Abstraction-based Model Checking using Modal Transition Systems

Patrice Godefroid[1], Michael Huth[2], and Radha Jagadeesan[⋆3]

[1] Bell Laboratories, Lucent Technologies, god@bell-labs.com
[2] Computing and Information Sciences, Kansas State University, huth@cis.ksu.edu
[3] Department of Computer Science, Loyola University of Chicago, radha@cs.luc.edu

Abstract. We present a framework for automatic program abstraction that can be used for model checking any formula of the modal mu-calculus. Unlike traditional *conservative* abstractions which can only prove universal properties, our framework can both prove and disprove any formula including arbitrarily nested path quantifiers. We discuss algorithms for automatically generating an abstract *Modal Transition System* (MTS) by adapting existing predicate and cartesian abstraction techniques. We show that model checking arbitrary formulas using abstract MTSs can be done at the same computational cost as model checking universal formulas using conservative abstractions.

1 Introduction

There are essentially two approaches for extending the applicability of model checking to programs written in general-purpose programming languages such as C or Java. The first approach consists of adapting existing model-checking techniques into a form of systematic testing that is applicable to processes executing arbitrary code (e.g., [16]); although sound, this approach is inherently incomplete for large systems. The second approach consists of automatically extracting a model out of a program by a static analysis of its code, and of analyzing this model using existing model-checking techniques (e.g., [1, 9]); although automatic abstraction can be complete, this approach is generally unsound since abstraction usually introduces unrealistic behaviors that may yield to spurious errors being reported when analyzing the model.

In this paper, we study the latter approach and show how automatic abstraction can be performed in such a way that it yields verification results whose completeness and soundness can be both guaranteed. We also show how automatic abstraction can be applied to check arbitrary formulas of the modal mu-calculus [22], thus including negation and arbitrarily nested path quantifiers. Maybe surprisingly, both extensions can be implemented in combination with existing abstraction techniques without incurring any significant computational overhead. Our algorithms could be used to extend the scope of existing tools for (conservative) automatic abstraction such as SLAM [1] and Bandera [9],

⋆ Supported by NSF CCR-9901071.

which currently support the verification of universal properties only [7]. Our algorithms to construct abstract transition systems can also be used in the context of the verification of arbitrary modal mu-calculus formulas with methods based on theorem-proving [30].

Allowing the specification of arbitrary formulas with nested path quantifiers makes it possible to express more elaborate properties of the temporal behavior of a reactive program, such as "for all possible input values, there exists an execution path of the system that allows the user to restart the service". Unfortunately, the verification of such properties necessitates the relation between the concrete program and an abstract program to be more constraining than a simulation relation [26, 25]. Although bisimulation [28, 27] over Labeled Transition Systems (LTSs) reflects all such general properties [18], it is persuasively argued in [24, 23] to be ill-suited for our context: as an equivalence relation, it confines the choice of an abstraction to the implementation's equivalence class, which is too limiting to allow for compact abstractions.

For this reason, we use Modal Transition Systems (MTS) [24, 23] for representing abstract systems in order to allow their specifications to be partially defined. MTSs are LTSs with two kinds of transitions, termed `may` and `must` transitions, satisfying the consistency condition that every `must`-transition is also a `may`-transition. A MTS can be "refined" by preserving at least all `must`-transitions (and maybe adding some) while eliminating some `may`-transitions. Since this refinement preorder on MTSs preserves all properties expressible in the modal mu-calculus, we can verify any such properties on the source (concrete) program by verifying these on any abstract MTS that is refined by this concrete program; conversely, if there exists a behavior of the abstract MTS that refutes the property, the existence of a refuting behavior of the concrete program is also immediately guaranteed.

An alternative representation for abstract systems is the *partial Kripke structure* [4, 5]. Partial Kripke structures are Kripke structures whose states are labeled with atomic propositions that can have any of three possible truth values: `true`, `false` or `unknown`. Partial Kripke structures are closely related to MTSs since the transition relation of a MTS can be viewed as a function associating each transition with one of three possible values: `must`-transitions correspond to the value `true`, `may`-transitions that are not `must`-transition are mapped to `unknown`, and absent transitions render `false`. It can be shown that any partial Kripke structure can be translated into an equivalent MTS, and vice versa. This correspondence makes it possible to apply the results of [4, 5] (in particular, model-checking algorithms and complexity bounds) to the context of MTSs. Conversely, the abstraction techniques developed in this paper can be adapted to the context of partial Kripke structures.

A crucial aspect of our model-checking framework is that it not only permits the abstraction of complete programs, but also the refinement of *partially* specified abstract programs by more concrete abstract programs, to adequately accommodate the incremental process of building more detailed abstractions by successive approximations, as used in SLAM or Bandera for instance.

We develop an expressive and flexible relational calculus for the sound specification of MTSs as abstractions. This calculus adapts the definitions of [12, 13] to *partially* specified systems and is complete in the sense that it can specify every refinement of MTSs. In particular, any abstract interpretation of data values extends to a relational abstraction expressible in the calculus. In this calculus, we specify two standard abstractions of abstract interpretation [10], namely predicate abstraction [17, 14, 32] and cartesian abstraction [3, 1] (also known as "independent attribute analysis"), and describe their implementations:

- When applied to `may`-transition relations only, the specifications and implementations we present coincide with traditional "conservative" abstraction.
- We discuss how these specifications can be implemented using standard tools (automatic theorem proving for quantifier-free first-order logic and BDDs), except for the use of Ternary Decision Diagrams (TDDs) [31] for cartesian abstraction. We show that the computational cost of constructing a `must`-transition relation is the same as that of constructing a `may`-transition relation.
- We show that our implementations are sound and (relatively) complete[1] with respect to their specifications in our calculus. Moreover, they conveniently model approximations in calls to a theorem prover as under-approximations of `must`-transitions and over-approximations of `may`-transitions.
- We prove that abstraction refinement is incremental for MTSs built using *cartesian* abstraction.

Predicate abstraction [17, 14, 32] is based on a set of predicates, $\Phi \stackrel{\mathrm{def}}{=} \{\phi_1, \ldots, \phi_n\}$, typically quantifier-free formulas of first-order logic (e.g. `(x == y+1) || (x < y-5)`). An abstract state is induced by n-ary conjunctions, called *monomials*, with each predicate ϕ_i contributing either ϕ_i or $\neg\phi_i$. This abstraction identifies concrete states that satisfy the same predicates in Φ.

Given a set of states represented by a formula of quantifier-free first-order logic ψ, the set ψ' of abstract `may`-successors states is defined as the disjunction of all monomials η such that $\mathtt{post}(\psi) \wedge \eta$ is satisfiable [17, 14].[2] Computing ψ' can be done using automatic theorem proving for quantifier-free formulas, and [14] shows how to use a representation based on BDDs [6] at a propositional level to compactly represent the construction of ψ' as a disjunction of conjunctions. We can compute `must`-transitions by dualizing, in a logical sense, the above construction: for ψ as above, we show that the set of `must`-successors is the disjunction of all monomials η such that $\psi \wedge \tilde{\mathtt{pre}}(\neg\eta)$ is unsatisfiable.[3]

Unfortunately, this approach is not incremental: adding a new predicate ϕ_{n+1} to Φ may not yield a refinement of the abstraction, and hence the entire abstraction may need to be recomputed. This shortcoming can be eliminated at the expense of enlarging the abstract state space: states are now built as disjunctions of abstract states from predicate abstraction. Using disjunctions can yield

[1] Which are perforce relative to the completeness of the underlying theorem prover.
[2] $\mathtt{post}(\psi)$ is the set of immediate successor states of states satisfying ψ.
[3] $\tilde{\mathtt{pre}}(\neg\eta)$ is the weakest precondition of states satisfying $\neg\eta$.

a `must`-component that is more precise than the one obtained from predicate abstraction, but can also be much more expensive: for n predicates, an abstraction using disjunctions can have 2^{2^n} states. This tradeoff between cost and precision is discussed in [8].

This limitation motivates the next layer of approximation: cartesian abstraction, which can be used on top of predicate abstraction in order to approximate sets of n-tuples by n-tuples of sets. We modify the work of [14] to synthesize abstract states and abstract `may`-successors for this composite abstraction, replacing BDDs by TDDs [31]. Then we construct `must`-transitions by dualizing, in the logical sense, the construction of `may`-transitions using cartesian abstraction.

We complete this framework with an algorithm for model checking any modal mu-calculus formula on an abstract MTS. Following [4, 5], any (three-valued) model-checking problem on MTSs can be reduced to two traditional (two-valued) model-checking problems on regular LTSs.

The rest of the paper is organized as follows. Section 2 discusses background material on MTSs. Section 3 formally develops a relational calculus of abstractions and proves a basic result that permits the methods of analysis of this paper. In Section 4, we apply these methods to predicate and cartesian abstraction and prove that cartesian abstraction allows for incremental refinement. Section 5 discusses three-valued model-checking for MTSs, and Section 6 concludes.

2 Background: Abstract Modal Transition Systems

MTSs [24, 23] are defined from labeled transition systems.

Definition 1 (Labeled transition systems). *A* labeled transition system *[27] (LTS) is a tuple $\mathcal{K} = (\Sigma_K, \mathtt{Act}, \longrightarrow)$, where Σ_K is a set of states,* $\mathtt{Act}$ *is a set of action symbols, and $\longrightarrow\ \subseteq \Sigma_K \times \mathtt{Act} \times \Sigma_K$ is a transition relation. We call $\mathcal{K}$* finitely-branching *if for each $s \in \Sigma_K$, the set $\{s' \in \Sigma_K \mid \exists \alpha \in \mathtt{Act}\colon (s, \alpha, s') \in\ \longrightarrow\}$ is finite.*

A strategy to reason about a complex program represented by an LTS $\mathcal{C}$ consists of (i) generating from $\mathcal{C}$ an abstract LTS $\mathcal{A}$, (ii) checking whether $\mathcal{A}$ satisfies a behavioral property ϕ, and (iii) transferring those results to the original program $\mathcal{C}$. For (i) and (iii), standard practice [10, 7] is to construct some $\mathcal{A}$ such that the initial states of $\mathcal{C}$ and $\mathcal{A}$ are related by a *simulation*.

Definition 2 (Simulation). *A relation $\rho \subseteq \Sigma_C \times \Sigma_A$ is a* simulation *[26] iff for any $c\,\rho\,a$ and $c \to^{\alpha} c'$ there is some $a' \in \Sigma_A$ such that $a \to^{\alpha} a'$ and $c'\,\rho\,a'$.*

The temporal logic $L_\forall$ whose abstract syntax is

$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid (\forall \alpha)\phi \mid \nu Z.\phi \qquad (1)$$

with $\alpha \in \mathtt{Act}$, variables $Z \in \mathtt{Var}$ for the greatest fixed point $\nu Z.\phi$, and usual semantics, expresses *universal properties* [29]. We assume here the semantics of (closed) formulas over LTSs is defined as sets of states. For instance, the semantics of $(\forall\alpha)\phi$ is:

$$[\![(\forall\alpha)\phi]\!] \stackrel{\text{def}}{=} \{s \in \Sigma_K \mid \text{for all } s' \in \Sigma_K,\ s \to^{\alpha} s' \text{ implies } s' \in [\![\phi]\!]\}.$$

A simulation relation $c \,\rho\, a$ ensures that $a \in [\![\phi]\!]$ (read "a satisfies ϕ") implies $c \in [\![\phi]\!]$. Thus, we may verify any universal property $\phi \in L_\forall$ (such as "For all paths, nothing bad will happen") at c by (i) computing an abstract model $\mathcal{A}$, (ii) establishing a simulation ρ satisfying $c \,\rho\, a$, and (iii) verifying ϕ at a. Unfortunately, a negative check $a \notin [\![\phi]\!]$ does not imply anything about the truth or falsity of $c \notin [\![\phi]\!]$. At most, debugging information obtained from such a negative check may be used to construct a more concrete version of $\mathcal{A}$ (a refinement), hoping that this more precise model either renders a positive check or that refined debugging information eventually "applies" to $\mathcal{C}$ as well.

In this paper, we argue that a better approach consists of using MTSs instead of LTSs for representing abstractions of LTSs.

Definition 3 (MTS). *A* MTS *[24] is a pair* $\mathcal{K} = (\mathcal{K}^{\text{must}}, \mathcal{K}^{\text{may}})$*, where* $\mathcal{K}^{\text{must}} = (\Sigma_K, \mathtt{Act}, \to_{\text{must}})$ *and* $\mathcal{K}^{\text{may}} = (\Sigma_K, \mathtt{Act}, \to_{\text{may}})$ *are LTSs such that* $\to_{\text{must}} \subseteq \to_{\text{may}}$.

An LTS is simply a MTS $\mathcal{K}$ where $\mathcal{K}^{\text{must}}$ equals $\mathcal{K}^{\text{may}}$. The intuition behind the inclusion above is that transitions that are necessarily true ($\mathcal{K}^{\text{must}}$) are also possibly true ($\mathcal{K}^{\text{may}}$). Reasoning about the existence of transitions of MTSs can be viewed as reasoning with a three-valued logic with truth values `true`, `false`, and `unknown` [4]: transitions that are necessarily true are `true`, transitions that are possibly true but not necessarily true are `unknown`, and transitions that are not possibly true are `false`.

Definition 4 (Refinement [24]). *An MTS* $\mathcal{A}_1$ *is a* refinement *of an MTS* $\mathcal{A}_2$ *if there exists a relation* $\rho \subseteq \Sigma_{A_1} \times \Sigma_{A_2}$ *such that (i)* ρ *is a simulation from* $\mathcal{A}_1^{\text{may}}$ *to* $\mathcal{A}_2^{\text{may}}$ *and (ii)* ρ *is a simulation from* $\mathcal{A}_2^{\text{must}}$ *to* $\mathcal{A}_1^{\text{must}}$*. In that case, we also say that* $\mathcal{A}_2$ *is an* abstraction *of* $\mathcal{A}_1$*. We write* $\prec$ *for the greatest refinement relation between MTSs.*

MTSs can be used to both verify and refute *any* property of the full modal mu-calculus, which is defined as follows [22]:

$$\phi ::= \mathtt{tt} \mid Z \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid (\exists\alpha)\phi \mid \mu Z.\phi \tag{2}$$

where $\alpha \in \mathtt{Act}$, and $Z \in \mathtt{Var}$ (variable for the least fixed point $\mu Z.\phi$).

Definition 5 (Semantics of modal logic [19]). *For a MTS* $\mathcal{K}$ *and any modal mu-calculus formula* ϕ*, we define a semantics* $[\![\phi]\!]_\sigma \in \mathcal{P}(\Sigma_K) \times \mathcal{P}(\Sigma_K)$*, where* $\mathcal{P}(\Sigma_K)$ *is the powerset of* Σ_K*, ordered by set inclusion,* $\sigma\colon \mathtt{Var} \to \mathcal{P}(\Sigma_K) \times \mathcal{P}(\Sigma_K)$ *is an environment, and* $[\![\phi]\!]_\sigma^{\text{nec}}$ *and* $[\![\phi]\!]_\sigma^{\text{pos}}$ *are the projection of* $[\![\phi]\!]_\sigma$ *to its first and second component, respectively:*

1. $[\![\mathtt{tt}]\!]_\sigma \stackrel{\text{def}}{=} \langle \Sigma_K,\ \Sigma_K \rangle$;

2. $\llbracket \neg\phi \rrbracket_\sigma \stackrel{\text{def}}{=} \langle \Sigma_K \setminus \llbracket \phi \rrbracket_\sigma^{\text{pos}},\ \Sigma_K \setminus \llbracket \phi \rrbracket_\sigma^{\text{nec}} \rangle$;
3. $\llbracket \phi_1 \wedge \phi_2 \rrbracket_\sigma \stackrel{\text{def}}{=} \langle \llbracket \phi_1 \rrbracket_\sigma^{\text{nec}} \cap \llbracket \phi_2 \rrbracket_\sigma^{\text{nec}},\ \llbracket \phi_1 \rrbracket_\sigma^{\text{pos}} \cap \llbracket \phi_2 \rrbracket_\sigma^{\text{pos}} \rangle$;
4. $\llbracket (\exists\alpha)\phi \rrbracket_\sigma \stackrel{\text{def}}{=} \langle \{s \in \Sigma_K \mid \textit{for some } s',\ s \rightarrow^a_{\text{must}} s' \textit{ and } s' \in \llbracket \phi \rrbracket_\sigma^{\text{nec}}\},$
$\{s \in \Sigma_K \mid \textit{for some } s',\ s \rightarrow^a_{\text{may}} s' \textit{ and } s' \in \llbracket \phi \rrbracket_\sigma^{\text{pos}}\} \rangle$.

The treatment of negation is due to P. Kelb [20] and allows for verifying ($s \in \llbracket \phi \rrbracket^{\text{nec}}$) and refuting ($s \in \llbracket \neg\phi \rrbracket^{\text{nec}}$) property ϕ at state s. For brevity, we did not present the standard least-fixed point semantics of $\mu Z.\phi$ (e.g., see [19]).

Theorem 1 (Soundness and consistency of semantics [19]). *For any MTSs, formulas ϕ, ψ of the modal mu-calculus, and environments σ:*

1. $\llbracket \phi \rrbracket_\sigma^{\text{nec}} \subseteq \llbracket \phi \rrbracket_\sigma^{\text{pos}}$;
2. $\llbracket \phi \wedge \neg\phi \rrbracket_\sigma^{\text{nec}} = \emptyset$; *and* $\llbracket \phi \vee \neg\phi \rrbracket_\sigma^{\text{pos}} = \Sigma_K$; *that is, the semantics is consistent for* $\llbracket\ \rrbracket^{\text{nec}}$ *and "complete" for* $\llbracket\ \rrbracket^{\text{pos}}$;
3. *if* $c \prec a$, *then* $a \in \llbracket \phi \rrbracket_\sigma^{\text{nec}}$ *implies* $c \in \llbracket \phi \rrbracket_\sigma^{\text{nec}}$; *and* $c \in \llbracket \phi \rrbracket_\sigma^{\text{pos}}$ *implies* $a \in \llbracket \phi \rrbracket_\sigma^{\text{pos}}$; *that is, verification and refutation of* ϕ *are sound;*
4. *For LTSs,* $\llbracket \phi \rrbracket_\sigma^{\text{nec}} = \llbracket \phi \rrbracket_\sigma^{\text{pos}}$ *and corresponds to the standard semantics for labeled transition systems.*

The semantics $\llbracket \phi \rrbracket^{\text{nec}}$ (without negation and fixed points) is the one given by Larsen [23]; it produces a logical characterization of refinement for finitely branching[4] MTSs [23]. Since $s \notin \llbracket \phi \rrbracket^{\text{pos}}$ iff $s \in \llbracket \neg\phi \rrbracket^{\text{nec}}$, this logical characterization can be extended to the full mu-calculus (including negation). We thus obtain that $c \prec a$ iff for all ϕ of the modal mu-calculus, $[a \in \llbracket \phi \rrbracket^{\text{nec}} \Rightarrow c \in \llbracket \phi \rrbracket^{\text{nec}}]$.

3 A Relational Calculus for Abstract MTSs

In [12], abstract interpretation frameworks are systematically defined through description relations $\rho\colon \Sigma_C \times \Sigma_A$ with suitable properties. We provide a general calculus for specifying abstract MTSs based on such relations.

Definition 6 (Relational abstraction). *Let* $\mathcal{A}_1 = (\mathcal{A}_1^{\text{must}}, \mathcal{A}_1^{\text{may}})$ *be an MTS. Given a set* Σ_{A_2} *of abstract states and a total relation*[5] $\rho\colon \Sigma_{A_1} \times \Sigma_{A_2}$, *we define* $\mathcal{A}_2 = (\Sigma_{A_2}, \mathtt{Act}, \xrightarrow{\text{must}}, \xrightarrow{\text{may}})$ *as follows:*

- $a_2 \rightarrow^\alpha_{\text{must}} a_2'$ *iff for all* $a_1 \in \Sigma_{A_1}$ *with* $a_1\, \rho\, a_2$ *there exists* $a_1' \in \Sigma_{A_1}$ *such that* $a_1'\, \rho\, a_2'$ *and* $a_1 \rightarrow^\alpha_{\text{must}} a_1'$;
- $a_2 \rightarrow^\alpha_{\text{may}} a_2'$ *iff there exist* $a_1 \in \Sigma_{A_1}$ *and* $a_1' \in \Sigma_{A_1}$ *such that* $a_1\, \rho\, a_2$, $a_1'\, \rho\, a_2'$, *and* $a_1 \rightarrow^\alpha_{\text{may}} a_1'$.

This definition is a tool to specify abstract MTSs. Its two components are similar to the universal abstraction $\alpha^{\forall\exists}$ and the (dual) existential abstraction $\alpha^{\exists\exists}$ defined in [11], and to the relations $R^{\forall\exists}$ and $R^{\exists\exists}$ in [12].

[4] `may`-components and `must`-components are finitely-branching.
[5] That is, $(\forall a_1 \in \Sigma_{A_1} \exists a_2 \in \Sigma_{A_2}\colon a_1\, \rho\, a_2) \wedge (\forall a_2 \in \Sigma_{A_2} \exists a_1 \in \Sigma_{A_1}\colon a_1\, \rho\, a_2)$.

Lemma 1. *Given $\mathcal{A}_1$ and $\mathcal{A}_2$ as above, $\mathcal{A}_2$ is a MTS and ρ is a refinement.*

Totality is a natural condition in applications and Definition 6 can express stepwise abstractions, products, sums, etc; moreover, it translates to other frameworks — e.g. the one based on Galois connections [10] — in the manner described in [12].

The specification in Definition 6 is also complete: given an MTS $\mathcal{A}_1$, any abstraction $\mathcal{A}_2$ of $\mathcal{A}_1$ via a total refinement relation $\prec$ can be constructed using Definition 6 by choosing ρ as $\prec$. The following example illustrates Definition 6.

Example 1. Let $\mathcal{A}_1$ be a complete MTS ($\mathcal{A}_1^{\text{may}}$ equals $\mathcal{A}_1^{\text{must}}$) whose infinite state space is given by all possible valuations of three integer variables `x`, `y`, and `z`. Any state c is of the form $\{\mathtt{x} \mapsto i, \mathtt{y} \mapsto j, \mathtt{z} \mapsto k\}$, for some integers i, j, k. Let us assume that transitions of $\mathcal{A}_1$ are those induced by the single assignment statement `x = z`, e.g. there is a transition from state c above to state $c' = \{\mathtt{x} \mapsto k, \mathtt{y} \mapsto j, \mathtt{z} \mapsto k\}$.

The predicates $\phi_1 \stackrel{\text{def}}{=} \mathtt{odd(x)}$, $\phi_2 \stackrel{\text{def}}{=} (\mathtt{y} > 0)$, and $\phi_3 \stackrel{\text{def}}{=} (\mathtt{z} < 0)$ induce an equivalence relation on the states of $\mathcal{A}_1$: two states are equivalent if they agree on all three predicates. Let Σ_{A_2} be the set of all sets of equivalence classes of states of $\mathcal{A}_1$. Therefore, states of $\mathcal{A}_2$ are representable as boolean formulas built from the ϕ_i's. By Definition 6, there is a `may`-transition from a to a' in Σ_{A_2} iff there are $c \in a$ and $c' \in a'$ such that c has a transition to c' in $\mathcal{A}_1$. Dually, there is a `must`-transition from a to a' iff, *for all* $c \in a$, there exists $c' \in a'$ such that c has a transition to c' in $\mathcal{A}_1$.

For instance, there is (i) a `may`-transition from the state $\phi_1 \wedge \phi_2 \wedge \phi_3$ to the states $\phi_1 \wedge \phi_2 \wedge \phi_3$ and $\neg\phi_1 \wedge \phi_2 \wedge \phi_3$; (ii) a `must`-transition from $\phi_1 \wedge \phi_2 \wedge \phi_3$ to the disjunction of monomials $(\phi_1 \wedge \phi_2 \wedge \phi_3) \vee (\neg\phi_1 \wedge \phi_2 \wedge \phi_3)$; but (iii) *no* `must`-transition from $\phi_1 \wedge \phi_2 \wedge \phi_3$ to any monomial, e.g. $\phi_1 \wedge \phi_2 \wedge \phi_3$ or $\neg\phi_1 \wedge \phi_2 \wedge \phi_3$.

4 Implementation of Relationally Specified MTSs

In this section, we consider in turn predicate abstraction (also called "boolean abstraction") and cartesian abstraction. When applied to `may`-transition relations only, the specifications and implementations we present coincide with traditional "conservative" abstractions. We implement these specifications with standard tools (automatic theorem-proving for quantifier-free first-order logic and BDDs), except for the use of TDDs. We show that the computational cost of constructing a `must`-transition relation is the same as that of constructing a `may`-transition relation. We then show that our implementations are sound and complete (relatively to the completeness of the underlying theorem prover) with respect to their specifications. Moreover, they conveniently model approximations in calls to a theorem prover as under-approximations of `must`-transitions and over-approximations of `may`-transitions. Importantly, we prove that abstraction refinement is incremental for MTSs built using cartesian abstraction.

Notation. For any predicate η on a set Σ_S of states, for any label $\alpha \in \texttt{Act}$, the post operator [10] and weakest precondition [15] are defined as

$$\begin{aligned}
\texttt{post}_\alpha(\eta) &\stackrel{\text{def}}{=} \{s' \in \Sigma_S \mid \exists s \in \Sigma_S\colon s \models \eta,\ s \to^\alpha s'\} \\
\tilde{\texttt{pre}}_\alpha(\eta) &\stackrel{\text{def}}{=} \{s \in \Sigma_S \mid \forall s' \in \Sigma_S\colon s \to^\alpha s' \text{ implies } s' \models \eta\}.
\end{aligned}$$

These operators satisfy several interesting relationships [10]. Here we only use the property that, for any predicates η, ψ on states, $\texttt{post}_\alpha(\psi) \wedge \eta$ is satisfiable if and only if $\psi \wedge \neg\tilde{\texttt{pre}}_\alpha(\neg\eta)$ is satisfiable.

Methodological assumptions. We assume that an abstract program is built by converting each program statement from a transformer operating on concrete states to a transformer operating on abstract states, as illustrated in Example 1. For notational convenience, we focus in what follows on the abstraction of a single program statement, and hence consider MTSs, `post`, and $\tilde{\texttt{pre}}$ without explicit action labels. For a given program statement and a quantifier-free formula η, we assume that $\tilde{\texttt{pre}}(\eta)$ is *quantifier-free* as well. This is the case for usual programming language constructs [17] and enables the use of decision procedures as implemented in tools such as SVC [14].

Predicate Abstraction. Predicate abstraction [17, 14, 32] collapses an infinite-state LTS into a finite-state MTS defined by choosing finitely many quantifier-free formulas of first-order logic.

Specification. The abstract states in the predicate abstraction are built out of monomials over predicates. Each abstract state corresponds to a set of concrete states that satisfy the same set of predicates. Formally, given a finite set of quantifier-free formulas of first-order logic, $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, and a "bit-vector" $b \in \{0,1\}^n$, we write $\langle b, \Phi\rangle$ for a monomial whose ith conjunct is ϕ_i if $b_i = 1$, and $\neg\phi_i$ otherwise.

Definition 7 (Predicate abstraction). *Given an LTS $\mathcal{S}$ and a finite set $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ of quantifier-free formulas of first-order logic, we derive a finite-state abstract MTS $\mathcal{B}_\Phi$ following Definition 6 ($\mathcal{A}_1$ is $\mathcal{S}$ and $\mathcal{A}_2$ is $\mathcal{B}_\Phi$) in such a way that:*

- $\rho \stackrel{\text{def}}{=} \rho^\Phi_{\text{b}} \subseteq \Sigma_S \times \{0,1\}^n$, *where* $s\,\rho^\Phi_{\text{b}}\,b$ *iff* $s \models \langle b, \Phi\rangle$*; and*
- $\Sigma_{\mathcal{B}_\Phi} \stackrel{\text{def}}{=} \{b \in \{0,1\}^n \mid s\,\rho^\Phi_{\text{b}}\,b$ *for some* $s \in \Sigma_S\}$, *which makes* ρ^Φ_{b} *total.*

Implementing `may`*-successors of a predicate abstraction.* Current tool-supported predicate-abstraction frameworks [17, 14, 32, 9, 2, 1] can be viewed as constructing an abstraction of the `may`-component of $\mathcal{B}_\Phi$ defined above. We now review how to compute the set of abstract `may`-successors for a single program statement.

Following [14], we use BDDs over boolean variables $x_1, x_2, \ldots, x_n$ as representations of such sets. If ψ is a boolean combination of $\{\phi_i \mid m \leq i \leq n\}$, we compute in (3) below a BDD, denoted by $H^{\text{may}}(\psi, \text{true}, m)$, for representing the set of may-successors of ψ. The definition of H^{may} is essentially the definition of H in [14] where $\texttt{post}(\psi) \wedge \eta$ is replaced by $\psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta)$ (to facilitate dualizing this construction later).

$$H^{\text{may}}(\psi,\eta,i) \stackrel{\text{def}}{=} \begin{cases} (x_i \wedge H^{\text{may}}(\psi,\eta \wedge \phi_i, i+1)) & \\ \vee(\neg x_i \wedge H^{\text{may}}(\psi,\eta \wedge \neg\phi_i, i+1)) & \text{if } 0 < i \leq n, \\ 1 \quad \text{if } i = n+1 \text{ and } \psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta) \text{ is satisfiable}, & \\ 0 \quad \text{if } i = n+1 \text{ and } \psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta) \text{ is unsatisfiable}. & \end{cases} \tag{3}$$

The BDD in (3) can be computed using standard BDD operations [6] plus the optimizations discussed in [14], while the satisfiability checks can be computed by calling a theorem prover. Unwinding the recursion in the definition above, it is clear that the set of may-successors of ψ computed by H^{may} is:

$$\texttt{next}(\psi)^{\text{may}}_{\text{b}} \stackrel{\text{def}}{=} \{b' \in \Sigma_B \mid \psi \wedge \neg\tilde{\texttt{pre}}(\neg\langle b', \Phi\rangle) \text{ is satisfiable }\}. \tag{4}$$

Implementing must*-successors of a predicate abstraction.* The logical duality of may versus must is captured by replacing the satisfiability check of $\psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta)$ in (3) by the unsatisfiability check of $\psi \wedge \tilde{\texttt{pre}}(\neg\eta)$ in the following equation (5).

$$H^{\text{must}}(\psi,\eta,i) \stackrel{\text{def}}{=} \begin{cases} (x_i \wedge H^{\text{must}}(\psi,\eta \wedge \phi_i, i+1)) & \\ \vee(\neg x_i \wedge H^{\text{must}}(\psi,\eta \wedge \neg\phi_i, i+1)) & \text{if } 0 < i \leq n, \\ 1 \quad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\texttt{pre}}(\neg\eta) \text{ is unsatisfiable}, & \\ 0 \quad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\texttt{pre}}(\neg\eta) \text{ is satisfiable}. & \end{cases} \tag{5}$$

Thus, the set of must-successors of ψ represented by the BDD $H^{\text{must}}(\psi, \text{true}, m)$ is:

$$\texttt{next}(\psi)^{\text{must}}_{\text{b}} \stackrel{\text{def}}{=} \{b' \in \Sigma_B \mid \psi \wedge \tilde{\texttt{pre}}(\neg\langle b', \Phi\rangle) \text{ is unsatisfiable }\}. \tag{6}$$

We now show that the BDDs computed by H^{may} and H^{must} represent exactly the transitions specified in Definition 7.

Theorem 2 (Soundness and completeness).

- $b \rightarrow_{\text{may}} b'$ *in* $\mathcal{B}_\Phi$ *iff* $b' \in \texttt{next}(\langle b, \Phi\rangle)^{\text{may}}_{\text{b}}$*;*
- $b \rightarrow_{\text{must}} b'$ *in* $\mathcal{B}_\Phi$ *iff* $b' \in \texttt{next}(\langle b, \Phi\rangle)^{\text{must}}_{\text{b}}$*.*

Proof. The proof follows from the direct application of the definitions and is omitted here due to space constraints.

Cost. In the worst case, the computation of $H^{\text{may}}(\langle b, \Phi\rangle, \text{true}, m)$ makes $O(2^n)$ calls to the theorem prover. Similarly, the computation of $H^{\text{must}}(\langle b, \Phi\rangle, \text{true}, m)$ makes at most $O(2^n)$ calls to the theorem prover. Hence, *the complexity of computing* H^{must} *is the same as the complexity of computing* H^{may}.

Note that optimizations for computing H^{may} discussed in [14] can also be used when computing H^{must}. Our algorithm also accommodates the complexity of theorem-proving by allowing the sound over-approximation of H^{may} and under-approximation of H^{must} as follows: in both cases, simply convert the absence of an answer, when truncating the computation performed by the satisfiability checker, into "satisfiable".

Predicate-Cartesian Abstraction. Unfortunately, predicate abstraction of MTSs is not incremental: adding a new predicate ϕ_{n+1} to Φ may not yield a refinement of the abstraction, and hence the entire abstraction may need to be recomputed. This is illustrated by the following example.

Example 2. Revisiting Example 1, if $\Phi = \{\phi_2, \phi_3\}$, then $\mathcal{B}_\Phi$ has four states, each with a `must`-transition to itself. However, adding the predicate ϕ_1 to Φ, there are no `must`-transitions from the abstract state $\phi_1 \wedge \phi_2 \wedge \phi_3$ (111) in $\mathcal{B}_{\{\phi_1\}\cup\Phi}$. This is quite unfortunate: the information about variable `y` is lost even though `y` is absent from the assignment `x = z`. But in Example 1 we saw that there *is* a `must`-transition from $\phi_1 \wedge \phi_2 \wedge \phi_3$ to the *disjunction* $(\phi_1 \wedge \phi_2 \wedge \phi_3) \vee (\neg\phi_1 \wedge \phi_2 \wedge \phi_3)$ that correctly captures the "absence of effect" on `y`.

Computing `must`-transitions with abstract states of the above kind can be expensive: given n predicates, there are a possible 2^{2^n} such states. This motivates our next topic: cartesian abstraction.

Specification. The basic idea behind cartesian abstraction is to approximate sets of tuples by a tuple of sets. For instance, a set $\{\langle 0, 1\rangle, \langle 1, 1\rangle\}$ is represented by $\{\langle \star, 1\rangle\}$, where $\star$ is used as a wildcard for different values (such as 0 and 1 in this example). Formally, given a finite set $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ of quantifier-free formulas of first-order logic and a "tri-vector" $c \in \{0, 1, \star\}^n$, we write $\langle c, \Phi\rangle$ for a monomial whose ith conjunct is ϕ_i if $c_i = 1$, $\neg\phi_i$ if $c_i = 0$, and `true` otherwise. Abstract states in $\mathcal{C}_\Phi$ are built out of "tri-vectors" of length n.

Definition 8 (Predicate-cartesian abstraction). *Given an LTS $\mathcal{S}$ and a finite set $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ of quantifier-free formulas of first-order logic, we derive a finite-state abstract MTS $\mathcal{C}_\Phi$ following Definition 6 ($\mathcal{A}_1$ is $\mathcal{S}$ and $\mathcal{A}_2$ is $\mathcal{C}_\Phi$) in such a way that:*

- $\rho \stackrel{\text{def}}{=} \rho_{\text{c}}^{\Phi} \circ \rho_{\text{b}}^{\Phi} \subseteq \Sigma_S \times \{0, 1, \star\}^n$, *where* $b\, \rho_{\text{c}}^{\Phi}\, c$ *iff* $\forall 1 \leq i \leq n : [c_i \neq \star \;\; \Rightarrow \;\; b_i = c_i]$; *and*
- $\Sigma_{\mathcal{C}_\Phi} \stackrel{\text{def}}{=} \{c \in \{0, 1, \star\}^n \mid b\, \rho_{\text{c}}^{\Phi}\, c$ *for some* $b \in \Sigma_{\mathcal{B}_\Phi}\}$, *which makes* ρ_{c}^{Φ} *total.*

The symbol $\star$ means "don't care" in the above definition. It is easy to show that, by construction, we have:

$$s\,(\rho_{\text{c}}^{\Phi} \circ \rho_{\text{b}}^{\Phi})\,c \qquad \text{iff} \qquad s \models \langle c, \Phi\rangle. \tag{7}$$

Note that the abstract MTS $\mathcal{C}'_\Phi$ obtained by abstracting $\mathcal{B}_\Phi$ (whose states are vectors of n-bits) with ρ_{c}^{Φ} is typically less precise than $\mathcal{C}_\Phi$. For instance, in the case of Examples 1 and 2 again, $\mathcal{C}'_\Phi$ would not contain any `must`-transition from state 111 (i.e., $\phi_1 \wedge \phi_2 \wedge \phi_3$), while $\mathcal{C}_\Phi$ does contain a `must`-transition from 111 to state $\star$11.

The MTS $\mathcal{C}_\Phi$ supports an approximate union operation, defined using the point-wise application of Kleene's alignment operator [21]: $c \cup c' \stackrel{\text{def}}{=} c''$, where $c''_i = c_i$ if $c_i = c'_i$ and $\star$ otherwise. This operation thus approximates disjunctions (sets) of monomials by tri-vectors. As previously mentioned, cartesian abstraction allows for incremental abstraction refinement:

Theorem 3 (Incremental refinement). *For $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ and $\Psi = \Phi \cup \{\phi_{n+1}, \phi_{n+2}, \ldots, \phi_{n+m}\}$, the MTS $\mathcal{C}_\Psi$ is a refinement of the MTS $\mathcal{C}_\Phi$.*

Proof. The refinement $\rho \subseteq \Sigma_{C_\Psi} \times \Sigma_{C_\Phi}$ is given by $\{(c', c) \mid c \text{ is a prefix of } c'\}$.

Implementing `may`*-successors of a predicate-cartesian abstraction.* Instead of representing the abstract post-operator with a BDD as in the predicate abstraction case, we now use a Ternary Decision Diagram [31], writing $[x/v]$ for the replacement of variable x with value $v \in \{0, 1, \star\}$:

$$G^{\text{may}}(\psi, \eta, i) \stackrel{\text{def}}{=} \begin{cases} ([x_i/1] \wedge G^{\text{may}}(\psi, \eta \wedge \phi_i, i+1)) \vee ([x_i/0] \wedge G^{\text{may}}(\psi, \eta \wedge \neg\phi_i, i+1)) \\ \vee ([x_i/\star] \wedge G^{\text{may}}(\psi, \eta, i+1)) \qquad \text{if } 0 < i \leq n, \\ 1 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \neg\tilde{\mathtt{pre}}(\neg\eta) \text{ is satisfiable,} \\ 0 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \neg\tilde{\mathtt{pre}}(\neg\eta) \text{ is unsatisfiable.} \end{cases} \tag{8}$$

The function G^{may} essentially computes the abstract post-operator of SLAM [1]. Unwinding the recursion in the above definition, the set of `may`-successors of ψ represented by the TDD $G^{\text{may}}(\psi, \text{true}, m)$ is:

$$\mathtt{next}(\psi)^{\text{may}}_{\text{c}} \stackrel{\text{def}}{=} \{c' \in \Sigma_C \mid \psi \wedge \neg\tilde{\mathtt{pre}}(\neg\langle c', \Phi\rangle) \text{ is satisfiable }\}. \tag{9}$$

Implementing `must`*-successors of a predicate-cartesian abstraction.* The TDD $G^{\text{must}}(\psi, \text{true}, m)$, defined below, represents the set of `must`-successors of ψ. Similar to our presentation of predicate abstraction, we are capturing the logical duality of `may` versus `must` by replacing the satisfiability check of $\psi \wedge \neg\tilde{\mathtt{pre}}(\neg\eta)$ by the unsatisfiability check of $\psi \wedge \tilde{\mathtt{pre}}(\neg\eta)$ in the following equation:

$$G^{\text{must}}(\psi, \eta, i) \stackrel{\text{def}}{=} \begin{cases} ([x_i/1] \wedge G^{\text{must}}(\psi, \eta \wedge \phi_i, i+1)) \vee ([x_i/0] \wedge G^{\text{must}}(\psi, \eta \wedge \neg\phi_i, i+1)) \\ \vee ([x_i/\star] \wedge G^{\text{must}}(\psi, \eta, i+1)) \qquad \text{if } 0 < i \leq n, \\ 1 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\mathtt{pre}}(\neg\eta) \text{ is unsatisfiable,} \\ 0 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\mathtt{pre}}(\neg\eta) \text{ is satisfiable.} \end{cases} \tag{10}$$

Again, by unwinding the recursion, the set of `must`-successors of ψ represented by the TDD $G^{\text{must}}(\psi, \text{true}, m)$ is thus defined by:

$$\mathtt{next}(\psi)^{\text{must}}_{\text{c}} \stackrel{\text{def}}{=} \{c' \in \Sigma_C \mid \psi \wedge \tilde{\mathtt{pre}}(\neg\langle c', \Phi\rangle) \text{ is unsatisfiable }\}. \tag{11}$$

The following theorem states that the TDDs computed by G^{may} and G^{must} represent exactly the transitions specified in Definition 8.

Theorem 4 (Soundness and completeness).

- *$c \rightarrow_{\text{may}} c'$ in $\mathcal{C}_\Phi$ iff $c' \in \mathtt{next}(\langle c, \Phi\rangle)^{\text{may}}_{\text{c}}$;*
- *$c \rightarrow_{\text{must}} c'$ in $\mathcal{C}_\Phi$ iff $c' \in \mathtt{next}(\langle c, \Phi\rangle)^{\text{must}}_{\text{c}}$.*

Proof. Similar to the proof of Theorem 2.

Cost. In the worst case, the computation of $G^{\mathrm{may}}(\langle c, \Phi\rangle, \mathrm{true}, m)$ makes $O(3^n)$ calls to the theorem prover. Similarly, the computation of $G^{\mathrm{must}}(\langle c, \Phi\rangle, \mathrm{true}, m)$ makes at most $O(3^n)$ calls to the theorem prover. Therefore, *the complexity of computing G^{must} is the same as the complexity of computing G^{may}.*

Note that the heuristics discussed in [1] for approximating the calculation of $G^{\mathrm{may}}(\langle c, \Phi\rangle, \mathrm{true}, m)$ by restricting the expansion of the recursion to a fixed depth (rather than n) can be applied when computing $G^{\mathrm{must}}(\langle c, \Phi\rangle, \mathrm{true}, m)$ as well. Again, the absence of answers from the theorem prover for satisfiability checks can be interpreted as "satisfiable" to yield a sound over-approximation of G^{may} and under-approximation of G^{must}.

5 Three-valued Model Checking on MTSs

The automatic-abstraction algorithms of the previous section can be used to generate a MTS $\mathcal{A}_2$ which, by construction, is guaranteed to be an abstraction (as defined in Definition 4) of a given, possibly initial and concrete, system $\mathcal{A}_1$. By Theorem 1, we can check a modal mu-calculus formula ϕ on $\mathcal{A}_1$ by analyzing $\mathcal{A}_2$ instead, resulting in three possible answers: either (i) ϕ is necessarily true on $\mathcal{A}_2$ — its initial state is contained in $[\![\ \phi\ ]\!]^{\mathrm{nec}}$ — and hence ϕ holds for $\mathcal{A}_1$ (the answer is `true`), or (ii) ϕ is only possibly true on $\mathcal{A}_2$ — its initial state is contained in $[\![\ \phi\ ]\!]^{\mathrm{pos}}$ only — and whether ϕ holds on $\mathcal{A}_1$ is unknown (the answer is `unknown`), or (iii) ϕ is not possibly true on $\mathcal{A}_2$ — its initial state is not contained in $[\![\ \phi\ ]\!]^{\mathrm{pos}}$ — and ϕ does not hold on $\mathcal{A}_1$ (the answer is `false`). We are thus left with a three-valued model-checking problem on MTSs which, following [5], can be reduced to two model-checking problems on LTSs as follows.

First, we rewrite formula ϕ to a formula ϕ^+ in positive normal form defined over all the clauses of (1) plus (2) by pushing all negations in ϕ inwards so that they apply only to `tt` or `ff` in ϕ^+. This is done using the classic rewrite rules: $\neg\neg\phi = \phi$, $\neg(\phi_1 \wedge \phi_2) = (\neg\phi_1) \vee (\neg\phi_2)$, $\neg((\exists\alpha)\phi) = (\forall\alpha)(\neg\phi)$, and $\neg(\mu Z.\phi) = \nu Z.(\neg\phi)$. Then, we translate ϕ^+ into a formula $T(\phi)$ by applying the following translation rules: for all $\alpha \in \mathtt{Act}$, replace all occurrences of $(\forall\alpha)$ in ϕ^+ by $(\forall\alpha_\forall)$ and replace all occurrences of $(\exists\alpha)$ in ϕ^+ by $(\exists\alpha_\exists)$.

Second, from the MTS $\mathcal{A}_2 = (\Sigma_{A_2}, \mathtt{Act}, \rightarrow_{\mathrm{must}}, \rightarrow_{\mathrm{may}})$, we define two LTSs $\mathcal{A}_2^{\mathrm{pess}}$ and $\mathcal{A}_2^{\mathrm{opt}}$, representing respectively the *pessimistic* and *optimistic* interpretations of $\mathcal{A}_2$ (see [5]). These two LTSs are defined over the set

$$\mathtt{Act}^{\mathrm{c}} \stackrel{\mathrm{def}}{=} \{\alpha_\forall \mid \alpha \in \mathtt{Act}\} \cup \{\alpha_\exists \mid \alpha \in \mathtt{Act}\} \tag{12}$$

of action symbols. Precisely, we define $\mathcal{A}_2^{\mathrm{pess}} = (\Sigma_{A_2}, \mathtt{Act}^{\mathrm{c}}, \rightarrow_{\mathrm{pess}})$ with

$$(s, \alpha_\forall, s') \in\ \rightarrow_{\mathrm{pess}} \text{ if } (s, \alpha, s') \in\ \rightarrow_{\mathrm{may}} \tag{13}$$

$$(s, \alpha_\exists, s') \in\ \rightarrow_{\mathrm{pess}} \text{ if } (s, \alpha, s') \in\ \rightarrow_{\mathrm{must}} \tag{14}$$

and we define $\mathcal{A}_2^{\mathrm{opt}} = (\Sigma_{A_2}, \mathtt{Act}^{\mathrm{c}}, \rightarrow_{\mathrm{opt}})$ with

$$(s, \alpha_\forall, s') \in\ \rightarrow_{\mathrm{opt}} \text{ if } (s, \alpha, s') \in\ \rightarrow_{\mathrm{must}} \tag{15}$$

$$(s, \alpha_\exists, s') \in\ \rightarrow_{\mathrm{opt}} \text{ if } (s, \alpha, s') \in\ \rightarrow_{\mathrm{may}}\ . \tag{16}$$

Finally, we model-check the modal mu-calculus formula $T(\phi)$ on the LTSs $\mathcal{A}_2^{\mathrm{pess}}$ and $\mathcal{A}_2^{\mathrm{opt}}$, and combine the results as specified in the following theorem.

Theorem 5 (Correctness of model checking algorithm). *Given a MTS $\mathcal{A}_2$ and a modal mu-calculus formula ϕ, let $T(\phi)$, $\mathcal{A}_2^{\mathrm{pess}}$, and $\mathcal{A}_2^{\mathrm{opt}}$ be the formula and the two LTSs (respectively) as defined above. For any state $s \in \Sigma_{A_2}$, we then have*

1. $s \in [\![\phi]\!]^{\mathrm{nec}}$ *iff* $(\mathcal{A}_2^{\mathrm{pess}}, s) \models T(\phi)$
2. $s \in [\![\phi]\!]^{\mathrm{pos}}$ *iff* $(\mathcal{A}_2^{\mathrm{opt}}, s) \models T(\phi)$.

Proof. By induction on the structure of ϕ.

The previous theorem is similar to Theorem 3 of [5]. It reduces three-valued model checking of MTSs to two traditional (two-valued) model-checking problems on regular LTSs, namely $(\mathcal{A}_2^{\mathrm{pess}}, s) \models T(\phi)$ and $(\mathcal{A}_2^{\mathrm{opt}}, s) \models T(\phi)$. Since the transformations performed to obtain $T(\phi)$, $(\mathcal{A}_2^{\mathrm{pess}}, s)$, and $(\mathcal{A}_2^{\mathrm{opt}}, s)$ can be done in constant space and time linear in the size of the formula and MTS respectively, three-valued model checking on MTSs has the same time and space complexity as two-valued model checking on LTSs. Moreover, the problem can be solved in practice using existing model-checking tools for LTSs, with all the optimizations that these tools may already implement. In particular, if the refined system $\mathcal{A}_1$ is concrete and composed of multiple concurrent LTSs or of recursive procedures (LTSs extended with a "call-stack"), the abstraction algorithms of the previous section will preserve the architecture of $\mathcal{A}_1$ when generating $\mathcal{A}_2$, and existing tools for model-checking concurrent or pushdown systems can be applied to $\mathcal{A}_2^{\mathrm{pess}}$ and $\mathcal{A}_2^{\mathrm{opt}}$.

6 Conclusions

We developed a framework for automatic program abstraction based on modal transition systems. This framework can be used for model-checking any formula of the modal mu-calculus, and is also applicable to the verification of concurrent and pushdown systems. It uses cartesian abstraction, implemented with TDDs and quantifier-free first-order-logic theorem-proving, to extend existing predicate-abstraction techniques to the verification of formulas containing arbitrarily nested path quantifiers. Cartesian abstraction has no significant cost overhead and is compatible with the standard incremental refinement process for adding more predicates.

Acknowledgments

We wish to thank Glenn Bruns and David Schmidt for inspiring discussions and helpful comments.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'2001*, volume 2031 of *LNCS*, pages 268–283, Genova, Italy, April 2001. Springer Verlag.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the Seventh International SPIN Workshop (SPIN 2000)*, volume 1885, pages 113–130. Springer Verlag, 2000.
3. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions on infinite state systems compositionally and automatically. In A. J. Hu and M. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427, pages 319–331, Vancouver, Canada, 1998. Springer Verlag.
4. G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *Proceedings of the 11th Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287. Springer Verlag, July 1999.
5. G. Bruns and P. Godefroid. Generalized Model Checking: Reasoning about Partial State Spaces. In *Proceedings of CONCUR'2000 (11th International Conference on Concurrency Theory)*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer Verlag, August 2000.
6. R. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
7. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
8. R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *SAS'95: Proc. 2d. Static Analysis Symposium*, Lecture Notes in Computer Science 983, pages 51–63. Springer, 1995.
9. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd Intl' Conference on Software Engineering*, June 2000.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
11. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.
12. D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.
13. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
14. S. Das, D. L. Dill, and S. Park. Experience with Predicate Astraction. In N. Halbwachs and D. Peled, editors, *Proc. of the 11th International Conference on Computer-Aided Verification*, pages 160–172, Trento, Italy, July 1999. Springer Verlag.
15. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
16. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

17. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Grumberg O., editor, *Conference on Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.
18. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.
19. M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In D. Sands, editor, *Proceedings of the European Symposium on Programming (ESOP'2001)*, volume 2028 of *LNCS*, pages 155–169, Genova, Italy, April 2001. Springer Verlag.
20. P. Kelb. Model checking and abstraction: a framework preserving both truth and failure information. Technical Report OFFIS, University of Oldenburg, Germany, 1994.
21. S. C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.
22. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
23. K. G. Larsen. Modal Specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 232–246. Springer Verlag, June 12–14 1989. International Workshop, Grenoble, France.
24. K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Third Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988.
25. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design: An International Journal*, 6(1):11–44, January 1995.
26. R. Milner. An algebraic definition of simulation between programs. In *2nd International Joint Conference on Artificial Intelligence*, pages 481–489, London, United Kingdom, 1971. British Computer Society.
27. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
28. D. M. R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *In Proc. of the 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, 1989.
29. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J. W. de Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1985.
30. H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proc. of the 11th Conference on Computer-Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 443–454. Springer, 1999.
31. T. Sasao. Ternary Decision Diagrams — Survey. In *Proceedings of the 27th International Symposium on Multi-valued Logic*, pages 241–250. IEEE, 1997.
32. W. Visser, S. J. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proc. of Formal Methods in Software Practice (FMSP'00)*, pages 3–12, Portland, Oregon, August 2000.