a set of target states $T \subseteq S$, and an abstraction $V \subseteq 2^{2^S \setminus \emptyset}$ that is precise for θ and T.

Output: Yes if $\theta \cap \langle 1 \rangle \Diamond T \neq \emptyset$, and No otherwise.

1. $W := T{\uparrow}_V^M$
2. **while** true **do**
3. $\quad W_1^M := \mu Y.(W \cup \mathrm{Cpre}_1^{V,M}(Y))$
4. $\quad W_1^m := \mu Y.(W \cup \mathrm{Cpre}_1^{V,m}(Y))$
5. $\quad$ **if** $W_1^m \cap \theta{\uparrow}_V^m = \emptyset$ **then return** No
6. $\quad$ **else if** $W_1^M \cap \theta{\uparrow}_V^M \neq \emptyset$ **then return** Yes
7. $\quad$ **else**
8. $\quad\quad$ choose $v \in (W_1^m \setminus W_1^M) \cap \mathrm{Cpre}_1^{V,m}(W_1^M)$
9. $\quad\quad$ let $v_1 := v \cap \mathrm{Cpre}_1(W_1^M{\downarrow})$ and $v_2 := v \setminus v_1$
10. $\quad\quad V := (V \setminus \{v\}) \cup \{v_1, v_2\}$
11. $\quad\quad W := W \cup \{v_1\}$
12. $\quad$ **end if**
13. **end while**

Algorithm 4 3-valued Abstraction Refinement for Safety Games

Input: A concrete game structure $G = \langle S, \lambda, \delta \rangle$, a set of initial states $\theta \subseteq S$, a set of target states $T \subseteq S$, and an abstraction $V \subseteq 2^{2^S \setminus \emptyset}$ that is precise for θ and T.

Output: Yes if $\theta \cap \langle 1 \rangle \Box T \neq \emptyset$, and No otherwise.

1. **while** true **do**
2. $\quad W_1^M := \nu Y.(T{\uparrow}_V^M \cap \mathrm{Cpre}_1^{V,M}(Y))$
3. $\quad W_1^m := \nu Y.(T{\uparrow}_V^m \cap \mathrm{Cpre}_1^{V,m}(Y))$
4. $\quad$ **if** $W_1^m \cap \theta{\uparrow}_V^m = \emptyset$ **then return** No
5. $\quad$ **else if** $W_1^M \cap \theta{\uparrow}_V^M \neq \emptyset$ **then return** Yes
6. $\quad$ **else**
7. $\quad\quad$ choose $v \in (W_1^m \setminus W_1^M) \cap \mathrm{Cpre}_2^{V,m}(V \setminus W_1^m)$
8. $\quad\quad$ let $v_1 := v \cap \mathrm{Cpre}_2(S \setminus (W_1^m{\downarrow}))$ and $v_2 := v \setminus v_1$
9. $\quad\quad$ let $V := (V \setminus \{v\}) \cup \{v_1, v_2\}$
10. $\quad$ **end if**
11. **end while**

Algorithm 5 Approximate 3-valued Abstraction Refinement for Reachability Games

Input: A concrete game structure $G = \langle S, \lambda, \delta \rangle$, a set of initial states $\theta \subseteq S$, a set of target states $T \subseteq S$, and an abstraction $V \subseteq 2^{2^S \setminus \emptyset}$ that is precise for θ and T.

Output: Yes if $\theta \cap \langle 1 \rangle \Diamond T \neq \emptyset$, and No otherwise.

1. **while** true **do**
2. $\quad W_1^{M-} := \mu Y.(T\uparrow_V^M \cup \mathrm{Cpre}_1^{V,M-}(Y))$
3. $\quad W_1^{m+} := \mu Y.(T\uparrow_V^m \cup \mathrm{Cpre}_1^{V,m+}(Y))$
4. $\quad$ **if** $W_1^{m+} \cap \theta\uparrow_V^m = \emptyset$ **then return** No
5. $\quad$ **else if** $W_1^{M-} \cap \theta\uparrow_V^M \neq \emptyset$ **then return** Yes
6. $\quad$ **else**
7. $\quad\quad$ choose $v \in (W_1^{m+} \setminus W_1^{M-}) \cap \mathrm{Cpre}_1^{V,m+}(W_1^{M-})$
8. $\quad\quad$ let $v_1 := v \cap \mathrm{Cpre}_1(W_1^{M-}\downarrow)$
9. $\quad\quad$ **if** $v_1 = \emptyset$ **or** $v_1 = v$
10. $\quad\quad\quad$ **then** split v arbitrarily into non-empty v_1 and v_2
11. $\quad\quad\quad$ **else** $v_2 = v \setminus v_1$
12. $\quad\quad$ **end if**
13. $\quad\quad$ let $V := (V \setminus \{v\}) \cup \{v_1, v_2\}$
14. $\quad$ **end if**
15. **end while**

Chapter 6

Interface Synthesis

6.1 Introduction

Verication of software systems is an extremely difficult problem due to the large size of program state-space. Software programs often include library functions and these functions are examples of *open systems.* The verification of such open systems becomes infeasible due to two main problems. Firstly, in order to verify a given program one needs to 'inline' the library function code and it increases the space complexity of the verification algorithms. Current formal techniques like model-checking can not handle the large state-space generated from the program variables. The second option is to verify the library functions a priori so that there is no need to inline them. For this purpose, most of the time experts write a small code containing a sequence of library functions calls (called client). The client code invokes the library functions to close the open system. The library functions are difficult to verify in the absence of exhaustive client program. Hence most of the verification approaches plug-in a client code to close the open-system.

6.1.1 Interface and Properties

The current research [64, 10, 17] avoids these two problems by applying modular verification techniques which builds a small call sequence graph, called *interface* representing the union of all client programs. The interface contains all possible call sequences which leads the library to error or illegal states. Similarly, the interface should contain all possible call sequences which avoids the error states. Henceforth the interface graph provides constrains on the use of the function calls from outside. The user can distinguish the legal call sequences from the illegal ones by simply looking at the interface. There are two immediate benefits of using the interfaces. Firstly, these interfaces are a light-weight representation of the libraries and the implementation of the library functions can be replaced by the interface. Secondly, the interfaces can be constructed without the help of any client program. The interface should be *safe* i.e. all illegal call sequences (which leads the library to the error states) will be present in the interface. The interface graph should be *permissive* i.e. all legal sequences will be present in the interface.

6.1.2 Related Work

However, there are some challenges in building succinct interfaces. The interface size can become exponential in terms of number of variables. A symbolic representation and abstraction techniques partition the state-space into a small number of regions where every region represents one node of the interface graph. Some researches apply these abstraction and symbolic techniques to obtain a small but safe and permissive interface.

The work by Alur et. al. ([10]) uses *Angluin's learning algorithm* L* to create an interface. The algorithm learns the interface language by asking membership and equivalence queries to teacher (here program). The generated interface is safe and minimal; but not permissive. They have used predicate abstraction to handle large case-studies. However, the user

needs to provide the predicates in these cases. There is no automatic abstraction refinement. The algorithm returns the minimal-size interface if the algorithm is not hit by timeout. Experimental results illustrate that even in small examples timeout occurs. The CEGAR approach by Henzinger et. al. ([64]) creates a safe and permissive interface. The size of the interface can be large enough depending on the chosen counter-example. The direct approach by Beyer et. al. ([17]) creates an interface which is safe and permissive. This method does not use abstraction; therefore the interface can become extremely large.

6.1.3 Contribution

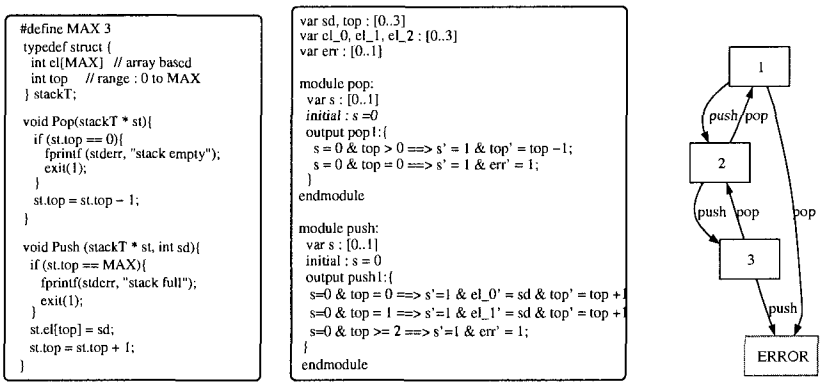
Unlike the related work, this work can also be used in unstructured or non-object oriented (C style) functions. In an object-oriented framework, every class variable is accessible to every class method and visible to the class methods. Instead, we assume that each function may also contain several local variables with limited scope within the function. Hence, we have *more general platform* to compute interface. Each of these functions can also have several sequential updates of variables, call to other functions even recursive calls to themselves. However, we compute the interface including only functions accessible to the user level.

First stage of the three-stage algorithm parses each C library function by CIL (C Intermediate Language)[82] and converts the function into TICC [37] input language. This language syntax is similar to the guarded-update language. We have implemented the next two stages in the symbolic tool TICC. The second stage computes the transition summary of each function. This modular algorithm handles each function separately including local variables within the scope. However, the space complexity of function summary becomes a bottleneck in order to compute big functions which may contain a large number of guarded-updates. Hence, we employ symbolic, three-valued abstraction-refinement algorithms. The abstraction in the summarization ensures small size, and successive refinement of the abstract states fine-tunes the

abstraction to obtain the safety and permissiveness. The last stage builds an interface graph from the abstract set of states. We describe different stages of building a symbolic, safe and permissive interface in the following example.

Example 4 (Motivating Example) *Figure 6.1(a) defines a stack data-type stackT and two functions push and pop. The data type stackT has an array of integers el of size MAX and an integer showing the top of the stack. The function pop returns error when the stack is empty i.e. top is zero. The function push returns error if the top is equal to MAX. Otherwise, the function copies sd into the el array at index top and increments the top later. Figure 6.1(b) shows how the next stage converts a small C code into a set of guarded-update rules. The variable* err *denotes the error in the library and the library goes to error state if the variable* err *equals 1. Figure 6.1(c) shows the interface graph from the set of rules. The initial state of the interface graph is* state 1 *where the stack is empty. A call to pop function from the initial state will move the library into an* ERROR *state. Similarly, a call of* push *form state* 3 *will lead the library to an error state due to the full-stack. We can note that the interface can create many legal as well as illegal sequences of stack functions. To check each of them we otherwise need a set of client programs.*

Finally, we discuss the applications of the safe and permissive interface graph. Firstly, any given client program can immediately verify with the help of the interface graph whether the function call sequence in the client leads the library to some error states. Secondly, the interface can provide an offline test-suite for a set of functions. Often the source-code of the library is unknown; however one can create a model program from the available documentation of the functions. The interface graph obtained from the model program can be used to investigate the implementation-under-test (IUT).



(a) Code (b) Rules (c) Rules

Figure 6.1: Stack Example

6.2 Algorithm

In this section we assume that the C functions are already parsed by CIL and modified into a software library module $Lib = (F_G, V_G, E, I)$. We describe the basic algorithms for abstract refinement and building interface from a given library Lib. We also provide some implementation specific optimizations.

6.2.1 Basic Algorithm

Algorithm 6 computes the interface for library $Lib = (F_G, V_G, E, I)$. The algorithm takes as input the library Lib, a set of functions $F \subseteq F_G$, an abstraction R. The first abstraction is obtained from the error set E and initial set I . Let us define $r_1 = \{s \in S_G \mid s \in E\}$, $r_2 = \{s \in S_G \mid s \notin E, s \in I\}$ and $r_3 = \{s \in S_G \mid s \notin E, s \notin I\}$. For $i \in \{1,2,3\}$, if r_i is non-empty, then we add the set to R as one of the initial abstract states. The algorithm 6 calls AbsRef for every function $f \in F$ separately to obtain a refined abstraction R w.r.t. the function. The procedure $BuildInterface$ returns an interface graph IG given the set of abstract states.

Algorithm 6 Explore(Lib, F, R)

Input: a library $Lib = (F_G, V_G, E, I)$, set of functions F, abstraction R

Output: Interface Graph IG

1. **for each** $f \in F$ **do** R:= AbsRef (R, f, E) **end for**

5. IG := BuildInterface(R, F, Lib)

Modular Verification : *AbsRef* (Algorithm 7) considers each function independently. The interface graph is an input-enabled interface automata. Hence every abstract state in the function can be checked individually for error reachability in one step function transition. The algorithm starts with the initial abstraction R and gathers a number of useful variables V_{abs} from the *support set* of the abstract states. The algorithm assigns the local abstraction R_f and the global abstraction R_G to R. The algorithm computes the must abstraction transition with respect to the abstraction R_f and the must pre-image S_M of the error set E. The set S_M determines the set of states of the function which eventually reach the error set E. The set S_M^f is a subset of S_M corresponding to the initial set of states of the function. One-step concrete pre-image S^1 of $S_M\downarrow$ checks whether any new states can be added to $S_M\downarrow$. If $S^1 \setminus S_M\downarrow$ is non-empty, then the algorithm refines then the local abstraction R_f, and the loop continues. Otherwise, it refines the global abstraction R_G with respect to S_M^f. We discuss the local and global refinements in the following paragraph. The algorithm terminates when each abstract state can either reach E or can not reach E in one function step.

Automatic Refinement : For refinement of the local abstraction R_f, the algorithm finds a variable $v \in V^f$ which is not in the set V_{abs} and is in the support set of $S_m^1 \setminus S_M\downarrow$. It adds the variable v to the 'significant set' V_{abs}. The algorithm also creates a new abstraction R^f with respect to different valuations of v. The refinement of global abstraction R_G happens after

the local abstraction reaches a fix-point and no new states can be added in the S_M set. The algorithm refines abstract states $r \in R_G$ that have non-empty intersections with both S_M^f and $\neg S_M^f$.

Building Interface : Algorithm 8 computes the interface graph from the abstraction R. The algorithm maintains a list (Q) to iterate over all abstract states. The procedure $append(Q, X)$ adds each element $x \in X$ at the end of Q. The procedure $member(Q, x)$ checks whether x is a member of Q. The procedure $removeFirst(Q)$ removes the first element from Q and returns the element. For a function f and abstraction R, the operator $Post_m^{f,R}(r)$ computes the may-successor of the region r. The algorithm adds an error-edge from the current state $curr$ to the error state Err if $Post_m^{f,R}(curr)$ and E have non-empty intersection. Otherwise, it appends the next state Q and adds a new edge $(curr, f, next)$. The algorithm terminates when the list Q is empty.

Example 5 Let us revisit the Integer Stack example (Figure 6.1) to illustrate the algorithms. We assume that the algorithm converts the library functions ($\{pop, push\}$) to the guarded-update rules (Figure 6.1(b)). Let us denote the state-space as S. Figure 6.2 illustrates the run of the *explore* algorithm(Algorithm 6). The initial abstract states r_0, r_1 and r_2 partitions the state-space S into three regions (Figure 6.2(a)). The region $r_0 = S\mid_{err=1}$ corresponds to error states, the region $r_1 = S\mid_{err=0,top=0}$ corresponds to the initial states without error states, the region $r_2 = S\mid_{err=0,top>0}$ corresponds to the non-initial non-error states.

The algorithm invokes AbsRef (Algorithm 7) for *pop* function; the *significant variables* are $V_{abs} := \{err, top\}$. Let *pop.s* denotes the local variable s at function *pop*. In the first iteration, the must predecessor S_M of error state r_0 fail to include any new states. However, predecessor of set S_M returns a set S^1 (i.e. $S\mid_{pop.s=0,top=0,err=0}$ for the function *pop*). The support set of $S^1 \setminus S_M$ contains variable *pop.s* that belongs to the set V^f, but not the set V_{abs}. The local

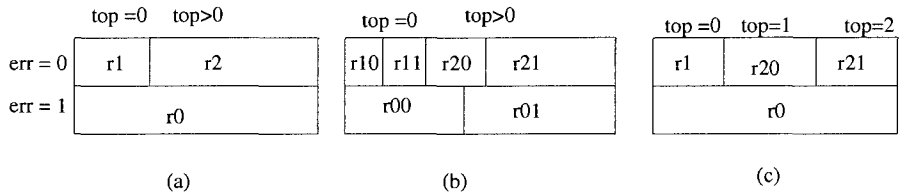


Figure 6.2: Run of the algorithm Explore on IntStack Example. (a) The initial abstraction (b) The local abstraction inside function (c) The final global abstraction.

refinement of R_f adds different valuations of variable *pop.s* (Figure 6.2(b)). The second digit of abstract states denotes the value of *pop.s* in the abstract state. In the next iteration, the must predecessor S_M becomes $\{r10, r00, r01\}$ and no new states can be added by the predecessor of set S_M. Hence the local abstraction R_f can not be further refined. The local refinement at Figure 6.2(b) can not be returned as the locally added variable *pop.s* can not reach outside the scope of function *pop*. The global set which leads the error set can be given by S_M^f. S_M^f is a subset of S_M corresponding to the initial state I_L^f. The initial set I_L^f for the pop function is $S \mid_{pop.s=0}$. Global abstraction R provides the global abstraction R_G for pop function. The algorithm refines the abstraction with respect to sets S_M^f and its compliment set $S \setminus S_M^f$. The algorithm returns with an unchanged abstraction.

Similarly, the local abstraction includes the local variable *push.s* for the *push* function. Even if the algorithm does not add any new global variable to the global refinement, it splits global abstract set r_2 with respect to the set of states (where top is 2 and err is 0) that reach error states in one push call. Figure 6.2(c) shows the final abstraction. The build interface algorithm (Algorithm 8) starts with the initial state r_1 and adds the edges in the graph (Figure 6.1(c)) until the algorithm finishes exploring every node with respect to all functions.

The interface generated by Explore algorithm is safe and permissive by construction. *AbsRef* Algorithm ensures the safety, and *BuildInterface* algorithm ensures permissiveness. The final abstraction R after calling *AbsRef* algorithms for each function $f \in F$ distinguishes error

reaching regions from the non-reaching ones. *BuildInterface* algorithm applies all functions to all abstract states ; therefore the interface graph captures all behaviors.

Theorem 1 *Explore (Algorithm 6) returns a safe and permissive interface.*

6.2.2 Implementation Optimizations

Approximate Abstract Function Summary and Predecessors: For practical purposes, we do not compute the abstract predecessor operators on the monolithic transition relations. Like [51], Equation 5.6 holds for approximate operators. The transition for a function $f \in F_G$ is represented as a number (say k) of guarded-update rules. For an abstraction $R \subseteq 2^{2^{S_f}}$, the must and may abstraction of rule $i \in \{1, \ldots, k\}$ can be given as follows:

$$i.trans_{m+}^{f,R} := \{(r_1, r_2) \in (R \times R) \mid r_1 \in i.guard\uparrow_V^m,\ r_2 \in i.update(r_1\downarrow)\uparrow_V^m\}$$

$$i.trans_{M-}^{f,R} := \{(r_1, r_2) \in (R \times R) \mid r_1 \in i.guard\uparrow_V^M,\ r_2 \in i.update(r_1\downarrow)\uparrow_V^m\}$$

For all $j \in \{m+, M-\}$, $X \subseteq 2^R$, the approximate transition relation, one step predecessor operator and multi-step predecessor operator can be given respectively as:

$$Trans_j^{f,R} := \bigcup_{i=1\ldots k} i.trans_j^{f,R}$$

$$Pre_j^{f,R,1}(X) := \{r \in R \mid Trans_j^{f,R}(r) \cap X \neq \emptyset\}$$

$$Pre_j^{f,R}(X) := \{r \in R \mid r \cap (\mu Y.(X \cup Pre_j^{f,R,1}(Y))) \neq \emptyset\}$$

. For disjunctive transition relation, the approximate may predecessor operator will be precise; however, the approximate must predecessor will be under-approximation of the precise one.

Theorem 2 *For each $f \in F$, $R \subseteq 2^{2^{S_f}}$, and $X \subseteq 2^R$, we have*

$$Pre_{M-}^{f,R}(X)\downarrow \subseteq Pre^{f,*}(X\downarrow) \subseteq Pre_{m+}^{f,R}(X)\downarrow.$$

Rule Partition for Function One more optimization will be partitioning the rule set of each function with respect to the abstraction to create *less splitting.* Computation of each individual rule for must abstraction can create huge under-approximation; hence may need more splitting.

Example 6 *In presence of If-Then-Else or Switch constructs in the source code, we may encounter the following rules after the translation.*

$$r_1 : hd = true ==> indata' = 0; hd' = false$$

$$r_2 : hd = false ==> indata' = 0; hd' = hd$$

The abstract set R is defined with respect to different valuations of indata *variable. If we consider each rule separately and apply the must abstraction, we miss the fact that the final value of variable indata will be* 0 *and does not depend on the initial value of hd. The must predecessor of* $S \mid_{indata=0}$ *will be* $\emptyset$ *for both rules since the must abstraction of guards will be empty-set. However, if we combine two rules by taking union of sets, then the must predecessor of* $S \mid_{indata=0}$ *will be S for the combined rule and there will not be any further splitting.*

The heuristic of rule set partition is obtained from the abstraction itself. If a function f has k rules, then i-th and j-th rules can be grouped together for an abstraction R if the condition $i.guard\uparrow_V^m = j.guard\uparrow_V^m$ holds.

Incremental Building of Interface: Algorithm 6 can be used for incremental addition of function sets; as we may not need to create the interface for all the functions at first. The algorithm returns the refined interface for the included functions only. The created interface can be used if we want to add more functions from the library.

6.3 Translation from C to Guard-Update Rules

In this section, we discuss the translation scheme to convert C functions into the "sociable interface automata" [39] format. This format contains several guarded-update rules and is the input format of the symbolic tool TICC. Here the front-end and back-end are independent. Hence one only need a different front-end to parse functions from any other language (like Java/C++) to generate the TICC input format models. The following stages of the algorithm can reuse the out tool TICC to create interface graphs.

The algorithm feeds C functions into CIL[82] tool that parses C source code and returns the control flow graph. The control flow graph contains block structure as nodes and conditions as the transitions. We have modified the control flow graph for each function into a number of guarded-update rules. The guards represent conditions and updates represent the assignments. The specific variable s defines the location of current block. For a variable v, the primed variable v' denotes the v in the next step. When the translator encounters a critical error condition (e.g. call to $exit(1)$) in the control flow graph; the global variable err equals to 1 in the translated library.

- Control Flow Structures: The C source like "if (a =0) {b=0;} else {b=1;}" is converted into the following rules:

$$a = 0, s = 0 ==> b' = 0, s' = 1;$$

$$a! = 0, s = 0 ==> b' = 1, s' = 1$$

  The switch and loop (like while, for) structures can be handled similarly.

- Variables and Data Structures: Currently, the algorithm supports unsigned integers with a small number (e.g. 4) of bits. The translation flattens the fixed-size arrays and structures. In the Integer Stack example, in Figure 6.1(b) shows how 3 integer variables represent an

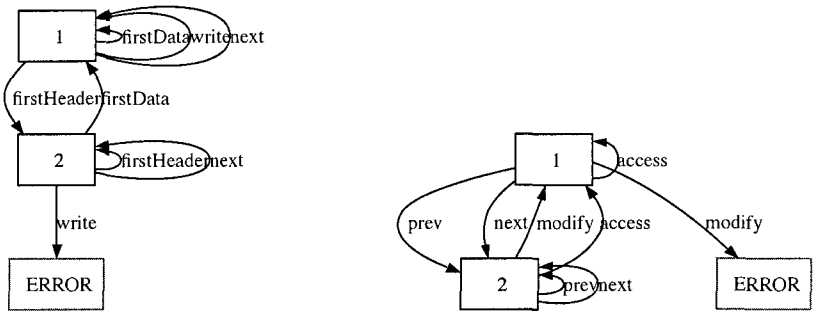
array of size 3. The structure elements are also flattened in the example. Currently, the translation does not directly handle pointers and recursive data types. However, we can manually translate the pointers into integers only if we know that the control flow of the function is independent of the value at its pointer location.

- Function Calls: Currently in order to compute the abstract transition for function f, we inline all the intermediate function call in the body of f. In the guarded-update rule semantics, the rules of the intermediate functions are explicitly added to the rules of f. The algorithm maintains an explicit stack to handle functions and stores the return address and the context variables in the stack. This trick can be applied to one function calling another function as well as the non-tail-recursive function calls. The tail-recursive function calls can be converted into loops and do not need the stack. In the Appendix, we demonstrate a complete translation of a recursive c function.

6.4 Results

In this section we will provide results of some case studies and compare with the related works.

Data Stream Case Study There is a data stream with a header of length 2^h and data of length 2^d where $h \leq d$. The program uses d bits to represent the pointer and 1 bit for the "error". The Boolean variable $isHeader$ is 1 when in the header and is 0 otherwise. There are four functions in the program. The function $FirstHeader$ and $FirstData$ takes the pointer to the first header and data location respectively. The function $Next$ moves the pointer within the header or data in a cyclic fashion. The function $Write$ results in an error when pointer points to header section. Figure 6.3(a) shows the interface for the data-stream example. State 1



(a) Data Stream (b) Bit-Array-Manipulator

Figure 6.3: Interfaces

represents that the pointer in the data part and state 2 represents that the pointer in the header part.

Bit Array Manipulator The Bit Array Manipulator has four functions : *prev*, *next*, *access* and *modify*. Two global variables *ptr* of length 2^k specify the current position of the pointer. The global Boolean variable *valid* denotes whether the pointer is valid. Another Boolean variable *err* specify the library error states. The functions *next* and *prev* respectively increments and decrements the current pointer and set the valid flag to true. The functions *access* resets the valid flag. The function $modify$ sets *err* to true when the valid is false, otherwise sets valid to false. Figure 6.3(b) shows the interface graph for the bit-array example. The state 1 represents that the valid bit is false and the state 2 represents that the valid bit is true.

Comparison Figure 6.4 shows a comparison of *explore* algorithm with the related work on these two examples. The first two columns show the name and different parameter values of the case-studies. The next column describes the running time (in milliseconds) of *explore* algorithm from the parsed guarded-update rules. The next column represents the number of non-error

Case Study	Params	Time (ms)	Regions	Direct	Learning	CEGAR
Data Stream	$h = 2, d = 12$	3	2	1028	2	257
	$h = 4, d = 12$	4	2	4112	2	257
	$h = 13, d = 13$	18	2	16384	2	2
Bit Array Manipulator	$k = 8$	2	2	68	2	2
	$k = 9$	4	2	130	2	2
	$k = 16$	8	2	16386	Timeout	2

Figure 6.4: Results

regions in the interface graph. The last three columns show non-error regions from other three related works; we obtain the data from [16]. The results for Direct algorithm show that *direct* algorithm runs fastest, but the size of interface graph is exponential in d. We obtain that the CEGAR algorithm provides minimal graph only when $h = d$ in the Data Stream example. The size of the graph in the CEGAR algorithm depends on the proper representation of variables with Boolean variables. The CEGAR approach refine by adding a new Boolean variable; which has a risk of splitting many abstract states unnecessarily. In contrast, *explore* algorithm keeps global abstraction separate from local abstraction inside the function and refines the global abstraction lazily with respect to the final reachable set (S_M^f). Learning algorithm provides the minimal graph, but slowest of all three approaches. *Explore* algorithm provides the same number of non-error regions as the learning algorithm. However, we can not compare time due to different platforms.

6.5 Application of Interfaces

In this section, we describe how a safe and permissive interface can be useful in the verification and testing of the software programs. The following section briefly describe the modifications needed for the interface to be compatible with these settings.

6.5.1 Software Verification with Interfaces

After the algorithm builds an interface graph for a set of functions, one can easily verify a given client program. The plan would be simulating the actions of the client program into the interface graph and check whether the simulation trace reaches the library error state (State ERROR). For example, a client with a single line $modify(b)$ on the BitArrayManipulator b can be simulated in the interface graph (Figure 6.3(b)). . We find a simulation trace from the initial state to the error state (ERROR) (State 1). There could be an infinite number of potential clients corresponding to those functions. We can compute the interface graph and model-check each of them.

6.5.2 Offline Test Case Generation

In the model-based testing (MBT) paradigm, the tester checks an implementation under test (IUT) with respect to a given model program (a specification of the IUT). The proposed algorithm can create an interface graph from the definitions of the functions given in the model program. We can create a C source regression test-suite from the interface generated from the libraries. However, we need to extend the function calls with the argument values to create a test-bench for the IUT. For example, Figure6.1(a) can be generated from the model program in Figure6.1(c). If we have a linked-list implementation of an integer stack of finite length, we can create an offline test-suite from the interface graph. The testing of the implementation with respect to the test-suite checks whether the interface goes to the error state if and only if the implementation goes to the error state. If we find a discrepancy between the behavior of the interface graph and the system implementation, we assume the possibility of bugs in the implementation source code.

6.6 Conclusions

In this chapter, we provide a new algorithm for interface synthesis with a local-global abstraction refinement framework. This framework is can dramatically reduce the state-space of the interface generation by hiding local variables inside each function. The abstract summarization of the functions provides scalability. The framework uses modular analysis to handle each function separately. In this generalized setting, any C-style set of functions can be handled.

The results illustrate that the algorithm provides a safe, permissive and sufficiently minimal (i.e. comparable to the learning algorithms) interface from the set of functions. We have provided the approximate, abstract predecessor operators to handle the state-space inside the function. The interface synthesis can be incremental : hence one can add new functions to the interface and it may lead to refinements corresponding to the function.

The user can immediately verify clients with respect to the interface graph and the graph can provide an offline test-suite for a new implementation. However, the translation engine is extremely basic. In the future, we want to work more on covering more aspects (e.g. pointers, recursive data types) of C source code such that we can have bigger case-studies. We want to see how we can use the shape analysis algorithms to translate complex data types. We also want to include CIL inside the tool TICC such that it can parse C functions and represent the rules directly in MDD format. We want to implement the back-end using a combination of MDD and SMT solvers such that the space-space problems can be handled better.

Algorithm 7 AbsRef(R, f, E)

Input: Abstraction R, function f, error set E

Output: updated R

1. $V_{abs} := \cup_{r \in R} support(r)$, R_f:=R
2. **loop**
3. $S_M := Pre_M^{f,R_f}(E)$; $S_M^f := S_M \cap I_L^f$
4. $S^1 := Pre^{f,1}(S_M\downarrow)$
5. $s_{new} := S^1 \setminus (S_M\downarrow)$
6. **if** $s_{new} := \emptyset$ **then** R_G:=R
7. **for each** $r \in R$ **do**
8. **if** $(r \cap S_M^f) \neq \emptyset$ & $(r \setminus S_M^f) \neq \emptyset$
9. R_G:=$R_G \cup \{r_1, r_2\} \setminus \{r\}$, where $r_1 := (r \cap S_M^f)$ and $r_2 := (r \setminus S_M^f)$
8. **return** R_G
7. **else**
8. split including a variable v from $\{v \in (V^f \setminus V_{abs}) \mid v \in support(s_{new})\}$
10. Abstraction R_f is refined for all valuations of v
11. **end if**

Algorithm 8 BuildInterface(R, F, Lib)

Input: Abstraction R, a set of functions F, a library $Lib = (F_G, V_G, E, I)$

Output: Interface Graph $IG = (N, T, T_e, In, Er)$

1. $Q, N, T, T_e, In, Er = \emptyset$
2. $append(Q, I); append(N, I \cup E); append(In, I); append(Er, E)$
3. **while** Q is non-empty **do**
4. curr := removeFirst(Q)
5. **for each** $f \in F$ do
6. next := $Post_m^{f,R}(curr)$
7. **if** (**not** member(N, next)) **then** append (Q, next); append (N,next) **endif**
8. **if** $(next \subseteq E)$ **then** $T_e := T_e \cup (curr, f, Er)$ **else** $T := T \cup (curr, f, next)$**endif**
9. **end for**
10. **end while**

Part IV

Probabilistic Abstraction

Chapter 7

Magnifying Lens Abstraction

7.1 Introduction

Markov decision processes (MDPs) provide a model for systems with both probabilistic and nondeterministic behavior, and they are widely used in probabilistic verification, planning, optimal control, and performance analysis [54, 15, 94, 33, 48]. MDPs that model realistic systems tend to have very large state spaces, and the main challenge in their analysis consists in devising algorithms that work efficiently on such large state spaces. In the non-probabilistic setting, abstraction techniques have been successful in coping with large state-spaces: abstraction enables to answer questions about a system by considering a smaller, more concise abstract model. This has spurred research into the use of abstraction techniques for probabilistic systems [32, 67, 81, 70]. We present a novel abstraction technique, called *magnifying-lens abstraction* (MLA), for the analysis of reachability and safety properties of MDPs with very large state spaces. We show that the technique can lead to substantial space savings in the analysis of MDPs.

An MDP is defined over a state space S. At every state $s \in S$, one or more *actions* are available; with each action is associated a probability distribution over the successor states.

We focus on *safety* and *reachability* properties of MDPs. A safety property specifies that the MDP's behavior should not leave a *safe* subset of states $T \subseteq S$; a reachability property specifies that the behavior should reach a set $T \subseteq S$ of *target* states. A controller can choose the actions available at each state so as to maximize, or minimize, the probability of satisfying reachability and safety properties. MLA computes converging upper and lower bounds for the maximal reachability or safety probability; the minimal probabilities can be obtained by duality. In its ability to provide both upper and lower bounds for the quantities of interest, MLA is similar to [70].

In the analysis of large MDPs, the main challenge lies in the representation of the value $v(s)$ of the reachability or safety probability at all $s \in S$. In contrast, actions and transition probabilities from each state s can usually be either computed on the fly, or represented in a compact fashion, via Kronecker representations or probabilistic guarded commands [85, 48, 66]. The goal of MLA is to reduce the space required for storing v and, secondarily, the running time of the analysis. To this end, MLA partitions the state space S of the MDP into *regions;* for each region r, it stores upper and lower bounds $v^+(r)$, $v^-(r)$ for the maximal reachability or safety probability. The values $v^+(r)$, $v^-(r)$ constitute bounds for all states $s \in r$. In order to update these estimates, MLA iterates over the regions, "magnifying" one of them at a time. When the region r is magnified, MLA computes $v^+(s)$, $v^-(s)$ at all concrete states $s \in r$ via value iteration, and then summarizes these results by setting $v^+(r) = \max_{s \in r} v^+(s)$ and $v^-(r) = \min_{s \in r} v^-(s)$. Figuratively, MLA slides a magnifying lens across the abstraction, enabling the algorithm to see the concrete states of one region at a time when updating the region values. Given a desired accuracy ε for the answer, MLA periodically splits regions r with $v^+(r) - v^-(r) > \varepsilon$ into smaller regions. In this way, the abstraction is refined in an adaptive fashion: smaller regions are used where finer detail is needed, guaranteeing the convergence of the bounds, and larger regions are used elsewhere, saving space. When splitting regions, MLA takes care to re-use information

gained in the analysis of the coarser abstraction in the evaluation of the finer one. MLA can be adapted to the problem of computing a control strategy by recording the optimal actions for the concrete states of interest, when they are magnified.

Related work on MDP abstraction. Compared with other approaches to MDP abstraction, MLA has two distinctive features:

1. it clusters states based on value, rather than based on the similarity in their transition function;

2. it updates the valuation of abstract states by considering the concrete states associated with the abstract states, rather than by considering an abstract model only.

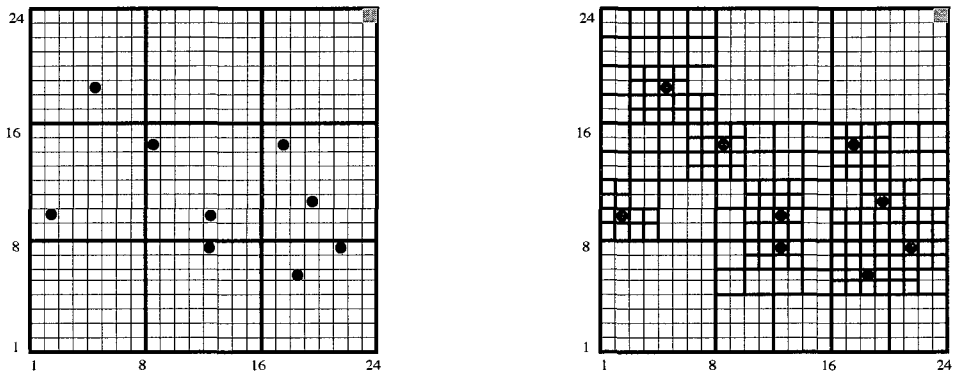
The second of the above points underlines how MLA is a semi-abstract, rather than fully abstract, approach to verification: the abstract computation still involves consideration of the concrete states, even though this is done in a way that provides space savings.

For the most part, approaches to MDP abstraction in the literature have followed another route, which we call very broadly the *full abstraction* approach: an abstract model is constructed, and then analyzed on the basis of an abstract transition structure, without further reference to the concrete model. These fully abstract approaches generally rely on clustering states that are similar not only in value, but also in transition structure: in this way, every region of concrete states can be summarized via an abstract state with an associated abstract transition structure. The abstract transition structure may, or may not, be similar to the concrete one. For instance, [70] bases the abstract transition structure on games, rather than MDPs: in this fashion, player 1 can represent the choice of action of the MDP, and player 2 can represent the uncertainty about the concrete state corresponding to the abstract state. This approach enables the computation of lower and upper bounds for properties of interest, similarly to MLA.

In a somewhat related spirit, but using entirely different technical means, [58] proposes to abstract Markov chains into *abstract Markov chains* whose transitions are labeled with intervals of probability, representing the uncertainty about the concrete state. Clustering states based on the similarity in their transition probabilities has also been used in [52], which proposes to find the coarsest refinement of an MDP where for each action, states in the same region have the same probability of going to other regions. An approach for the verification of probabilistic reachability properties via abstraction has been proposed in [32]. The abstraction is built through successive refinements starting from a coarse partition based on the property. Several other approaches also, in fact, rely on constructing MDP abstractions based on simulation or abstract interpretation [67, 81, 80]; all of these approaches rely on clustering states with similar transition structure, and representing these clusters of states, and their transition structures, via compact abstract representations.

The full-abstraction approach outlined above, and the partial value-based approach followed by MLA, each have advantages. The full-abstraction result can handle unbounded, and (depending on the specific approach) even infinite state spaces. In contrast, the space savings afforded by MLA are limited to a square-root factor (a system of size n can be studied in $O(\sqrt{n})$ space), due to the need to consider the concrete states corresponding to each abstract one. Furthermore, the full-abstraction approaches typically need to construct the abstract model only once; in contrast, MLA needs to refer to concrete states (albeit not all of them at once) during the computation.

On the other hand, the ability of MLA to cluster states based on value only, disregarding differences in their transition relation, can lead to compact abstractions for systems where full abstraction provides no benefit. We will give below an example supporting this. Furthermore, in MLA the abstraction is refined dynamically, depending on the required accuracy of the analysis; there is no need to "guess" the right state partition in advance. In our experience, MLA is



(a) Initial Abstraction

(b) Final Abstraction

Figure 7.1: Initial, and final refined abstraction, for the problem of motion planning in a 24×24 minefield. The circles denote the mines.

particularly well-suited to problems where there is a notion of *locality* in the state space, so that it makes sense to cluster states based on variable values — even though their transition relations may not be similar. Many planning and control problems are of this type. MLA instead is not as well-suited to problems where clustering states based on variable values is less effective. Approaches based on predicate abstraction could lend the MLA approach more generality.

An example of Magnifying-Lens Abstraction. To illustrate MLA, and its potential benefits, we give a simple example. We consider the problem of navigating an $n \times n$ minefield. The minefield contains m mines, each with coordinates (x_i, y_i), for $1 \leq i \leq m$, where $1 \leq x_i < n$, $1 \leq y_i < n$. We consider the problem of computing the maximal probability with which a robot can reach the target corner (n, n), from all $n \times n$ states. At interior states of the field, the robot can choose among four actions: *Up, Down, Left, Right;* at the border of the field, actions that

lead outside of the field are missing. From a state $s = (x, y) \in \{1, \ldots, n\}^2$ with coordinates (x, y), each action causes the robot to move to square (x', y') with probability $q(x', y')$, and to "blow up" (move to an additional sink state) with probability $1 - q(x', y')$. For action *Right,* we have $x' = x + 1$, $y' = y$; similarly for the other actions. The probability $q(x', y')$ depends on the proximity to mines, and is given by

$$q(x', y') = \prod_i^m \exp\left(-0.7 \cdot \left((x' - x_i)^2 + (y' - y_i)^2\right)\right).$$

The problem, for $n = 24$, is illustrated in Figure 7.1.

Intuitively, it is desirable to group the 8×8 states in the top-middle area into a single region r_0: since no mines are nearby, the robot can freely roam in r_0, so that the maximal probability of reaching the target corner is essentially constant across r_0. Indeed, to a human trying to determine a best path to the target corner, the states in r_0 are essentially equivalent. When the 8×8 concrete states are grouped in r_0, MLA leads to accurate results, since it can analyze the dynamics inside r_0 when r_0 is magnified. We also note how, in this example, the ability of MLA to refine the abstraction adaptively is crucial. As shown in Figure 7.1(b), MLA is able to use small regions close to mines, and large regions elsewhere. If we insisted on a uniform region size, then we would have to adopt the smallest size throughout, and no space savings would be possible.

On the other hand, the full-abstraction approaches described earlier, such as [32, 81, 70], based on probabilistic simulation [95], are not well suited to this example. Such techniques would associate with an abstract state, such as r_0, a summary of the transition structure from states $s \in r_0$, and use that summary to analyze the abstraction. The problem is that the states in r_0, while similar in value, are not similar in transition structure: the states on the border of r_0 can transition outside of r_0, while those in the interior cannot. In the abstraction, the probability of going from r_0 to the region at the right hand side will be modeled as being in an interval

$[0, q]$, for some q close to 1 (all mines are far away). Consequently, previous techniques would have yielded a lower bound of 0, and an upper bound close to 1, for the maximum probability of reaching the target corner. Similarly, the technique of [52] would lead to recursively splitting the MDP, until the regions consisted of only one concrete state each.

Other related work. MLA is reminiscent to methods that represent value functions via ADDs or MTBDDs [30, 11] with an approximation factor used to merge leaves. The similarity, however, is superficial: MLA leads to far more precise results in the analysis; we discuss this in the conclusions, where the appropriate notation will be available.

MLA is also loosely reminiscent of *adaptive mesh refinement* (AMR) methods used in the solution of partial differential equations [14]. There are, however, two important differences between MLA and AMR. In AMR, separate lower and upper bounds are not kept. AMR methods perform computation at the finest mesh sizes only where needed. In MLA, due to the discrete nature of MDPs, we have no way of computing over a "coarse mesh" only: to update valuations over a region, we need to "magnify" the region to its individual states. Thus, MLA is forced to consider the individual states over the whole system, and it summarizes and returns the results in terms of lower and upper bounds, which are well-suited to answering verification questions.

7.2 Magnifying-Lens Abstraction

Magnifying-lens abstractions (MLA) is a technique for the analysis of reachability and safety properties of MDPs. Let v^* be the valuation on S that is to be computed: v^* is one of $V^{\min}_{\Box T}$, $V^{\max}_{\Box T}$, $V^{\min}_{\Diamond T}$, $V^{\max}_{\Diamond T}$. Given a desired accuracy $\varepsilon_{abs} > 0$, MLA computes upper and lower bounds for v^*, spaced less than ε_{abs}. MLA starts from an initial partition R of S, and computes the lower and upper bounds as valuations u^- and u^+ over R. The partition is refined, until the

difference between u^- and u^+, at all regions, is below a specified threshold. To compute u^- and u^+, MLA iteratively considers each r in turn, and performs a *magnified iteration:* it improves the estimates for $u^-(r)$ and $u^+(r)$ using value iteration on the concrete states $s \in r$.

The MLA algorithm is presented as Algorithm 9. The algorithm has parameters T, f, g, which have the same meaning as in Algorithm ValIter. The algorithm also has parameters $\varepsilon_{float} > 0$ and $\varepsilon_{abs} > 0$. Parameter ε_{abs} indicates the maximum difference between the lower and upper bounds returned by MLA. Parameter ε_{float}, as in ValIter, specifies the degree of precision to which the local, magnified value iteration should converge. MLA should be called with ε_{abs} greater than ε_{float} by at least one order of magnitude: otherwise, errors in the magnified iteration can cause errors in the estimation of the bounds. Statement 2 initializes the valuations u^- and u^+ according to the property to be computed: reachability properties are computed as least fix-points, while safety properties are computed as greatest fix-points [49]. A useful time optimization, not shown in Algorithm 9, consists in executing the loop at lines 6–9 only for regions r where at least one of the neighbor regions has changed value by more than ε_{float}.

Magnified iteration. The algorithm performing the magnified iteration is given as Algorithm 10. The algorithm is very similar to Algorithm 1, except for three points.

First, the valuation v (which here is local to r) is initialized not to $[T]$, but rather, to $u^-(r)$ if $f = \max$, and to $u^+(r)$ if $f = \min$. Indeed, if $f = \max$, value iteration converges from below, and $u^-(r)$ is a better starting point than $[T]$, since $[T](s) \leq u^-(r) \leq v^*(s)$ at all $s \in r$. The case for $f = \min$ is symmetrical.

Second, for $s \in S \backslash r$, the algorithm uses, in place of the value $v(s)$ which is not available, the value $u^-(r')$ or $u^+(r')$, as appropriate, where r' is such that $s \in r'$. In other words, the algorithm replaces values at concrete states outside r with the "abstract" values of the regions to which the states belong. To this end, we need to be able to efficiently find the "abstract"

Algorithm 9 MLA$(T, f, g, \varepsilon_{float}, \varepsilon_{abs})$ Magnifying-Lens Abstraction

1. $R :=$ some initial partition.
2. **if** $f = \max$ **then** $u^- := \mathbf{0}$; $u^+ := \mathbf{0}$ **else** $u^- := \mathbf{1}$; $u^+ := \mathbf{1}$
3. **loop**
4. **repeat**
5. $\hat{u}^+ := u^+$; $\hat{u}^- := u^-$;
6. **for** $r \in R$ **do**
7. $u^+(r) :=$ MagnifiedIteration$(r, R, T, \hat{u}^+, \hat{u}^-, \hat{u}^+, \max, f, g, \varepsilon_{float})$
8. $u^-(r) :=$ MagnifiedIteration$(r, R, T, \hat{u}^-, \hat{u}^-, \hat{u}^+, \min, f, g, \varepsilon_{float})$
9. **end for**
10. **until** $||u^+ - \hat{u}^+|| + ||u^- - \hat{u}^-|| \leq \varepsilon_{float}$
11. **if** $||u^+ - u^-|| \geq \varepsilon_{abs}$
12. **then** $R, u^-, u^+ :=$ SplitRegions$(R, u^-, u^+, \varepsilon_{abs})$
13. **else return** R, u^-, u^+
14. **end if**
15. **end loop**

counterpart $[s]_R$ of a state $s \in S$. We use the following scheme, similar to schemes used in AMR [14]. Most commonly, the state-space S of the MDP consists in value assignments to a set of variables $X = \{x_1, x_2, \ldots, x_l\}$. We represent a partition R of S, together with the valuations u^+, u^-, via a binary decision tree. The nodes of the tree are labeled by $\langle y, i \rangle$, where $y \in X$ is the variable according to which we split, and i is the position of the bit (0 =LSB) of the variable according to whose value we split. The leaves of the tree correspond to regions, and they are labeled with u^-, u^+ values. Given s, finding $[s]_R$ in such a tree requires time logarithmic in $|S|$.

Third, once the concrete valuation v is computed at all $s \in r$, Algorithm 10 returns

Algorithm 10 MagnifiedIteration$(r, R, T, u, u^-, u^+, h, f, g, \varepsilon_{float})$

v: a valuation on r

1. **if** $f = \max$
2. **then for** $s \in r$ **do** $v(s) = u^-(r)$
3. **else for** $s \in r$ **do** $v(s) = u^+(r)$
4. **repeat**
5. $\hat{v} := v$
6. **for all** $s \in r$ **do**

$$v(s) = f\left([T](s),\ g\left\{\sum_{s' \in r} p(s,a,s') \cdot \hat{v}(s') + \sum_{s' \in S \setminus r} p(s,a,s') \cdot u([s]_R) \;\middle|\; a \in \Gamma(s)\right\}\right)$$

7. **until** $||v - \hat{v}|| \leq \varepsilon_{float}$
8. **return** $h\{v(s) \mid s \in r\}$

the minimum (if $h = \min$) or the maximum (if $h = \max$) of $v(s)$ at all $s \in r$, thus providing a new estimates for $u^-(r)$, $u^+(r)$, respectively.

Adaptive abstraction refinement. We denote the *imprecision* of a region r by $\Delta(r) = u^+(r) - u^-(r)$. MLA adaptively refines a partition R by splitting all regions r having $\Delta(r) > \varepsilon_{abs}$. This is perhaps the simplest possible refinement scheme. We experimented with alternative refinement schemes, but none of them gave consistently better results. In particular, we considered splitting the regions with high Δ-value, all whose successors, according to the optimal moves, have low Δ-value: the idea is that such regions are the ones where precision degrades. While this reduces somewhat the number of region splits, the total number of refinements is increased, and the resulting algorithm is not clearly superior, at least in the examples we considered. We also experimented with splitting all regions $r \in R$ with $\Delta(r) > \delta$, for a threshold δ that is initially

set to $\frac{1}{2}$, and that is then gradually decreased to ε_{abs}. This approach, inspired by simulated annealing, also failed to provide consistent improvements.

In the minefield example, each region is *squarish* (horizontal and vertical sizes differ by at most 1); we split each such squarish region into 4 smaller squarish regions. In more general cases, the following heuristic for splitting regions is widely applicable, and has worked well for us. The user specifies an ordering $x_0, x_1, \ldots, x_l$ for the state variables X defining S: this specifies a priority order for splitting regions. As previously mentioned, we represent a partition R via a decision tree, whose leaves correspond to the regions. In the refinement phase, we split a leaf according to the value of a new variable (not present in that leaf), following the variable ordering given by the user. Precisely, to split a region r, we look at the label $\langle x_j, i\rangle$ of its parent node. If $i > 0$, we split according to bit $i-1$ of x_j; otherwise, we split according to the MSB of x_{j+1}. A refinement of this technique allows the specification of groups of variables, whose ranges are split in interleaved fashion. Once a region r has been split into regions r_1, r_2, we set $u^-(r_j) = u^-(r)$ and $u^+(r_j) = u^+(r)$ for all $j = 1, 2$. A call to SplitRegions$(R, u^+, u^-, \varepsilon_{abs})$ returns a triple $\tilde{R}, \tilde{u}^-, \tilde{u}^+$, consisting of the new partition with its upper and lower bounds for the valuation.

Correctness. The following theorem summarizes MLA correctness.

Theorem 3 *For all MDPs $M = \langle S, A, \Gamma, p\rangle$, all $T \subseteq S$, and all $\varepsilon_{abs} > 0$, the following assertions hold.*

1. Termination. *For all $\varepsilon_{float} > 0$, and for all $f, g \in \{\min, \max\}$, the call $MLA(T, f, g, \varepsilon_{float}, \varepsilon_{abs})$ terminates.*

2. (Partial) correctness. *Consider any $g \in \{\max, \min\}$, any $\varepsilon_{abs} > 0$, and any $\triangle \in \{\Box, \Diamond\}$, and let $f = \min$ if $\triangle = \Box$, and $f = \max$ if $\triangle = \Diamond$. The following holds. For all $\delta > 0$,*

there is $\varepsilon_{float} > 0$ *such that:*

$$\forall r \in R: \qquad u^+(r) - u^-(r) \;\leq\; \varepsilon_{abs}$$

$$\forall s \in S: \qquad u^-([s]_R) - \delta \;\leq\; V^g_{\triangle T}(s) \;\leq\; u^+([s]_R) + \delta$$

where $(R, u^-, u^+) = MLA(T, f, g, \varepsilon_{float}, \varepsilon_{abs})$.

We note that the theorem establishes the correctness of lower and upper bounds only within a constant $\delta > 0$, which depends on ε_{float}. This limitation is inherited from the value-iteration scheme used over the magnified regions. If linear programming [54, 15] were used instead, then MLA would provide true lower and upper bounds. However, in practice value iteration is preferred over linear programming, due to its simplicity and great speed advantage, and the concerns about δ are solved — in practice, albeit not in theory — by choosing a small $\varepsilon_{float} > 0$.

7.3 Experimental Results

In order to evaluate the time and space performance of MLA, we have implemented a prototype, and we have used it for three case studies: the minefield navigation problem, the Bounded Retransmission Protocol [32], and the ZeroConf protocol for the autonomous configuration of IP addresses [27, 70].

When comparing MLA to ValIter, we compute the space needs of the algorithms as follows. For ValIter, we take the space requirement to be equal to $|S|$, the domain of v. For MLA, we take the space requirement to be the maximum value of $2 \cdot |R| + \max_{r \in R} |r|$ that occurs every time MLA is at line 4 of Algorithm9: this gives the maximum space required to store the valuations u^+, u^-, as well as the values v for the largest magnified region. Since $\max_{r \in R} |r| \geq (|S|/|R|)$, the space complexity of the algorithm is (lower) bounded by a square-root function $\sqrt{8 \cdot |S|}$.

Algorithm	Space	Time
ValIter	16,384	21.97
MLA	7,926	123.54

MLA Iteration Details			
#Abs	$\|R\|$	D	Time
1	144	0.99	9
2	576	0.83	38
3	2,312	0.66	47
4	3,256	0.64	11
5	3,566	0.02	14
6	3,899	0.01	2

(a) $n = 128$, $m = 128$

Algorithm	Space	Time
ValIter	65,536	130.18
MLA	7,944	185.13

MLA Iteration Details			
#Abs	$\|R\|$	D	Time
1	256	0.98	49
2	985	0.65	76
3	1,513	0.77	12
4	2,341	0.60	17
5	3,844	0.01	29

(b) $n = 256$, $m = 128$

Algorithm	Space	Time
ValIter	262,144	1,065.36
MLA	30,180	3,199.31

MLA Iteration Details			
#Abs	$\|R\|$	D	Time
1	576	0.99	299
2	2,295	0.77	1648
3	4,347	0.77	206
4	7,171	0.66	228
5	11,678	0.52	362
6	14,862	0.01	453

(c) $n = 512$, $m = 512$

Figure 7.2: Comparison between MLA and ValIter for $n \times n$ mine-fields with m mines, for $\varepsilon_{abs} = 10^{-2}$ and $\varepsilon_{float} = 10^{-4}$. Mine densities ($m/n^2$) are (a) 1/64, (b) 1/512, and (c) 1/512. All times are in seconds. #Abs is the number of abstraction steps (number of loops 3–15 of MLA), and $D = \max_{r \in R}(u^+(r) - u^-(r))$.

7.3.1 Minefield Navigation

We experimented with different-size mine-fields in the mine-field example. In all cases, the mines were distributed in a pseudo-random fashion across the field. The performance of algorithms ValIter and MLA, for $\varepsilon_{abs} = 0.01$, are compared in Figure 7.2. As we can see, the space savings are 2.06 for a mine density of 1/64, and an average of 8.47 for a mine density of 1/512. This comes at a cost in running time, which is of 5.67 for a mine density of 1/64, and 1.42 to 3.00 for a mine density of 1/512. Especially for lower mine densities, MLA provides space savings that are larger than the incurred time penalty. The space savings are even more pronounced when we decrease the desired precision of the result to $\varepsilon_{abs} = 0.1$, as indicated in

Algorithm	Space	Time
ValIter	16,384	20.51
MLA	3,672	54.51

(a) $n = 128$, $m = 128$

Algorithm	Space	Time
ValIter	65,536	130.08
MLA	4,548	126.40

(b) $n = 256$, $m = 128$

Algorithm	Space	Time
ValIter	262,144	1,065.65
MLA	15,476	1,853.01

(c) $n = 512$, $m = 512$

Figure 7.3: Comparison between MLA and ValIter for $n \times n$ mine-fields with m mines, for $\varepsilon_{abs} = 10^{-1}$ and $\varepsilon_{float} = 10^{-2}$. Mine densities (m/n^2) are (a) 1/64, (b) 1/512, and (c) 1/512. All times are in seconds.

Figure 7.3.

7.3.2 The ZeroConf Protocol

The ZeroConf protocol [27] is used for the dynamic self-configuration of a host joining a network; it has been used as a test-bed for the abstraction method considered in [70]. We consider a network with 4 existing hosts, and 32 total IP addresses; protocol messages have a certain probability of being lost during transmission. We consider the problem of determining the worst-case probability of a host eventually acquiring an IP address: this is a probabilistic reachability problem.

The abstraction approach of [70] reduces the problem from 26,121 concrete reachable states to 737 abstract states. MLA reduces the problem to 131 regions, requiring a total space of 1267 (including also the space to perform the magnification step) for $\varepsilon_{abs} = 10^{-3}$ and $\varepsilon_{float} = 10^{-6}$. We cannot compare the running times, due to the absence of timing data in [70].

7.3.3 Bounded Retransmission Protocol

We also considered the Bounded Retransmission Protocol described in [32]. We compared the performance of algorithms ValIter and MLA on "Property 1" from [32], stating that

N	MAX	ValIter time	#Reachable states	MLA space	MLA time
16	3	0.08	1,966	918	27.38
32	5	0.21	5,466	2,604	140.79
64	5	0.40	10,650	5,380	266.53

Figure 7.4: Comparison between MLA and ValIter for BRP. N denotes number of chunks and MAX denotes the maximum number of retransmissions. All times are in seconds.

the sender eventually does not report a successful transmission. The results are compared in Figure 7.4, for $\varepsilon_{abs} = 10^{-2}$ and $\varepsilon_{float} = 10^{-4}$. MLA achieves a space saving of a factor of 2, but at the price of a great increase in running time.

7.3.4 Discussion

From these examples, it is apparent that MLA does well on problems where there is some notion of "distance" between states, so that "nearby" states have similar values for the reachability or safety property of interest. These problems are common in planning and control. As we discussed in the introduction, many of these problems do not lend themselves to abstraction methods based on the similarity of transition relations, such as [70, 32], and other methods based on simulation. We believe the MLA algorithm is valuable for the study of this type of problems. We note that each mine affects a region of size 5×5 by more than the desired precision $\varepsilon_{abs} = 10^{-2}$. Therefore, while the mine density is only $1/512$, the ratio of "disturbed" vs. "undisturbed" state space is $25/512$, or $1/20$. This is a typical value in planning problems with sparse obstacles.

On the other hand, for problems where simulation-based methods can be used, these methods tend to be more effective than MLA, as they can construct, once and for all, a small abstract model on which all properties of interest can be analyzed.

7.4 Conclusions

A natural question about MLA is the following: why does MLA consider the concrete states at each iteration, as part of the "magnification" steps, rather than constructing an abstract model once and for all, and then analyze it, as other approaches to MDP abstraction do [32, 67, 81, 70]? The answer has two parts. First, we cannot build an abstract model once and for all: our abstraction refinement approach would require the computation of several abstractions. Second, we have found that the cost of building abstractions that are sufficiently precise, without resorting to a "magnification" step, is substantial, negating any benefits that might derive from the ability to perform computation on a reduced system.

To understand the performance issues in constructing precise abstractions, consider the problem of computing the maximal reachability probability. To summarize the maximal probability of a transition from a region r to r_1, we need to compute $P_r^+(r_1) = \min_{s \in r} \max_{\pi \in \Pi} \Pr_s^\pi(r \mathcal{U} r_1)$, where $\mathcal{U}$ is the "until" operator of linear temporal logic [78]; this quantity is related to building abstractions via *weak simulation* [95, 12, 86]. These probability summaries are not additive: for $r_1 \neq r_2$, we have that $P_r^+(r_1) + P_r^+(r_2) \leq P^+(r_1 \cup r_2)$, and equality does not hold in general. Indeed, these probability summaries constitute *capacities,* and they can be used to analyze maximal reachability properties via the Choquet integral [93, 59, 60]. To construct a fully precise abstraction, one must compute $P_r^+(R')$ for all $R' \subseteq R$, clearly a daunting task. In practice, in the minefield example, it suffices to consider those $R' \subseteq R$ that consist of neighbors of r. To further lower the number of capacities to be computed, we experimented with restricting R' to unions of no more than k regions, but for all choices of k, the algorithm either yielded grossly imprecise results, or proved to be markedly less efficient than MLA.

The space savings provided by MLA are bounded by a square-root function of the state space. We could improve this bound by applying MLA hierarchically, so that each magnified

region is studied, in turn, with a nested application of MLA.

Symbolic representations such as ADDs and MTBDDs [30, 11] have been used for representing the value function compactly [48, 66]. The decision-tree structure used by MLA to represent regions and abstract valuations is closely related to MTBDDs. The space savings are limited by the fact that the value function is usually slightly different at different states. MLA is loosely reminiscent of approaches that cluster MTBDDS leaves with values within a specified $\varepsilon > 0$. However, the similarity is superficial: such leaf-clustering corresponds in MLA to taking $\varepsilon_{abs} = \varepsilon_{float} = \varepsilon$, and yields considerably poorer results than clustering according to ε_{abs}, and computing according to ε_{float}, as MLA does. In particular, MTBDDS leaf-clustering approaches do not yield lower and upper bounds for the property of interest. In the next chapter we intend to explore symbolic implementations of MLA, where separate MTBDDs will be used to represent lower and upper bounds.

Chapter 8

Symbolic Magnifying Lens Abstraction

8.1 Introduction

Markov decision processes (MDPs) provide a model for systems with both probabilistic and nondeterministic behavior, and are widely used in probabilistic verification, planning, inventory optimal control, and performance analysis [54, 15, 94, 33, 92]. At every state of an MDP, one or more *actions* are available; each action is associated with a probability distribution over the successor states. We focus on *safety* and *reachability* properties of MDPs. A safety property specifies that the MDP's behavior should not leave a *safe* subset of states; a reachability property specifies that the behavior should reach a set of *target* states. A controller can choose the actions available at each state so as to maximize, or minimize, the probability of satisfying reachability and safety properties. MDPs that model realistic systems tend to have very large state spaces, and therefore the main challenge in analyzing such MDPs consists in devising algorithms that work efficiently on large state spaces.

In the non-probabilistic setting, abstraction techniques have been successful in coping with large state-spaces: by ignoring details not relevant to the property under study, abstraction

makes it possible to answer questions about a system through the analysis of a smaller, more concise abstract model. This has spurred research into the use of abstraction techniques for probabilistic systems [32, 67, 81, 70]. The majority of these techniques follow a *full abstraction* approach: an abstract model is constructed and, during its analysis, all details about the concrete system are forgotten.

In [50], de Alfaro and Roy proposed an alternative approach, called *magnifying-lens abstraction* (MLA) [50]. This is based on partitioning the state space of an MDP into regions and then analyzing ("magnifying") the states of each region separately. The lower and upper bounds for the magnified region are updated by computing the minimum and maximum values over the states of the region. Figuratively, MLA slides a magnifying lens across the abstraction, enabling the algorithm to see the concrete states of one region at a time when updating the region values.

Regions are refined adaptively until the difference between the lower and upper bounds for all regions is within some specified accuracy. In this way, the abstraction is refined in an adaptive fashion: smaller regions are used when finer detail is required, guaranteeing the convergence of the bounds, and larger regions are used elsewhere, saving space. When splitting regions, MLA takes care to re-use information gained in the analysis of the coarser abstraction in the evaluation of the finer one. In its ability to provide both upper and lower bounds for the quantities of interest, MLA is similar to [70].

Although experimental results have demonstrated that using MLA leads to space savings, the explicit representation of the probabilistic transition system employed in [50] placed a limit on the size of MDPs that could be analyzed. A successful approach to overcome the limitations of explicit representations has been to employ symbolic data structures. In particular, BDDs (binary decision diagrams) [21] and MTBDDs (multi-terminal binary decision diagrams) [29, 11] have been shown to enable the compact representation and analysis of very large MDPs

[48, 83, 65].

In this work we combine MLA with symbolic representations to improve scalability. More precisely, we adapt the MLA algorithm of [50] to the symbolic domain, yielding an approach that we call Symbolic Magnifying-Lens Abstraction (SMLA). We show that the "magnified" computation performed on the regions, and the "sliding" of the magnification from one region to the next, can be performed symbolically in a natural and efficient fashion. We have implemented SMLA in the probabilistic model checking tool PRISM [66, 88] and, through a number of case studies, demonstrate that SMLA leads to useful space savings.

MLA, and its symbolic variant SMLA, differ from other approaches to MDP abstraction [70] in that they can be profitably applied to systems where there are many states with similar value, but not necessarily similar transition structure. For instance, consider a system with an integer state variable x, with range $[0, \ldots, N]$, and assume that from every state where x has value $0 < n < N$, there are transitions to states where x has values $n-1$, n, and $n+1$. Classical abstraction schemes associate with each region (set of states) a single abstract state, whose transition relation over-approximates all the transition relations of the concrete states it represents. In such a transition-based abstraction, it is not useful to group the concrete values $[0, \ldots, N]$ for x into regions consisting of intervals $I_1, \ldots, I_k$. In fact, since the states at the endpoints of each interval can leave the interval, but the states in the interior cannot, the abstract transition relation associated with each interval would have to be a gross over-approximation of the concrete transition relations, leading to considerable loss of precision.

In MLA and SMLA, as long as the value of the property of interest is similar in states in the same interval, abstraction is possible and useful. Indeed, experimentally we noticed that SMLA performs well in many problems with integer-valued state variables, where the properties vary gradually with the value of the state variables. Problems in planning, inventory control, and similar often belong to this category. On the other hand, when it is possible to

use symmetry or structural knowledge of an example, and aggregate states of similar transition relation, approaches such as [32, 70, 71] yield superior results.

8.1.1 Symbolic model checking of MDPs

Due to the sizes of the MDPs that typically arise in probabilistic verification case studies, considerable effort has been invested into building efficient model checking implementations. In particular, *symbolic* techniques, which use extensions of Binary Decision Diagrams (BDDs), have proved successful in this area. Here we focus on the use of Multi-Terminal Binary Decision Diagrams (MTBDDs). This data structure lies at the heart of the probabilistic model checker PRISM and has been used to model check quantitative properties of probabilistic models with as many as 10^{10} states (see for example [72, 56]). In this section, we give a brief overview of these techniques. For more detailed coverage of the MTBDD-based implementation of MDP model checking in PRISM, see [83].

MTBDDs. Multi-terminal BDDs (MTBDDs) are rooted, directed acyclic graphs associated with a set of ordered, Boolean variables $x_1 < \ldots < x_n$. An MTBDD M represents a function $f_{\mathsf{M}}(x_1, \ldots, x_n) : \mathbb{B}^n \rightarrow \mathbb{R}$ over these variables. The graph contains two types of nodes: *non-terminal* and *terminal*. A non-terminal node m is labeled by a variable $var(m) \in \{x_1, \ldots, x_n\}$ and has two children, $then(m)$ and $else(m)$. A terminal node m is labeled by a real number $val(m)$. The Boolean variable ordering $<$ is imposed onto the graph by requiring that a child m' of a non-terminal node m is either terminal or non-terminal and satisfies $var(m) < var(m')$. The value of $f_{\mathsf{M}}(x_1, \ldots, x_n)$, the function which the MTBDD represents, is determined by traversing M from the root node, and at each subsequent node m taking the edge to $then(m)$ or $else(m)$ if $var(m)$ is 1 or 0 respectively. A BDD is simply an MTBDD with the restriction that labels on terminal nodes can only 0/1.

Representation of MDPs using MTBDDs. MTBDDs have been used, from their inception [11, 29], to encode real-valued vectors and matrices. An MTBDD v over variables $(x_1, \ldots, x_n)$ represents a function $f_{\mathsf{v}} : \mathbb{B}^n \to \mathbb{R}$. A real vector $\mathbf{v}$ of length 2^n is simply a mapping from $\{1, \ldots, 2^n\}$ to the reals $\mathbb{R}$. Using the standard binary encoding of integers, the variables $\{x_1, \ldots, x_n\}$ can represent $\{1, \ldots, 2^n\}$. Hence, an MTBDD v can represent the vector $\mathbf{v}$.

In a similar way, we can consider a square matrix M of size 2^n by 2^n to be a mapping from $\{1, \ldots, 2^n\} \times \{1, \ldots, 2^n\}$ to $\mathbb{R}$. This can be represented by an MTBDD over $2n$ variables, n for rows (current-state variables) and n for columns (next-state variables). According to the commonly-used heuristic for minimizing MTBDD size, the variables for rows and columns are ordered alternately.

MTBDDs can thus easily represent the probabilistic transition matrix of a Markov chain. Furthermore, with a simple extension of this scheme, the probabilistic transition function $p : S \times A \to D(S)$ of an MDP can also be represented. Since the set of actions A is finite, we can view p as a function $S \times A \times S \to [0, 1]$. For an MDP with 2^n states, and letting $k = \text{ceil}(\log_2 |A|)$, the probabilistic transition function p is equivalently seen as a function from $\{1, \ldots, 2^n\} \times \{1, \ldots, 2^k\} \times \{1, \ldots, 2^n\}$ to $\mathbb{R}$, which can easily be represented by an MTBDD over $2n + k$ variables.

MTBDDs are efficient because they are stored in reduced form, with duplicate nodes merged and redundant ones removed. Their size (number of nodes) is heavily dependent on the ordering of their Boolean variables. Although the problem of deriving the optimal ordering for a given MTBDD is an NP-hard problem, by using heuristics [65, 83], probabilistic models with a degree of regularity can be represented extremely compactly by MTBDDs.

Model checking of MDPs using MTBDDs. Once a model's MTBDD representation has been constructed, it can be analyzed, for example using value iteration to compute minimum and

maximum reachability and safety probabilities. This comprises two stages. First, a graph-based analysis is performed to find the states for which the corresponding probability is 0 or 1 [48]. This can be implemented using standard BDD techniques for calculating fix-points. Secondly, numerical computation is applied to compute probabilities for the remaining states. For this, standard iterative methods such as value iteration, can be implemented using standard MTBDD operations, including for example algorithms from [11, 29] for matrix-vector multiplication.

8.2 Symbolic MLA

In this section, we present a symbolic implementation of the MLA algorithm using MTBDDs. Before doing so, we highlight some important aspects of the implementation.

We first note that a potential obstacle in the use of MLA is that, although substantial savings in terms of storage for solution vectors can be made, there is still a need to store the probabilistic transition function of the MDP in full. A symbolic approach alleviates this problem: it is often the case that a very compact MTBDD representation of the probabilistic transition function of the MDP can be constructed.

Secondly, it is also common that *qualitative* probabilistic verification (i.e. checking for which states of the MDP the probabilities for a reachability/safety property are exactly 0 or 1) can be applied to much larger models than can be analyzed quantitatively. This is because qualitative properties can be model checked using only graph-based algorithms that operate on the underlying transition relation, allowing an efficient implementation with BDDs. This means that a symbolic version of MLA can also benefit from this: qualitative verification is applied to the full MDP *before* applying the MLA algorithm (this process is often referred to as *pre-computation*). Numerical computation need then only be done for states with a probability that is neither 0 or 1. Furthermore, states with probability 0 or 1 can be removed from the MDP

completely, reducing computation significantly and decreasing round-off errors.

Finally, we observe that symbolic techniques are very well-suited to MLA, in terms of the representation of solution vectors. Recall that, because of the way that MLA operates, it requires separate storage of the numerical solution vector for the current region being magnified (by algorithm MI see Section 7.2) and the lower/upper bounds for each region. Furthermore when the value for a state not in the current magnified region is required, the region contains that state must be determined before the relevant value can be looked up. Because of the way that MTBDDs exploit regularity, representing real-valued vectors with many similar values is often very efficient. This allows us to store the solution vector for all states of the MDP concurrently, avoiding potentially expensive partition look-ups. Since MLA considers each region sequentially, the solution vector will contain fewer distinct values than would be required for standard value iteration. Thus, we expect a symbolic implementation of MLA to be less memory-intensive than a symbolic version of value iteration.

8.2.1 Symbolic Magnifying-Lens Abstraction (SMLA)

The symbolic version of MLA is shown in Algorithm 11. As for standard MLA (Algorithm 9), the symbolic version is parameterized by operators $f, g \in \{\max, \min\}$ (used to select maximum/minimum reachability/safety properties) and convergence thresholds ε_{float} and ε_{abs}. The other parameter is a BDD T representing the set of target states (T in Algorithm 9). We also assume a BDD reach representing the set of reachable states of the MDP and an MTBDD trans representing its probabilistic transition function. In the latter, the MTBDD variables representing the rows (source states), columns (target states) and nondeterminism (actions) are denoted *rvars*, *cvars* and *ndvars*, respectively.

The first part of Algorithm 11 (lines 1-5) shows the use of BDD-based pre-computation steps [48, 83] in order to obtain the BDDs yes and no, representing the sets of states for which

the probability is exactly 1 or 0, respectively. If this covers all states of the MDP, no further work is required. Otherwise, rows corresponding to states in yes or no are removed from the probabilistic transition function trans (line 5). Here (and elsewhere in the algorithms) we use a simple infix notation to denote the application of binary operators (such as $\vee$ or $\times$) to BDDs or MTBDDs. This is done using the standard APPLY operator [21].

The remainder of Algorithm 11 comprises the symbolic version of MLA. We start with an initial partition R, returned by the CreateInitialPartition() routine (see Section 8.2.3 for details). The partition is implemented as a list of BDDs, each one representing a region in R. The main part of Algorithm 11 corresponds quite closely to the original MLA algorithm (Algorithm 9). Initialization of solution vectors (lines 8 and 9) is easily achieved using the MTBDD operation CONST(k) which returns the trivial MTBDD representing the real value k. Similarly, checking for convergence of the main loop can be done with the operation MAXDIFF($\mathsf{u}_1, \mathsf{u}_2$) which computes the maximum point-wise difference between MTBDDs u_1 and u_2).

The MTBDDs representing the lower (u^-) and upper (u^+) bounds for each region are computed by the SMI function, described below. After a global iteration terminates, the algorithm calls the Split(...) method to refine the regions for which the difference between the lower and upper bounds (u^- and u^+) is greater than ε_{abs}. After each refinement, the algorithm copies u^- values to u^+ for the reachability objectives and u^+ values to u^- for safety objectives.

8.2.2 Symbolic Magnified Iteration (SMI)

The core part of the MTBDD-based implementation of MLA is called *Symbolic Magnified Iteration (SMI)* and is shown in Algorithm 12. It performs a symbolic value iteration algorithm inside the region represented by BDD r from the current partition R. The algorithm is also passed the MTBDD trans′ representing the (filtered) probabilistic transition function of the MDP, the BDD T representing the set of target states, and the MTBDD u, which stores the (upper or

lower) bound for every state's corresponding region. The other parameters h, f, g and ε_{float}, are as for the non-symbolic version in Algorithm 10.

The algorithm initializes the solution vector v with the vector u (line 1) and then the MTBDD trans′ is filtered further to include only transitions for the current region (line 2). The loop (lines 3-12) updates the solution vector v until the results of two successive iterations differ less than ε_{float}. The first two lines of the loop perform a matrix-vector multiplication of the transition probability matrix of the MDP with (a permuted copy of) the solution vector v. This corresponds to the summations in line 6a of Algorithm 10. In line 7, the operator $g \in \{\max, \min\}$ is applied over the nondeterministic variables *ndvars* of the resulting MTBDD (the first part of line 6a from Algorithm 10). In line 8, the operator f is applied point-wise with the BDD T representing the target states (line 6b of Algorithm 10). Finally, the new solution vector v′ is computed by setting values for all states not in the current region (r) to their values in u, using the MTBDD operation ITE (If-Then-Else).

Once the while loop terminates, the algorithm computes the maximum (if h=max) or minimum (if h=min) value *val* of the region by using FindMax (or FindMin). Finally the algorithm returns a solution vector with value *val* for the current region and the old solution value from u for all other regions.

8.2.3 The Splitting Order

The creation of the initial partition and the way in which it is subsequently split are governed by two user parameters: *strat* and *level*. Splitting operations are based on a priority order $X_{ord} = \langle x_1, x_2, \ldots, x_n \rangle$ of the MTBDD variables representing the state space of the MDP. In the adaptive refinement scheme of MLA, each call to the routine Split subdivides a region into two using the next MTBDD variable from the order X_{ord} (we call this the *splitting index*). Since the MLA algorithm does not refine regions with $u^+(r) - u^-(r) \leq \varepsilon_{abs}$, after a refinement,

different regions may have different splitting indices.

The order X_{ord} is determined by the choice of a splitting strategy *strat*: either "*consecutive*" or "*interleaved*". In the default MTBDD variable ordering (for an MDP derived from a PRISM model), MTBDD variables are grouped according to the (model-level) variable to which they correspond and ordered consecutively. For *strat=consecutive*, we take X_{ord} to be this default ordering. For *strat=interleaved*, on the other hand, the MTBDD variables corresponding to different (model-level) variables are interleaved.

The initial creation of a partition (by routine CreateInitialPartition) is determined by $X_{ord} = \langle x_1, x_2, \ldots, x_n \rangle$ and the parameter *level*. Each region in the initial partition is created by splitting on MTBDD variables $x_1, x_2, \ldots, x_{level}$ (i.e. the splitting index for each region is *level*).

8.3 The Case Studies and Results

We have implemented the symbolic MLA algorithm within the probabilistic model checker PRISM and, in this section, present results for the following MDP case studies.

Inventory Problem. We have modeled an inventory as an MDP. The variable "*stock*" denotes the current number of items in the inventory and "*init*" denotes the initial item count. The variable "*time*" keeps track of time elapsing. At each time step, the demand of the item is 1 with a probability p and 0 with $1 - p$. The probability p is a function of current number of items present in the inventory. The manager of the inventory visits the inventory every 7 time units and he has two actions to choose from: either place an order or do not place one. The property we are checking is the "minimum probability that the stock reach its minimum amount within MAXTIME time units". In PCTL, the reachability property can be expressed

Example	Parameters	States	Trans	PRISM		MLA		
		10^3	10^3	Time	Node	Time	Node	Reg
Inventory	st=512, T=512	26	34	14	14K	15	2K	340
	st=1K, T=1K	106	135	54	26K	61	4K	676
	st=2K, T=2K	425	535	233	50K	270	9K	1,348
	st=4K, T=4K	1,698	2,130	896	99K	1,056	17K	2,692
	st=5K, T=5K	2,653	3,325	1,243	120K	1,424	21K	3,364
	st=10K, T=10K	10,605	13,275	7,118	241K	7,551	43K	3,363
Minefield	n=256,m=100	65	299	75	57K	263	8K	2,041
	n=512,m=200	262	1,128	627	91K	1,493	14K	4,164
	n=1024,m=300	1,048	4,316	3,625	127K	5,463	20K	6,324
Hotel Booking	c=127, b=63, T=15	131	645	4	30K	46	8K	903
	c=255, b=127, T=31	1,048	5,202	44	118K	1,013	37K	6,350
	c=511,b=255, T=31	4,194	20,889	2,072	373K	9,971	118K	25,491
Secretary	c=100, T=100	30	61	2	15K	7	3K	269
	c=100, T=200	90	180	3	17K	11	3K	345
	c=200, T=200	121	242	10	33K	27	6K	471
	c=300, T=400	451	903	24	55K	62	9K	463
	c=500, T=1000	2,252	4,502	88	106K	199	17K	733
	c=1000, T=2000	9,004	18,005	392	233K	802	32K	768
Zeroconf	N=4,M=32,K=4	26	50	88	127K	50	14K	22
	N=8, M=32, K=4	552	1,728	1,307	722K	650	49K	64
	N=8, M=128, K=4	2,092	6,552	3,221	857K	2,593	151K	19

Figure 8.1: Experimental results: Symbolic MLA, compared to PRISM

as $P_{\min=?}[\Diamond\ (stock{=}1 \wedge time{<}MAXTIME))]$.

Robot in a Minefield. We consider the problem of navigating an $n \times n$ minefield. The minefield contains m mines, each with coordinates (x_i, y_i), for $1 \leq i \leq m$, where $1 \leq x_i < n$, $1 \leq y_i < n$. We consider the problem of computing the maximal probability with which a robot can reach the target corner (n, n), from all $n \times n$ states. At interior states of the field, the robot can choose among four actions: *Up, Down, Left, Right;* at the border of the field, actions that lead outside of the field are missing. From a state $s = (x, y) \in \{1, \ldots, n\}^2$ with coordinates (x, y), each action causes the robot to move to square (x', y') with probability $q(x', y')$, and to "blow up" (move to an additional sink state) with probability $1 - q(x', y')$. For action *Right,* we have $x' = x + 1$, $y' = y$; similarly for the other actions. The probability $q(x', y')$ depends on the

strat	*level*	Node	Time (s)	Regions	*strat*	*level*	Node	Time(s)	Regions
cons	1	60K	50	191	*inter*	1	60K	254	942
cons	4	19K	57	191	*inter*	4	40K	255	942
cons	7	12K	60	214	*inter*	7	18K	258	946
cons	11	11K	95	752	*inter*	11	10K	307	1057
cons	15	13K	191	3043	*inter*	15	11K	441	2705

Figure 8.2: Effect of splitting strategy ('cons', 'inter' denote consecutive and interleaving respectively) and initial splitting index (Secretary: c=300, $MAXTIME$=400)

proximity to mines, and is given by

$$q(x', y') = \prod_i^m \exp\Big(-0.7 \cdot \Big((x' - x_i)^2 + (y' - y_i)^2\Big)\Big).$$

Optimal Stopping Game: Secretary Selection. We have modeled one application of the optimal stopping game. One boss starts interviewing c candidates for the post of secretary. After each interview, he can either select the candidate or continue the process with the remaining candidates. If the boss does not select the candidate, then the candidate is eliminated from the selection process. The variable "*time*" is used to keep track of the time that has elapsed.The boss can compare whether the current candidate is the best so far or if a better candidate was interviewed previously. If the current candidate is the best among all candidates seen, then the variable "*best*" is assigned to 1. The boss does not know the (merit) order of the candidates; hence we model assignment of the variable with a probabilistic update. The probability that the current one is the best among c candidates is set equal to $1/c$. If the boss selects a candidate, then the variable "*stop*" is assigned to 1. The property we are checking is the "maximum probability that the interviewer has selected a non-best candidate before the timeout". In PCTL, the reachability property can be expressed as $P_{\max=?}[\diamond\ (stop{=}1 \wedge best{=}0 \wedge time{<}MAXTIME)]$.

Hotel Booking Problem. We have modeled an instance of the over-booking problem for a hotel during a multiple-day conference. The conference-chairperson books b rooms for the registered participants in a hotel with v rooms. The variable "*days*" keeps track of days that

have elapsed since the start of the conference. The participants can appear at any day during the conference but some of the booked rooms remain vacant during the conference season due to "no-show" of the participants. The hotel manager takes this factor into account and over-books the hotel during the peak seasons. When he books a hotel room and the conference participant does not appear, the manager suffers a loss. Similarly he will be in trouble whenever he allows a non-conference visitor without keeping a room booked and the conference guest appears, requiring him to find an alternative room for the guest at higher cost. The arrival of the participants is probabilistic. The property we are checking will be the "maximum probability that a conference guest arrives within the duration of the conference and does not get a room". In PCTL, the reachability property can be expressed as $P_{\max=?}[\Diamond\ (v{=}0 \wedge b{>}0 \wedge days{<}MAXTIME)]$.

Zeroconf Protocol. The Zeroconf protocol [27] is used for the dynamic self-configuration of a host joining a network; it has been used as a test-bed for the abstraction method considered in [70]. We consider a network with N existing hosts, and M total IP addresses; protocol messages have a certain probability of being lost during transmission. The variable K denotes the maximum number of probes can be sent by the new host. We consider the problem of determining the maximal probability of a host eventually acquiring an IP address.

Results. Our experiments were run on an Intel 2.16 GHz machine with 2GB RAM. We used $\varepsilon_{float} = 0.01$, ε_{abs}=0.1 for both PRISM and MLA and, unless otherwise stated (see next section), an initial splitting index (*level*) of $\lfloor k/2 \rfloor$, where k is the number of MTBDD variables representing the MDP's state space. For the splitting strategy (*strat*), we used "*consecutive*" for all model, except the minefield.

Figure 8.1 summarizes the results for all case studies. The first two columns show the name and parameters of the MDP model. The third and fourth columns gives the number

of states and transitions for each model. The remaining columns show the performance of analyzing the MDPs, using both PRISM and symbolic MLA. In both cases, we give the total time required (which includes model building and model checking) and the peak MTBDD node count (which includes the partial transition relation and the solution vectors). For MLA, we also show the final number of generated regions. We used the MTBDD engine of PRISM, since (a) it is generally the best performing engine for MDPs; and (b) it is the only one that can scale to the size of models we are aiming towards. More detailed experimental data is available from:

`www.soe.ucsc.edu/~pritam/qest08.html`.

Discussion. The "Nodes" columns of Figure 8.1 demonstrate the efficiency of the symbolic implementation of MLA: the memory requirements are significantly lower than the equivalent statistics for PRISM's MTBDD engine. As discussed earlier in Section 8.2, this is due to the fact that MLA analyzes each region in isolation, resulting in a smaller number of distinct values in the solution vectors. For the Zeroconf example, this phenomenon actually results in MLA also outperforming PRISM in terms of solution time.

It is also clear, from the sizes of the MDPs in the table, that the symbolic version of MLA is able to handle MDPs considerably larger than were previously feasible for the existing explicit implementation of [50]. Thanks to this, another positive conclusion which we can draw from the results is that MLA generates relatively small numbers of regions for the analysis of even large MDPs.

Finally, we also experimented with different parameter values for the splitting strategy (*strat*) and initial splitting index (*level*). Figure 8.2 shows results for the secretary selection case study ($c = 300$ and $MAXTIME = 400$). For smaller values of the initial splitting index, there are less regions initially but these regions are relatively large, resulting in higher memory consumption. Increasing the splitting index produces smaller regions, which take less space and

time to analyze, however more global iterations are required, resulting in longer total solution times. Hence, in our results (Figure 8.1), we opted for a trade-off by using a splitting index close to $k/2$, where k is the number of MTBDD variables representing the state space.

For the results in Figure 8.2 (and for most of our case studies), the "*consecutive*" strategy performs better than the "*interleaved*" strategy, both in terms of memory usage, time and number of regions. For the minefield problem, however, the reverse is true. This is due to the "grid-like" nature of the model and the fact that the state-space is described by a pair of co-ordinates, x and y. It is more effective to refine the state space into square regions of the grid.

8.4 Conclusion

We have presented a symbolic implementation of the magnifying-lens abstraction (MLA) technique of [50], using the multi-terminal binary decision diagram (MTBDD) data structure. This was implemented in the probabilistic model checker PRISM and applied to a range of MDP case studies. The results demonstrate that symbolic MLA yields significant gains in memory usage over standard (symbolic) implementations of MDP verification, as provided by PRISM. Furthermore, in some cases this also produce better performance in terms of time. Our results also show that symbolic MLA can be applied to much larger MDPs than its explicit counterpart.

In the future, we plan to make a comparison of our approach with other MDP abstraction techniques, including the game-based approach of [70]. We also plan to investigate the integration of more advanced symbolic representations of state space partitions, such as [53].

Algorithm 11 SMLA($\mathsf{T}, f, g, \varepsilon_{float}, \varepsilon_{abs}$) Symbolic Magnifying-Lens Abstraction

1. **if** $g = \max$
2. **then** $\mathsf{no} :=$ PROB0A(T) ; $\mathsf{yes} :=$ PROB1E(T)
3. **else** $\mathsf{no} :=$ PROB0E(T) ; $\mathsf{yes} :=$ PROB1A(T)
4. **if** $\mathsf{no} \vee \mathsf{yes} = \mathsf{reach}$ **then return** yes
5. $\mathsf{trans}' := \mathsf{trans} \times \neg(\mathsf{no} \vee \mathsf{yes})$
6. $R :=$ CreateInitialPartition()
7. **if** $f = \max$
8. **then** $\mathsf{u}^- := \mathsf{u}^+ :=$ CONST(0)
9. **else** $\mathsf{u}^- := \mathsf{u}^+ :=$ CONST(1)
10. **loop**
11. **repeat**
12. $\hat{\mathsf{u}}^+ := \mathsf{u}^+$; $\hat{\mathsf{u}}^- := \mathsf{u}^-$
13. **for each** $\mathsf{r} \in R$ **do**
14. $\hat{\mathsf{u}}^+ :=$ SMI($\mathsf{r}, R, \mathsf{trans}', \mathsf{yes}, \hat{\mathsf{u}}^+, \max, f, g, \varepsilon_{float}$)
15. $\hat{\mathsf{u}}^- :=$ SMI($\mathsf{r}, R, \mathsf{trans}', \mathsf{yes}, \hat{\mathsf{u}}^-, \min, f, g, \varepsilon_{float}$)
16. **end for**
17. **until** MAXDIFF($\mathsf{u}^+, \hat{\mathsf{u}}^+$) $\leq \varepsilon_{float}$ &
MAXDIFF($\mathsf{u}^-, \hat{\mathsf{u}}^-$) $\leq \varepsilon_{float}$
18. **if** MAXDIFF($\mathsf{u}^+, \mathsf{u}^-$) $\geq \varepsilon_{abs}$
19. **then** $R, \mathsf{u}^-, \mathsf{u}^+ :=$ Split($R, \mathsf{u}^-, \mathsf{u}^+, \varepsilon_{abs}$)
20. **else return** $(\mathsf{u}^- + \mathsf{u}^+)/2$
21. **end if**
22. **if** $f = \max$ **then** $\mathsf{u}^+ := \mathsf{u}^-$ **else** $\mathsf{u}^- := \mathsf{u}^+$
23. **end loop**

Algorithm 12 SMI($\mathsf{r}, R, \mathsf{trans}', \mathsf{T}, \mathsf{u}, h, f, g, \varepsilon_{float}$) Symbolic Magnified Iteration

1. $\mathsf{v} := \mathsf{u}$
2. $\mathsf{trans}'' := \mathsf{trans}' \times \mathsf{r}$
3. $done :=$ **false**
4. **while** ($done$!= **true**) **do**
5. $\quad \mathsf{tmp} :=$ PERMUTE(v, $rvars$, $cvars$)
6. $\quad \mathsf{tmp} :=$ MVMULT(trans'', tmp)
7. $\quad \mathsf{tmp} :=$ REPLACE(g, tmp, $ndvars$)
8. $\quad \mathsf{tmp} :=$ APPLY(f, tmp, T)
9. $\quad \mathsf{v}' :=$ ITE(r, tmp, u)
10. $\quad$ **if** MAXDIFF(v', v) $< \varepsilon_{float}$ **then** $done :=$ **true**
11. $\quad \mathsf{v} := \mathsf{v}'$
12. **end while**
13. **if** ($h = \max$)
14. $\quad$ **then** $val :=$ FINDMAX(ITE(r, v, CONST(0)))
15. $\quad$ **else** $val :=$ FINDMIN(ITE(r, v, CONST(1)))
16. **return** ITE(r, CONST(val), u)

Chapter 9

Conclusions

9.1 Summary

The current trend in software and system engineering is towards *component-based design*. In this method, a number of design units called *components* make a complex design. Components are typically open systems that have inputs provided by other components and provide inputs to other components. Designers face a number of design issues to create a complex design from these components. A designed system, expected to achieve a series of tasks following its specification, may not behave properly due to the following reasons. Firstly, one or more components may contain bugs and behave in an undesirable way. Secondly, components make assumptions on their environment, and assume that the actual conditions will meet these assumptions. A number of bug-free components may not work together if their input assumptions are violated. Hence, verification of a complex system-design can be reduced to the verification of the components and communication among them.

The interaction between components in a design can be modeled via games, and a large volume of studies on design and verification shows how games can be used to analyze component

compatibility and system correctness. However, while games provide an appropriate, mathematical model for interaction, solving the games is often impossible with current algorithms, due to the large state-space of games representing practical components, together with the inherent complexity of game-solving techniques. In this thesis, we propose algorithms for the efficient analysis of games with large state spaces.

We present two novel algorithm families in the dissertation: (1) Game-based Three Valued Abstraction (GTVA) for two-player games/transition systems, and (2) Magnifying Lens Abstraction (MLA) for Markov Decision Processes (MDPs). GTVA evaluates the winning objectives on the abstract game-model in three-valued style (*yes*, *no*, *maybe*) and refines the abstraction by adding more details to the *maybe* abstract states. However, other approaches construct abstract models; thus verification becomes extremely expensive. We describe how to achieve efficient enumerative and BDD-based symbolic implementations of the algorithm. MLA partitions the state-space of MDP into regions and then computes upper and lower bounds on the regions, rather than on the concrete states. MLA iterates over the regions to evaluate these limits and considers the concrete states of each region in turn, as if one were moving a magnifying lens across the abstraction and viewing the concrete states corresponding to the current region. The algorithm refines the regions in an adaptive manner, splitting regions where we need more details until the difference between the bounds is smaller than a user-given accuracy. We also provide a symbolic form of algorithm MLA (SMLA).

We have implemented the proposed algorithms, and we have applied them to real-life applications, including planning, protocol verification, and interface synthesis for software libraries. The symbolic three-valued algorithms for reachability, safety, compatibility, and refinement properties have been implemented in the tool TICC; case-studies illustrate the accuracy and efficiency benefits of the GTVA algorithms over other approaches. We have implemented the symbolic version of MLA in the tool PRISM. The experimental results indicate that MLA can

provide accurate answers, with savings in the memory requirements. These algorithms promise to make the analysis of practical component-based designs possible by pushing the limits of the size of games that can be solved.

9.2 Future Directions

The difference between the design complexity and the validation capacity will increase in future. As verification researchers, we need to exercise more effective techniques to cope with the pressure. Scalable verification and testing techniques will continue to play a vital role in future. I want to devote my future research to seek a number of scalable verification and test-case generation techniques to bridge the increasing gap. In addition to that, there is a common trend towards multi-core designs and multi-threaded programs. I want to contribute to verification and testing of concurrent designs. However, the list is not exhaustive. The future focus lies in compact data structures, a combination of techniques, and concurrent designs.

SMT for Abstraction-Refinement and Test-case Generation: Currently, Satisfiable Modulo Theories (SMT) solvers have become the state-of-the-art solvers for the model-checking case-studies. I have implemented all algorithms in the thesis using symbolic data structure such as, BDDs (and its various extensions). Although the canonicity property of BDDs is particularly useful, the space-requirement restricts its use in real-life case-studies. I want to modify three-valued abstraction-refinement algorithms to suit the queries of the solvers, and make the algorithms more scalable. I also want to apply SMT techniques in the test case/stimuli generation algorithms.

Combination of Scalable Techniques : I have worked on various techniques. The combination of two or more approaches makes the algorithm scalable. For example, I developed

symbolic algorithms for three-valued abstraction-refinement techniques. However, integration of the scalable techniques may require careful adjustment. I want to work on a combination of different techniques in the future.

Interfaces to Test-cases : In the interface synthesis problem, we have only focused to create an interface graph. I want to extend the framework to the testing of parameterized library functions. The global state-space in the symbolic domain will provide the (symbolic) parameter ranges to the functions. Initially, I want to work on sequential programs. Later I want to consider an extension of the framework to concurrent programs (where each function is sequential, but the system may have more than one function active at a given time). This project can also be useful in hardware verification field. The interface can be used as a permissive set of test-benches in the hardware systems.

Combination of Static and Dynamic Techniques : There is a research trend to combine static and dynamic techniques to make the validation more scalable. All problems cannot be caught by static analysis (e.g. array out of bound, buffer overflow). Moreover, the static techniques like model checking are not particularly cost-effective techniques due to exhaustive search of state-spaces. Hence combination of static and run-time verification techniques are more pragmatic approaches. I will explore different algorithms that interleave static (property driven, model-checking-based test-case generation), and dynamic (executing the test case) in this direction.

Bibliography

[1] B. Adler, L. de Alfaro, L. D. D. Silva, M. Faella, A. Legay, V. Raman, and P. Roy. TICC: a tool for interface compatibility and composition. In *CAV 06: Proc. of 18th Conf. on Computer Aided Verification*, volume 4144 of *Lect. Notes in Comp. Sci.*, pages 59–62. Springer-Verlag, 2006.

[2] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society Press, 1999.

[3] R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. 27th ACM Symp. Theory of Comp.*, 1995.

[4] R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. 27th Ann. ACM Symp. Theory of Computing*, pages 363–372, 1995.

[5] R. Alur, L. de Alfaro, T. Henzinger, and F. Mang. Automating modular verification. In *CONCUR 99: Concurrency Theory*, volume 1664 of *Lect. Notes in Comp. Sci.*, pages 82–97. Springer-Verlag, 1999.

[6] R. Alur and T. Henzinger. Modularity for timed and hybrid systems. In *CONCUR 97:*

Concurrency Theory. 8th Int. Conf., volume 1243 of *Lect. Notes in Comp. Sci.*, pages 74–88. Springer-Verlag, 1997.

[7] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[8] R. Alur, T. Henzinger, and O. Kupferman. Alternating time temporal logic. *J. ACM*, 49:672–713, 2002.

[9] R. Alur, A. Itai, R. P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Inf. Comput.*, 118(1):142–157, 1995.

[10] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, 2005.

[11] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Journal of Formal Methods in System Design*, 10(2/3):171–206, 1997.

[12] C. Baier and H. Hermanns. Weak bisimulation for fully probabilistic processes. In *CAV 97: Proc. of 9th Conf. on Computer Aided Verification*, volume 1254 of *Lect. Notes in Comp. Sci.*, pages 119–130. Springer-Verlag, 1997.

[13] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[14] J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[15] D. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, 1995. Volumes I and II.

[16] D. Beyer, T. A. Henzinger, and V. Singh. Three Algorithms for Interface Synthesis: A Comparative Study. Technical report, 2006.

[17] D. Beyer, T. A. Henzinger, and V. Singh. Algorithms for interface synthesis. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*, LNCS 4590, pages 4–19. Springer-Verlag, Berlin, 2007.

[18] A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. In *FATES*, pages 32–46. Springer, 2005.

[19] A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical Report MSR-TR-2005-04, Microsoft Research, January 2005. Short version of this report was presented at FATES 2005.

[20] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 187–193. Springer, 2001.

[21] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[22] J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[23] C. Campbell and M. Veanes. State exploration with multiple state groupings. In D. Beauquier, E. Börger, and A. Slissenko, editors, *12th International Workshop on Ab-*

stract State Machines, ASM'05, March 8–11, 2005, Laboratory of Algorithms, Complexity and Logic, University Paris 12 - Val de Marne, Créteil, France, pages 119–130, 2005.

[24] A. Chackrabarti, L. de Alfaro, M. Jurdziński, K. Chatterjee, T. Henzinger, and F. Mang. CHIC: Checker for interface compatibility, 2003. www-cad.eecs.berkeley.edu/ tah/chic/.

[25] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Synchronous and bidirectional component interfaces. In *CAV*, pages 414–427, 2002.

[26] K. Chatterjee, L. de Alfaro, and T. A. Henzinger. Trading memory for randomness. In *QEST*, pages 206–217, 2004.

[27] S. Cheshire, B. Adoba, and E. Gutterman. Dynamic configuration of ipv4 link local addresses (internet draft).

[28] A. Church. Logic, arithmetics, and automata. In *Proc. International Congress of Mathematicians, 1962*, pages 23–35. Institut Mittag-Leffler, 1963.

[29] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10((2/3):149–169, 1997.

[30] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *International Workshop for Logic Synthesis*, 1993.

[31] E. Clarke, O. Grumberg, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Proc. of 12th Conf. on Computer Aided Verification*, Lect. Notes in Comp. Sci. Springer-Verlag, 2000.

[32] P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *Proc. of PAPM/PROBMIV*, volume 2165 of *Lect. Notes in Comp. Sci.*, pages 39–56. Springer-Verlag, 2001.

[33] L. de Alfaro. *Formal Verification of Probabilistic Systems.* PhD thesis, Stanford University, 1997. Technical Report STAN-CS-TR-98-1601.

[34] L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In *CONCUR 99: Concurrency Theory. 10th Int. Conf.*, volume 1664 of *Lect. Notes in Comp. Sci.*, pages 66–81. Springer-Verlag, 1999.

[35] L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification (Theory in Practice)*, volume 2772 of *Lect. Notes in Comp. Sci.*, pages 269–289. Springer-Verlag, 2003.

[36] L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.

[37] L. de Alfaro, B. Ader, M. Faella, A. Legay, V. Raman, P. Roy, and L. Dias Da Silva. TICC: Tool for interface compatibility checking, 2006. http://dvlab.cse.ucsc.edu/dvlab/Ticc.

[38] L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B. Wang. Mocha: A model checking tool that exploits design structure. In *ICSE 01: Proceedings of the 23rd International Conference on Software Engineering*, pages 835–836, 2001.

[39] L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *FROCOS: Frontiers of Combining Systems, Proc. of the 5th Intl. Workshop*, volume 3717 of *Lect. Notes in Comp. Sci.*, pages 81–105. Springer-Verlag, 2005.

[40] L. de Alfaro, M. Faella, T. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *CONCUR 03: Concurrency Theory. 14th Int. Conf.*, volume 2761 of *Lect. Notes in Comp. Sci.*, pages 144–158. Springer-Verlag, 2003.

[41] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proc. 19th IEEE Symp. Logic in Comp. Sci.*, pages 170–179, 2004.

[42] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM Press, 2001.

[43] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT 01: 1st Intl. Workshop on Embedded Software*, volume 2211 of *Lect. Notes in Comp. Sci.*, pages 148–165. Springer-Verlag, 2001.

[44] L. de Alfaro and T. Henzinger. Interface-based design. In *Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School.* Kluwer, 2004.

[45] L. de Alfaro, T. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In *CONCUR 01: Concurrency Theory. 12th Int. Conf.*, Lect. Notes in Comp. Sci. Springer-Verlag, 2001.

[46] L. de Alfaro, T. Henzinger, and F. Mang. Detecting errors before reaching them. In *CAV 00: Proc. of 12th Conf. on Computer Aided Verification*, volume 1855 of *Lect. Notes in Comp. Sci.*, pages 186–201. Springer-Verlag, 2000.

[47] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM, 2001.

[48] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. In *TACAS 00: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 395–410. Springer-Verlag, 2000.

[49] L. de Alfaro and R. Majumdar. Quantitative solution of omega-regular games. *Journal of Computer and System Sciences*, 68:374–397, 2004.

[50] L. de Alfaro and P. Roy. Magnifying-lens abstraction for Markov decision processes. In *Proc. CAV'07*, volume 4590 of *LNCS*, pages 325–338. Springer-Verlag, 2007.

[51] L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2007.

[52] T. Dean and R. Givan. Model minimization in markov decision processes. In *AAAI/IAAI*, pages 106–111, 1997.

[53] S. Derisavi. A symbolic algorithm for optimal Markov chain lumping. In O. Grumberg and M. Huth, editors, *Proc. TACAS'07*, volume 4424 of *Lect. Notes in Comp. Sci.*, pages 139–154. Springer-Verlag, 2007.

[54] C. Derman. *Finite State Markovian Decision Processes.* Academic Press, 1970.

[55] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* The MIT Press, 1989.

[56] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker. A formal analysis of Bluetooth device discovery. *Int. Journal on Software Tools for Technology Transfer*, 8(6):621–632, 2006.

[57] E. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *Proc. 32nd IEEE Symp. Found. of Comp. Sci.*, pages 368–377. IEEE Computer Society Press, 1991.

[58] H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In A. Valmari, editor, *13th International SPIN Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*. Springer, 2006.

[59] I. Gilboa. Expected utility with purely subjective non-additive probabilities. *Journal of Mathematical Economics*, 16:65–88, 1987.

[60] I. Gilboa and D. Schmeidler. Additive representations of non-additive measures and the choquet integral. Discussion Papers 985, Northwestern University, Center for Mathematical Studies in Economics and Management Science, 1992. available at http://ideas.repec.org/p/nwu/cmsems/985.html.

[61] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.

[62] T. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *30th Int. Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2719, pages 886–902. LNCS, 2003.

[63] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.

[64] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th*

ACM SIGSOFT international symposium on Foundations of software engineering, pages 31–40, New York, NY, USA, 2005. ACM.

[65] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming*, 56(1-2):23–67, 2003.

[66] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS 06: Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lect. Notes in Comp. Sci.*, pages 441–444. Springer-Verlag, 2006.

[67] M. Huth. On finite-state approximations for probabilistic computational-tree logic. *Theor. Comp. Sci.*, 346(1):113–134, 2005.

[68] C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design and Process Technology, IDPT'02*, Pasadena, California, June 2002.

[69] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey, 1996.

[70] M. Kwiatkowska, G. Norman, and D. Parker. Game-based abstraction for markov decision processes. In *Proc. of QEST: Quantitative Evaluation of Systems*, pages 157–166. IEEE Computer Society, 2006.

[71] M. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In T. Ball and R. Jones, editors, *Proc. CAV'06*, volume 4114 of *Lect. Notes in Comp. Sci.*, pages 234–248. Springer-Verlag, 2006.

[72] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In G. Berry, H. Comon,

and A. Finkel, editors, *Proc. CAV'01*, volume 2102 of *LNCS*, pages 194–206. Springer-Verlag, 2001.

[73] E. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, 2001.

[74] X. Leroy. Objective caml. http://caml.inria.fr/ocaml/index.en.html.

[75] H. Li and C. Lam. Using anti-ant-like agents to generate test threads from the uml diagrams. In *Proc. Testcom 2005*, LNCS. Springer, 2005.

[76] N. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

[77] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proc. of 12th Annual Symp. on Theor. Asp. of Comp. Sci.*, volume 900 of *Lect. Notes in Comp. Sci.*, pages 229–242. Springer-Verlag, 1995.

[78] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[79] D. Martin. An extension of Borel determinacy. *Annals of Pure and Applied Logic*, 49:279–293, 1990.

[80] A. McIver and C. Morgan. *Abstraction, Refinement, and Proof for Probabilistic Systems.* Monographs in Computer Science. Springer-Verlag, 2004.

[81] D. Monniaux. Abstract interpretation of programs as Markov decision processes. *Science of Computer Programming*, 58(1–2):179–205, 2005.

[82] G. Necula, S. McPeak, W. Weimer, R. To, and A. Bhargava. CIL: Infrastructure for C program analysis and transformation.

[83] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, 2002.

[84] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE*, pages 225–240, 1999.

[85] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *SIGMETRICS '85: Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 147–154, New York, NY, USA, 1985. ACM Press.

[86] A. Plilippou, I. Lee, and O. Sokolsky. Weak bisimulation for probabilistic systems. In *CONCUR 00: Concurrency Theory. 11th Int. Conf.*, volume 1877 of *Lect. Notes in Comp. Sci.*, pages 334–349. Springer-Verlag, 2000.

[87] A. Pnueli and R. Rosner. Distributed-reactive systems are hard to synthesize. In *Proc. 31th IEEE Symp. Found. of Comp. Sci.*, pages 746–757, 1990.

[88] PRISM web site. www.prismmodelchecker.org.

[89] C. S. Păsăreanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In *Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005.

[90] P. Ramadge and W. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25:206–230, 1987.

[91] P. Roy, D. Parker, G. Norman, and L. de Alfaro. Symbolic magnifying lens abstraction in markov decision processes. In *QEST*, pages 103–112, 2008.

[92] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems,* P. Panangaden and F. van Breugel (eds.), volume 23 of *CRM Monograph Series.* American Mathematical Society, 2004.

[93] D. Schmeidler. Integral representation without additivity. *Proceedings of the American Mathematical Society*, 97:255–261, 1986.

[94] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, MIT, 1995. Technical Report MIT/LCS/TR-676.

[95] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In *CONCUR 94: Concurrency Theory. 5th Int. Conf.*, volume 836 of *Lect. Notes in Comp. Sci.*, pages 481–496. Springer-Verlag, 1994.

[96] S. Shoham. A game-based framework for ctl counter-examples and 3-valued abstraction-refinement. Master's thesis, TECHNION - Israel Institute of Technology, 2003. Paper with O. Grumberg in Proceedings of CAV 2003.

[97] S. Shoham. A game-based framework for CTL counter-examples and 3-valued abstraction-refinement. In *CAV 03: Proc. of 15th Conf. on Computer Aided Verification*, Lect. Notes in Comp. Sci., pages 275–287. Springer-Verlag, 2003.

[98] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *TACAS*, volume 2988 of *Lect. Notes in Comp. Sci.*, pages 546–560. Springer-Verlag, 2004.

[99] S. Shoham and O. Grumberg. 3-valued abstraction: More precision at less cost. In *Proc. 21st IEEE Symp. Logic in Comp. Sci.*, pages 399–410, 2006.

[100] F. Somenzi. Cudd: Cu decision diagram package. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.

[101] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT, 1998. URL: http://www.cs.ualberta.ca/ sutton/book/ebook/the-book.html.

[102] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[103] J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.

[104] M. van der Bij, A. Rensink, and J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing: Third International Workshop, FATES 2003*, volume 2931 of *LNCS*, pages 86–100. Springer, 2004.

[105] M. Vardi. Automatic verification of probabilistic concurrent finite-state systems. In *Proc. 26th IEEE Symp. Found. of Comp. Sci.*, pages 327–338. IEEE Computer Society Press, 1985.

[106] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 273–282. ACM, 2005.

[107] M. Veanes, P. Roy, and C. Campbell. Online testing with reinforcement learning. In *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 240–253. Springer, 2006.