Proof-carrying Bounded Model Checking for All languages

Thesis Proposal

Yi Zhang

University of Illinois at Urbana-Champaign

Abstract. Model checking has been used extensively to algorithmically check temporal property of models. Research on software (bounded) model checking has been focusing on various techniques(e.g., symbolic model checking, predicate abstraction, partial order reduction, etc.) to solve the state-space explosion problem. Various tools such as SLAM, CBMC and CORRAL were proposed. There are two major problems shared among these tools: 1) They only support specific programming languages 2) There is no guarantee that the implementations of the model checkers are correct. To solve the problems, my thesis proposes a proof-carrying bounded model checker for all languages. We can encode both the operational semantics of a language and modal μ-logic as theories and notations in matching μ-logic. As a result, bounded model checking becomes a strategy for searching the proof for a given pattern within k step. The model checker for a language can be obtained for free once the operational semantics for that language is defined. The result of bounded model checking a pattern is an proof object which is checkable by an external proof checker. Moreover, since the bounded model checker is derived from the actual operational semantics, there is no discrepancy between the model checker and the actual language semantics. This proposal document describes the work that I have done so far towards this goal, and my proposed plan for the rest of the thesis work.

1 Introduction

Model checking has been used extensively to algorithmically check temporal property of models. Early application of model checking were in hardware and protocol design. Later the idea was applied to software. Due to the advent of sophisticated SAT and SMT solvers, symbolic model checking becomes a popular approach to cope with infinite program states. Various techniques and tools were proposed and many tools such as SLAM [2] and CORRAL [36] are used in the daily development in companies like Microsoft. Most of the model checking tools were developed for a fixed target language, and the language semantics is usually hardcoded within the implementation of their model checking algorithms. Due to their monolithic design nature, retargeting them to another language requires significant effort. There are two approaches to retarget the model checker to a new language. One is to replace the hardcoded language semantics with the new target language semantics, which may requires substantial amount of work. The other approach is to implement a translator from the new language to the existing language, which has its own limitation if the new language is quite different from the existing one. Moreover, since the language semantics is hardcoded in the model checker, it is very hard to test the hardcoded semantics conforming to the actual behavior of the code, let alone there is no guarantee on the correctness of the tool itself.

The recent work [11] on matching μ-logic shows various logic including modal μ-logic can be defined as theories in matching μ-logic. This suggests that matching μ-logic offers a unifying playground to specify properties and reason about transition systems. Since operational semantics already provides the transition relation between program states, it is natural to build a (bounded) model checker for all languages based on operational semantics in matching μ-logic. This approach admits several advantages. First, the model checker is language-independent and highly reusable. In addition, operational semantics is executable and thus it can be easily tested. Since the model checker is automatically derived from the operational semantics, there is no need to worry about the discrepancies between the model checker and the actual behavior of the code. Second, we formalized the bounded model checking as a strategy to search proof for a given pattern in the matching μ-logic proof system within k step. Each step in the model checking algorithm can be justified by a matching μ-logic proof rule. In the end, the model checker is able to produce a proof object which is checkable by an external proof checker. As a result, there is no need to trust the implementation of the model checker anymore.

The goal of my thesis is to demonstrate and improve the scalability and practicality of such language-independent bounded model checker. Specific work to support this goal falls into these categories:

– Implementing the language-independent model checker and proof generation.
– Instantiating the model checker by plugging-in various real-world programming language semantics and improving its performance.
– Applying the derived model checker to real-world systems and applications, demonstrating their practical feasibility both in isolation and by comparison to state-of-the-art tools specifically crafted for the target languages.

2 Background

This section presents the preliminaries of matching μ-logic and modal μ-logic.

2.1 Matching μ-Logic Preliminaries

The recent work [11] proposed matching μ-logic, an extension of matching logic [45] with a least fixpoint μ-binder. It is shown that matching μ-logic captures many important logics as special instances, including modal μ-logic as well as dynamic logic and various temporal logic. Therefore, matching μ-logic serves as a unifying foundation for specifying and reasoning about fixpoints and induction, programming languages and various program analysis.

Matching μ-Logic syntax

Definition 1. *A* matching μ-logic signature *or simply a* signature $\mathbb{\Sigma} = (S, \textsc{Var}, \Sigma)$ *is a triple. S is a nonempty set of* sorts. $\textsc{Var} = \{\text{EV}\textsc{ar}_s\}_{s\in S} \cup \{\text{SV}\textsc{ar}_s\}_{s\in S}$ *is a disjoint union* *of two S-indexed sets of variables: the* element variables *denoted as $x{:}s$, $y{:}s$, etc. in* EVar*, and the* set variables *denoted as $X{:}s, Y{:}s$, etc in* SVar. $\Sigma = \{\Sigma_{s_1\ldots s_n,s}\}_{s_1,\ldots,s_n,s\in S}$ *is an $(S^*\times S)$-indexed set of countably many* many-sorted symbols. *When $n = 0$, we write $\sigma \in \Sigma_{\lambda,s}$ and say σ is a* constant*. Matching μ-logic $\mathbb{\Sigma}$-*patterns *or simply* ($\mathbb{\Sigma}$-)patterns *are defined inductively for all sorts $s, s', s_1, \ldots, s_n \in S$ as follows:*

$$\begin{aligned}\varphi_s &::= x{:}s \in \mathrm{EVar}_s \mid X{:}s \in \mathrm{SVar}_s \mid \varphi_s \wedge \varphi_s \mid \neg\varphi_s \mid \exists x{:}s'.\varphi_s \\ &\mid \sigma(\varphi_{s_1}, \ldots, \varphi_{s_n}) \quad \text{if } \sigma \in \Sigma_{s_1\ldots s_n,s} \\ &\mid \mu X{:}s.\varphi_s \quad \text{if } \varphi_s \text{ is positive in } X{:}s,\end{aligned}$$

Note that we only quantify over element variables, not set variables. We say φ_s is positive *in $X{:}s$ if every free occurrence of $X{:}s$ is under an even number of negations. We let* Pattern($\mathbb{\Sigma}$) = $\{\text{Pattern}_s\}_{s\in S}$ *denote the set of all matching μ-logic $\mathbb{\Sigma}$-patterns and feel free to drop the signature $\mathbb{\Sigma}$.*

We first give some intuition about the patterns in matching μ-logic and then formalize its semantics in Definition 3. Intuitively, patterns evaluate to the sets of elements that *match* them. An element variable $x{:}s$ is a pattern that is matched by exactly one element; A set variables $X{:}s$ evaluates to *set*. $\varphi_1 \wedge \varphi_2$ is matched by elements matching both φ_1 and φ_2; $\neg\varphi$ is matched by elements not matching φ; $\exists x{:}s'.\varphi$ is a pattern that allows us to abstract away irrelevant parts (i.e., $x{:}s'$) of the structures, which can match patterns $\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})$. The least fixpoint pattern $\mu X{:}s.\,\varphi_s$ gives *the least solution* (under set containment) of the equation $X{:}s = \varphi_s$ of set variable $X{:}s$ (this should be taken as merely intuition at this stage, because we may not have equality in the theories). The condition of positive occurrence guarantees the existence of such a least solution.

We adopt the following derived constructs as syntactic sugar:

$$\begin{aligned}&\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) && \forall x{:}s.\varphi \equiv \neg\exists x{:}s.\neg\varphi \\ &\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2 && \top_s \equiv \exists x{:}s.x{:}s \\ &\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) && \bot_s \equiv \neg\top_s \\ &\nu X{:}s.\varphi_s \equiv \neg\mu X{:}s.\neg\varphi_s[\neg X{:}s/X{:}s] &&\end{aligned}$$

Intuitively, $\varphi_1 \vee \varphi_2$ is matched by elements matching φ_1 or φ_2; $\top_s$ is matched by all elements (in the sort universe s); and $\bot_s$ is matched by no elements. We drop sort s whenever possible, so we write $x, \top, \bot$ instead of $x{:}s, \top_s, \bot_s$.

Matching μ-Logic semantics In matching μ-logic, symbols are interpreted as *relations*, and thus matching μ-logic patterns evaluate to *sets of elements* (those "matching" them).

Definition 2. *Given $\mathbb{\Sigma} = (S, \Sigma)$, a* matching μ-logic $\mathbb{\Sigma}$-model $M = (\{M_s\}_{s\in S}, \{\sigma_M\}_{\sigma\in\Sigma})$, *or simply a* ($\mathbb{\Sigma}$)-model, *contains*

– *a nonempty carrier set M_s for each sort $s \in S$;*
– *an interpretation $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$ for each $\sigma \in \Sigma_{s_1\ldots s_n,s}$, where $\mathcal{P}(M_s)$ is the powerset of M_s.*

We overload the letter M to also mean the S-indexed set $\{M_s\}_{s\in S}$. We tacitly use the same letter σ_M to mean its *pointwise extension*, $\sigma_M : \mathcal{P}(M_{s_1})\times\cdots\times\mathcal{P}(M_{s_n}) \to \mathcal{P}(M_s)$, defined as:

$$\sigma_M(A_1,\ldots,A_n) = \bigcup\{\sigma_M(a_1,\ldots,a_n) \mid a_1 \in A_1,\ldots,a_n \in A_n\}$$

for all $A_1 \subseteq M_{s_1},\ldots,A_n \subseteq M_{s_n}$.

Proposition 1. *For all $A_i, A'_i \subseteq M_{s_i}$, $1 \le i \le n$, the pointwise extension σ_M has the following* property of propagation*:*

$$\sigma_M(A_1,\ldots,A_n) = \emptyset \textit{ if } A_i = \emptyset \textit{ for some } 1 \le i \le n,$$
$$\sigma_M(A_1 \cup A'_1,\ldots,A_n \cup A'_n) = \textstyle\bigcup_{1\le i\le n, B_i\in\{A_i,A'_i\}}\sigma_M(B_1,\ldots,B_n),$$
$$\sigma(A_1,\ldots,A_n) \subseteq \sigma(A'_1,\ldots,A'_n) \textit{ if } A_i \subseteq A'_i \textit{ for all } 1 \le i \le n.$$

Definition 3. *Let $\mathbb{\Sigma} = (S, \textsc{Var}, \Sigma)$ be a signature with $\textsc{Var} = \textsc{EVar} \cup \textsc{SVar}$, and $M = (\{M_s\}_{s\in S}, \{\sigma_M\}_{\sigma\in\Sigma})$ be a $\mathbb{\Sigma}$-model. A* valuation *$\rho\colon \textsc{Var} \to (M \cup \mathcal{P}(M))$ is a function such that $\rho(x) \in M_s$ for all $x \in \textsc{EVar}_s$ and $\rho(X) \in \mathcal{P}(M_s)$ for all $X \in \textsc{SVar}_s$. Its* extension *$\bar\rho\colon \textsc{Pattern} \to \mathcal{P}(M)$ can be inductively defined as:*

- $\bar\rho(x) = \{\rho(x)\}$ *for all $x \in \textsc{EVar}_s$;*
- $\bar\rho(X) = \rho(X)$ *for all $X \in \textsc{SVar}_s$;*
- $\bar\rho(\varphi_1 \wedge \varphi_2) = \bar\rho(\varphi_1) \cap \bar\rho(\varphi_2)$, *for $\varphi_1, \varphi_2 \in \textsc{Pattern}_s$;*
- $\bar\rho(\neg\varphi) = M_s \setminus \bar\rho(\varphi)$ *for all $\varphi \in \textsc{Pattern}_s$;*
- $\bar\rho(\exists x.\varphi) = \bigcup_{a\in M_{s'}} \overline{\rho[a/x]}(\varphi)$ *for all $x \in \textsc{Var}_{s'}$;*
- $\bar\rho(\sigma(\varphi_1, ..., \varphi_n)) = \sigma_M(\bar\rho(\varphi_1), ..., \bar\rho(\varphi_n))$ *for $\sigma \in \Sigma_{s_1\ldots s_n,s}$;*
- $\bar\rho(\mu X.\,\varphi) = \mu\mathcal{F}^{\rho}_{\varphi,X}$ *for all $X \in \textsc{SVar}_s$, where $\mathcal{F}^{\rho}_{\varphi,X}(A) = \overline{\rho[A/X]}(\varphi)$ for all $A \subseteq M_s$.*

where "$\setminus$" is set difference, $\rho[a/x]$ denotes the M-valuation ρ' with $\rho'(x) = a$ and $\rho'(y) = \rho(y)$ for all $y \not\equiv x$ and $\rho[A/X]$ is the ρ' with $\rho'(X) = A$ and $\rho'(Y) = \rho(Y)$ for all $Y \not\equiv X$. Note $\mathcal{F}^{\rho}_{\varphi,X}$ is monotone, since φ is positive in X.

Proposition 2. *The following propositions hold:*

- $\bar\rho(\top_s) = M_s$ *and* $\bar\rho(\bot_s) = \emptyset$*;*
- $\bar\rho(\varphi_1 \vee \varphi_2) = \bar\rho(\varphi_1) \cup \bar\rho(\varphi_2)$*;*
- $\bar\rho(\varphi_1 \to \varphi_2) = M_s \setminus (\bar\rho(\varphi_1) \setminus \bar\rho(\varphi_2))$ *for $\varphi_1, \varphi_2 \in \textsc{Pattern}_s$;*
- $\bar\rho(\varphi_1 \leftrightarrow \varphi_2) = M_s \setminus (\bar\rho(\varphi_1) \triangle \bar\rho(\varphi_2))$ *for $\varphi_1, \varphi_2 \in \textsc{Pattern}_s$;*
- $\bar\rho(\forall x.\varphi) = \bigcap_{a\in M_{s'}} \overline{\rho[a/x]}(\varphi)$, *for all $x \in \textsc{Var}_{s'}$;*

where "$\triangle$" is set symmetric difference.

Definition 4. *We say pattern φ is* valid *in M, written $M \vDash \varphi$, iff $\bar\rho(\varphi) = M$ for all $\rho\colon \textsc{Var} \to M$. Let Γ be a set of patterns called* axioms*. We write $M \vDash \Gamma$ iff $M \vDash \psi$ for all $\psi \in \Gamma$. We write $\Gamma \vDash \varphi$ and say that φ is* valid *in Γ iff $M \vDash \varphi$ for all $M \vDash \Gamma$. We abbreviate $\emptyset \vDash \varphi$ as $\vDash \varphi$. We call the pair $(\mathbb{\Sigma}, \Gamma)$ a* matching μ-logic $\mathbb{\Sigma}$-theory*, or simply a* ($\mathbb{\Sigma}$-)theory*. We say that M is* a model of the theory $(\mathbb{\Sigma}, \Gamma)$ *iff $M \vDash \Gamma$.*

Important notations In this section, we give a compact summary of some important definitions and notations that are needed in this proposal. All of them can be defined using patterns.

Definition 5. *For any (not necessarily distinct) sorts s, s', let us consider a unary symbol $\lceil \_ \rceil_s^{s'} \in \Sigma_{s,s'}$, called the* definedness symbol*, and the pattern/axiom $\lceil x{:}s \rceil_s^{s'}$, called* (Definedness)*. We define* totality "$\lfloor \_ \rfloor_s^{s'}$", equality "$=_s^{s'}$", membership "$\in_s^{s'}$", *and* set containment "$\subseteq_s^{s'}$" *as derived constructs:*

$$\lfloor \varphi \rfloor_s^{s'} \equiv \neg \lceil \neg \varphi \rceil_s^{s'} \qquad \varphi_1 =_s^{s'} \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor_s^{s'}$$
$$x \in_s^{s'} \varphi \equiv \lceil x \wedge \varphi \rceil_s^{s'} \qquad \varphi_1 \subseteq_s^{s'} \varphi_2 \equiv \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor_s^{s'}$$

and feel free to drop the (not necessarily distinct) sorts s, s'.

Functions and *partial functions* can be defined by axioms:

$$\text{(Function)} \qquad \exists y \,.\, \sigma(x_1, \ldots, x_n) = y$$
$$\text{(Partial Function)} \qquad \exists y \,.\, \sigma(x_1, \ldots, x_n) \subseteq y$$

(Function) requires $\sigma(x_1, \ldots, x_n)$ to contain exactly one element and (Partial Function) requires it to contain at most one element (recall that y evaluates to a singleton set).

Similarly, we say a pattern φ is *functional*, if it satisfies $\exists y \,.\, y = \varphi$; a pattern φ is *functional-like*, if it satisfies $(\exists y \,.\, y = \varphi) \vee \neg \lceil \varphi \rceil$;

Constructors are extensively used in building programs and data, as well as semantic structures to define and reason about languages and programs. They can be characterized in the "no junk, no confusion" spirit [23]. Let $\mathbb{\Sigma} = (S, \Sigma)$ be a signature and $C = \{c_i \in \Sigma_{s_i^1 \ldots s_i^{m_i}, s_i} \mid 1 \le i \le n\} \subseteq \Sigma$ be a set of symbols called *constructors*. Consider the following axioms/patterns:

(No Junk) for all sorts $s \in S$:

$$\bigvee_{c_i \in C \text{ with } s_i = s} \exists x_i^1{:}s_i^1 \ldots \exists x_i^{m_i}{:}s_i^{m_i} \,.\, c_i(x_i^1, \ldots, x_i^{m_i})$$

(No Confusion I) for all $i \neq j$ and $s_i = s_j$:

$$\neg (c_i(x_i^1, \ldots, x_i^{m_i}) \wedge c_j(x_j^1, \ldots, x_j^{m_j}))$$

(No Confusion II) for all $1 \le i \le n$:

$$(c_i(x_i^1, ..., x_i^{m_i}) \wedge c_i(y_i^1, ..., y_i^{m_i})) \rightarrow c_i(x_i^1 \wedge y_i^1, ..., x_i^{m_i} \wedge y_i^{m_i})$$

Intuitively, (No Junk) says everything is constructed; (No Confusion I) says different constructs build different things; and (No Confusion II) says constructors are injective.

Matching μ-Logic proof system We first give the definition to *context*:

Definition 6. *A* context *C is a pattern with a distinguished placeholder variable $\square$. We write $C[\varphi]$ to mean the result of* replacing $\square$ with φ without any α-renaming*, so free* *variables in φ may become bound in $C[\varphi]$,* different *from capture-avoiding substitution. A* single symbol context *has the form $C_\sigma \equiv \sigma(\varphi_1, \ldots, \varphi_{i-1}, \square, \varphi_{i+1}, \ldots, \varphi_n)$ where $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and $\varphi_1, \ldots, \varphi_{i-1}, \varphi_{i+1}, \ldots, \varphi_n$ are patterns of appropriate sorts. A* nested symbol context *is inductively defined as follows:*

- *$\square$ is a nested symbol context, called the* identity context*;*
- *if C_σ is a single symbol context, and C is a nested symbol context, then $C_\sigma[C[\square]]$ is a nested symbol context.*

Intuitively, a context C is a nested symbol context iff the path to $\square$ in C contains only symbols and no logic connectives.

$\mathcal{H}_\mu$, $\mathcal{H}$	(Propositional Tautology)	φ if φ is a propositional tautology over patterns of the same sort
	(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
	($\exists$-Quantifier)	$\varphi[y/x] \to \exists x.\varphi$
	($\exists$-Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x.\varphi_1) \to \varphi_2}$ if $x \notin FV(\varphi_2)$
	(Propagation$_\bot$)	$C_\sigma[\bot] \to \bot$
	(Propagation$_\vee$)	$C_\sigma[\varphi_1 \vee \varphi_2] \to C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
	(Propagation$_\exists$)	$C_\sigma[\exists x.\varphi] \to \exists x.C_\sigma[\varphi]$ if $x \notin FV(C_\sigma[\exists x.\varphi])$
	(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C_\sigma[\varphi_1] \to C_\sigma[\varphi_2]}$
	(Existence)	$\exists x.\, x$
	(Singleton Variable)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ where C_1 and C_2 are nested symbol contexts.
$\mathcal{H}_\mu$	(Set Variable Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(Pre-Fixpoint)	$\varphi[\mu X.\, \varphi/X] \to \mu X.\varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\, \varphi \to \psi}$

Figure 1. Sound and complete proof system $\mathcal{H}$ of matching logic (above the double line) and the proof system $\mathcal{H}_\mu$ of matching μ-logic

Figure 1 shows the proof system $\mathcal{H}_\mu$ for matching μ-logic. $\mathcal{H}$ is the proof system for matching logic. It has four categories of proof rules. The first consists of all propositional tautologies as axioms and (Modus Ponens). The second completes the (complete) axiomatization of pure predicate logic (two rules). The third category contains four rules that capture the property of propagation (Proposition 1). The fourth category contains two technical proof rules that are needed for the completeness result of $\mathcal{H}$.

$\mathcal{H}_\mu$ extends $\mathcal{H}$ with three additional proof rules (see Fig. 1). Rules (Pre-Fixpoint) and (Knaster-Tarski) are standard proof rules about least fixpoints as in modal μ-logic [34]; Rule (Set Variable Substitution) allows us to prove from $\vdash \varphi$ any substitution $\vdash \varphi[\psi/X]$ for $X \in$ SVar. That X is a set variable is crucial. Note that matching μ-logic cannot have a proof system that is both sound and complete for all theories, because one can capture precisely the model $(\mathbb{N}, +, \times)$ of natural numbers with addition and multiplication with a finite number of matching μ-logic axioms, and the model $(\mathbb{N}, +, \times)$, by Gödel's first incompleteness theorem [24], is not axiomatizable.

We let $\mathcal{H}_\mu$ denote the extended proof system in Fig. 1, and from here on we write $\Gamma \vdash \varphi$ instead of $\Gamma \vdash_{\mathcal{H}_\mu} \varphi$.

Theorem 1 (Soundness of $\mathcal{H}_\mu$). $\Gamma \vdash \varphi$ *implies* $\Gamma \vDash \varphi$.

2.2 Modal μ-Logic Preliminaries

We chose modal μ-logic [34] as the logic for expressing program properties. Modal *mu*-logic is one of the most important logics in model checking. Due to its expressiveness, a number of temporal logics can be translated into it. In this section, we give a brief introduction to matching μ-logic. We define transition system and modal μ-logic as theories in matching μ-logic in Section 3

Modal μ-Logic syntax and semantics The *syntax* of modal μ-logic [34] is parametric in a countably infinite set PVar of propositional variables. Modal μ-logic *formulas* are given by the grammar:

$$\varphi ::= p \in \text{PVar} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \circ\varphi \mid \mu X.\, \varphi \text{ if } \varphi \text{ is positive in } X$$

where $p, X \in \text{PVar}$ are propositional variables. As a convention, p is used for free variables while X is used for bound ones. Derived constructs are defined as usual, e.g., $\bullet\varphi \equiv \neg\circ\neg\varphi$.

Modal μ-logic semantics is given using *transition systems* $\mathbb{S} = (S, R)$, with S a nonempty set of *states* and $R \subseteq S \times S$ a *transition relation*, and valuations $V\colon \text{PVar} \to \mathcal{P}(S)$, as follows:

- $[\![p]\!]_V^{\mathbb{S}} = V(p)$;
- $[\![\varphi_1 \wedge \varphi_2]\!]_V^{\mathbb{S}} = [\![\varphi_1]\!]_V^{\mathbb{S}} \cap [\![\varphi_2]\!]_V^{\mathbb{S}}$;
- $[\![\neg\varphi]\!]_V^{\mathbb{S}} = S \setminus [\![\varphi]\!]_V^{\mathbb{S}}$;
- $[\![\circ\varphi]\!]_V^{\mathbb{S}} = \{s \in S \mid s\,R\,t \text{ implies } t \in [\![\varphi]\!]_V^{\mathbb{S}} \text{ for all } t \in S\}$;
- $[\![\mu X.\, \varphi]\!]_V^{\mathbb{S}} = \bigcap\{A \subseteq S \mid [\![\varphi]\!]_{V[A/X]}^{\mathbb{S}} \subseteq A\}$;

A modal μ-logic formula φ is *valid*, denoted $\vDash_\mu \varphi$, if for all transition systems $\mathbb{S}$ and all valuations V, we have $[\![\varphi]\!]_V^{\mathbb{S}} = S$.

3 Current Result

In this section, we define transition systems and modal μ-logic as theories in matching μ-logic. We also formalize the bounded model checking as proof searching in the proof system $\mathcal{H}_\mu$.

3.1 Defining transition systems in Matching μ-Logic

Transition systems are important in computer science. They are used to model various types of hardware and software systems, and many algorithms and techniques are proposed to analyze transition systems and to reason about their properties.

We define a signature of transition systems $\Sigma^{\mathsf{TS}} = (\{State\}, \{\bullet \in \Sigma^{\mathsf{TS}}_{State,State}\})$, where "$\bullet$" is a unary symbol (one-path next). We regard propositional variables in PVar as matching μ-logic set variables. We write $\bullet\varphi$ instead of $\bullet(\varphi)$, and define $\circ\varphi \equiv \neg\bullet\neg\varphi$. Then all modal μ-logic formulas φ are matching μ-logic Σ^{TS}-patterns of sort *State*. Σ^{TS}-models *exactly* captures the transition systems, where $\bullet \in \Sigma^{\mathsf{TS}}_{State,State}$ is interpreted as the transition relation R. Specifically, for any transition system $\mathbb{S} = (S, R)$, we can regard $\mathbb{S}$ as a Σ^{TS}-model where S is the carrier set of *State* and $\bullet_{\mathbb{S}}(t) = \{s \in S \mid s\, R\, t\}$ contains all *R-predecessors* of t. See the following illustration:

$$\begin{array}{llllll} \cdots & s & \xrightarrow{R} s' & \xrightarrow{R} s'' & \cdots & \text{// states} \\ & \bullet\bullet\varphi & \bullet\varphi & \varphi & & \text{// patterns} \end{array}$$

In other words, $\bullet\varphi$ is matched by states that *have at least one next state* that satisfies φ. Its dual $\circ\varphi \equiv \neg\bullet\neg\varphi$ (called "all-path next") is matched by s if *for all* states t such that $s\, R\, t$ and we have t satisfies φ. In particular, if s has no successor, then s matches $\circ\varphi$ for all φ. Figure 2 illustrates the meaning of $\bullet\varphi$ and $\circ\varphi$.

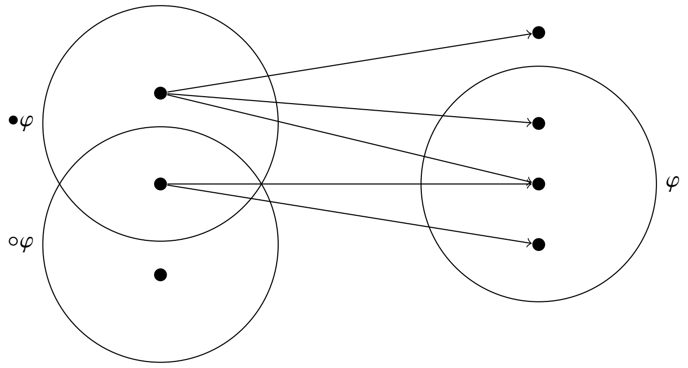

Figure 2. Strong next $\bullet\varphi$ and weak next $\circ\varphi$.

3.2 Language semantics as transition system

A Language semantics can be seen as the definition of a transition system. In $\mathbb{K}$ framework (`http://kframework.org`), a language semantics is defined as a finite set of *reachability rules* over the signature $\Sigma^{\mathsf{RS}} = \Sigma^{\mathsf{cfg}} \cup \{\bullet \in \Sigma_{Cfg,Cfg}\}$. Σ^{cfg} is the signature of *static program configurations*. It may have various sorts and symbols, among which there is a distinguished sort *Cfg*. Fix a Σ^{cfg}-model M^{cfg} and M^{cfg}_{Cfg} is the set of all configurations. *Reachability rules*, or simply *rules* have the form $\varphi_1 \Rightarrow \varphi_2$, where φ_1, φ_2 are Σ^{cfg}-patterns (without μ). A reachability system yields a transition system

$\mathbb{S} = (M^{\mathsf{cfg}}_{Cfg}, R)$ where $s\ R\ t$ if there exist a rule $\varphi_1 \Rightarrow \varphi_2 \in S$ and an M^{cfg}-valuation ρ such that $s \in \bar{\rho}(\varphi_1)$ and $t \in \bar{\rho}(\varphi_2)$.

In matching μ-logic, we axiomatize the same transition system by:

– desugar each reachablitiy rule $\varphi_{l_i} \Rightarrow \varphi_{r_i}$ to $\forall x_i.\varphi_{l_i} \to \bullet\varphi_{r_i}$, $x_i \in FV(\varphi_{l_i})$
– introduce a *STEP* axiom schema synthesized from all the rules in the semantics:

$$\varphi \to \circ \bigvee_{i=1} \exists x_i.\lceil \varphi_{l_i} \wedge \varphi \rceil \wedge \varphi_{r_i}$$

φ is an arbitrary pattern of sort *Cfg*.

We denote the resulting Σ^{TS}-theory Γ^{Lang}. The set of "one-step" axioms describes what should be included in the transition relation R. The *STEP* axiom ensures that no junk is added to the transition relation R. It is equivalent to say that the transition relation R is the *least* set that satisfies the reachability rules. Note that given the current configuration φ, $\varphi_w = \bigvee_{i=1} \exists x_i.\lceil \varphi_{l_i} \wedge \varphi \rceil \wedge \varphi_{r_i}$ captures the *least* set among all φ' such that $\varphi \to \circ\varphi'$ holds.

3.3 Defining Modal μ-Logic in Matching μ-Logic

Modal μ-logic can be regarded as an empty theory in a vanilla matching μ-logic without quantifiers, over a signature containing only one sort and only one symbol, which is unary. When reasoning about programs and their properties, we need to extend proposition variables in PVar to arbitrary static program configuration pattern of sort *Cfg*. Since we regard propositional variables in PVar as matching μ-logic set variables when defining the transition systems, it is safe to do so with any modification. We can add more temporal modalities as derived constructs:

$$\begin{aligned} \text{“eventually” } \Diamond\varphi &\equiv \mu X.\, \varphi \vee \bullet X \\ \text{“always” } \Box\varphi &\equiv \nu X.\, \varphi \wedge \circ X \\ \text{“(strong) until” } \varphi_1\ U\ \varphi_2 &\equiv \mu X.\, \varphi_2 \vee (\varphi_1 \wedge \bullet X) \\ \text{“well-founded” } \mathsf{WF} &\equiv \mu X.\circ X \quad \text{// no infinite paths} \end{aligned}$$

Note that these definitions work on general transition systems and there is no requirement on the model to be *linear future* or *infinite future* (e.g., LTL). We can use the same pattern to reason about transition systems that have terminating states.

3.4 Bounded Model Checking as Proof-Searching in $\mathcal{H}_\mu$

The above sections suggests that matching μ-logic may offer a unifying playground to specify and reason about transition systems, by means of Σ^{TS}-theories/models. As a result, we can specify the program configurations and their properties as matching μ-logic pattern and reason about the validity of the pattern in the proof system $\mathcal{H}_\mu$.

Since dealing with modal μ-logic in general is hard, we only consider some special cases. We assume that the pattern is of the form $\varphi_{\mathsf{init}} \to \varphi_{\mathsf{prop}}$. φ_{init} is a static configuration pattern(i.e., without μ or *next* symbol), which represents a set of initial configuration. Especially, we require the φ_{init} to be of the form $\varphi_1 \vee \cdots \vee \varphi_n$, where $\varphi_i = t_i(x) \wedge p_i(x)$. $t_i(x)$ is a pattern formed only with configuration constructors, domain values and variables. $p_i(x)$ is a predicate over variables. φ_{prop} encodes the temporal property that we want to check. It is a pattern of the form $\mu X.\, \varphi_{\mathsf{sub}}$ (or $\nu X.\, \varphi_{\mathsf{sub}}$). Moreover, we have four additional requirements on φ_{sub}:

- φ_{sub} does not contain μ, ν or $\bullet$, but $\circ$ is allowed.
- set variable X only appears under $\circ$.
- $\neg$ only appears before static configuration sub-patterns [1].
- sub-patterns of φ_{sub} do not have form $\circ\varphi_1 \vee \circ\varphi_2$

For example, given the IMP semantics [2], the following pattern checks the *non-division-by-zero* property [3]:

```
    (<k> while (true) {
           n = 100 / i;
           i = i + -1; } ~> K1 </k>
     <state> n |-> N:Int i |-> I:Int </state> ∧ I > 5)
→ □ (∀X.∀Y.∀K2.∀S.
        (<k> X:Int / Y:Int ~> K2 </k>
         <state> S </state>) → Y =/= 0)
```

Given the search depth k, bounded model checking the pattern $\varphi_{\mathsf{init}} \rightarrow \varphi_{\mathsf{prop}}$ can be seen as a fully automatic procedure to derive the proof tree of the pattern in the proof system $\mathcal{H}_\mu$ by limiting the depth of the proof tree. Since we only consider the finite future (bounded by depth k) of the model, we can simply use $\nabla X.\varphi = \varphi[\nabla X.\, \varphi/X]$, where $\nabla \in \{\mu, \nu\}$, to unroll the fixpoint pattern without worrying about whether it is a least or a greatest fixpoint.

The bounded model checking algorithm is described in the Algorithm 1. The algorithm uses the proof rules in $\mathcal{H}_\mu$ to derive the sub proof goals of the form $\varphi_{l'} \rightarrow \varphi_{r'}$from the current proof goal. Conceptually, it keeps doing equivalent rewriting of the current proof goal and separate sub proof goals $\varphi_{l'} \rightarrow \varphi_{r'}$ into:

(i) $\varphi_{r'}$ does not contain $\circ$ operator. We can check its validity in the current iteration.
(ii) $\varphi_{r'}$ does contain $\circ$ operator. We can only check its validity in the future iteration.

For condition (i), we can directly encode the pattern as Z3 query and check its validity. For condition (ii), we use *STEP* axiom to compute the "precise" pattern $\varphi_{w'}$ that represents the "all-path next" states of $\varphi_{l'}$. Since $\varphi_{l'} \rightarrow \circ\varphi_{w'}$, we only need to prove $\circ\varphi_{w'} \rightarrow \circ\varphi$ in order to prove $\varphi_{l'} \rightarrow \circ\varphi$. By applying (Framing) rule, we only need to prove $\varphi_{w'} \rightarrow \varphi$ in the next iteration. Note that the depth D in the bounded model checking algorithm constraints the maximum number of $\circ$ computed in each path starting from the root in the proof tree.

As an example, Figure 3 shows part of the proof tree for the pattern $\varphi_l \rightarrow \mu X.\varphi \wedge \circ X$. If φ_l does not have successors, φ_w becomes $\bot$ and $\varphi_w \rightarrow \mu X.\varphi \wedge \circ X$ becomes $\top$ immediately.

[1] patterns do not contain μ, ν or $\bullet$-derived symbols
[2] `https://github.com/kframework/k/blob/master/k-distribution/tutorial/1_k/2_imp/lesson_5/imp.k`
[3] $\square$ is a derived construct defined in Section 3.3

```
**Input:** $\varphi_{\text{init}} \to \varphi_{\text{prop}}$, Axiom set $\Gamma^{\text{Lang}}$, Depth D
**Output:** Proved, Failed Or Unknown
cGoals = $\{\varphi_{\text{init}} \to \varphi_{\text{prop}}\}$;
nGoals = $\emptyset$;
depth = $0$;
while depth $\leq D$ do
    increment depth;
    if cGoals is $\emptyset$ then
        return Proved;
    end
    while cGoals $\neq \emptyset$ do
        pick $\varphi_l \to \varphi_r$ from cGoals, cGoals = cGoals $\setminus \{\varphi_l \to \varphi_r\}$;
        if $\varphi_l = \varphi_{l_1} \vee \cdots \varphi_{l_n}$ then
            // case analysis
            cGoals = cGoals $\cup \{\varphi_{l_1} \to \varphi_r, \cdots, \varphi_{l_n} \to \varphi_r\}$;
        else if $\varphi_r$ doesn't contain $\circ$ operator then
            encode $\neg(\varphi_l \to \varphi_r)$ in Z3;
            if sat then
                return Failed;
            end
        else if $\varphi_r = \varphi_{r_1} \wedge \varphi_{r_2}$ then
            cGoals = cGoals $\cup \{\varphi_l \to \varphi_{r_1}, \cdots, \varphi_n \to \varphi_{r_2}\}$;
        else if $\varphi_r = \varphi_{r_1} \vee \varphi_{r_2}$ and $\varphi_{r_1}$ doesn't contain $\circ$ operator then
            cGoals = cGoals $\cup \{(\varphi_l \wedge \neg\varphi_{r_1}) \to \varphi_{r_2}\}$;
        else if $\varphi_r = \nabla X.\varphi$ then
            cGoals = cGoals $\cup \{\varphi_l \to \varphi[\nabla X.\varphi/X]\}$;
        else if $\varphi_r = \circ\varphi$ then
            compute $\varphi_w$ using STEP axiom in $\Gamma^{\text{Lang}}$;
            nGoals = nGoals $\cup \{\varphi_w \to \varphi\}$
    end
    cGoals = nGoals;
    nGoals = $\emptyset$;
end
if cGoals is $\emptyset$ then
    return Proved;
else
    return Unknown;
end
```

Algorithm 1: Bounded Model Checking

$$\dfrac{\dfrac{\dfrac{\cdots}{\varphi_l \to \varphi} \qquad \dfrac{\dfrac{\cdot}{\varphi \to \circ\varphi_w}\,(\text{STEP}) \quad \dfrac{\dfrac{\cdots}{\varphi_w \to \mu X.\varphi \wedge \circ X}}{\circ\varphi_w \to \circ(\mu X.\varphi \wedge \circ X)}\,(\textsc{Framing})}{\varphi_l \to \circ(\mu X.\varphi \wedge \circ X)}\,(\textsc{Modus Ponens})}{\varphi_l \to \varphi \wedge \circ(\mu X.\varphi \wedge \circ X)}}{\varphi_l \to \mu X.\varphi \wedge \circ X}$$

Figure 3. sub proof tree of the pattern $\varphi_l \to \mu X.\varphi \wedge \circ X$

3.5 Prototype and Result

I prototyped the above algorithm in the $\mathbb{K}$ haskell-backend [4]. In the prototype, I focused on the pattern $\varphi_{\mathsf{init}} \to \Box(\forall X.\varphi_{\mathsf{pattern}}(X) \to \varphi_{\mathsf{predicate}}(X))$. It is a very general global invariant pattern which can be used to check that $\varphi_{\mathsf{predicate}}(X)$ holds on every reachable state that matches $\varphi_{\mathsf{pattern}}(X)$ from the initial state φ_{init}. Properties such as *non-division-by-zero* and *assertions* can be encoded in this pattern.

My prototype can work on some simple semantics like IMP in the $\mathbb{K}$ framework. In one example, the prototype takes 311 seconds to explore 4494 symbolic states and sends over 14000 queries to Z3 solver [18]. The initial results show that there is large room for improving the performance of the bounded model checker.

4 Proposed Work

Before highlighting the plans for the rest of the thesis work, below is a summary of part of the thesis work accomplished so far:

- I worked on improving the performance of runtime verification technique and applying runtime verification to the robot domain.
- I helped to apply the $\mathbb{K}$' deductive program verifiers to high-profile commercial smart contracts and demonstrate its scalability and practicality.
- I defined programming language semantics as transition systems in the matching μ-logic. I formalized the bounded model checking as a proof searching problem in the matching μ-logic. I proposed an algorithm to do bounded model checking on many general patterns.
- I implemented a bounded model checker for all languages in the $\mathbb{K}$ haskell backend. I evaluated the checker on some simple semantics such as IMP.

The remaining part of the thesis work is about making the bounded model checker scalable and practical.

- To improve the performance of the bounded model checker. In the initial experiment, I found that most of the time is spent on querying Z3 [18]. I plan to do most of the reasoning locally and minimize the queries that are sent to Z3. What's more, I can take advantage of Z3's incremental solving ability to compute the "weak next" pattern faster by pushing and popping the conditions of the semantics rules.

[4] `https://github.com/kframework/kore`

- To evaluate the bounded model checker on real-world semantics such as KEVM [30]. The KEVM semantics is actively developed and used to verify several high-profile smart contracts. The ability to work with KEVM can demonstrate the scalability and practicality of the checker.
- To *conceptually* compare our approach with some existing bounded model checker such as CBMC [13] and CORRAL [36].
- To generate the proof object after model checking a pattern. If the bounded model checker returns *Failed*, it should provide a concrete counter-example to the user. If the checker returns *Proved* or *Unknown*, it should output the whole proof tree as the proof object.
- To model various abstraction and reduction techniques in the literature as proof strategy or theory transformations to the original semantics. On one hand, when deriving the sub-proof goals, instead of using the φ_w pattern for computing the exact weak next states, we can use patterns in the abstract domain. This maintains over-approximations for the reachable states. On the other hand, we can do theory transformation on the original operational semantics to reduce the size of the transition system while preserving the property to check. [21] demonstrates a similar approach in the context of partial order reduction [1, 22] for rewriting semantics of programming languages.

5 Related Work

I present related work for runtime verification, program verification and bounded model checking.

5.1 Runtime Verification

Runtime Verification (RV) [4,7,8,10,20,27,28,33] is a technique for monitoring program executions against formal properties. The general idea is that the RV tool instruments the program based on the properties so that executing instrumented programs generates appropriate events and creates *monitors* to listen to events and check the properties. Since the first papers on RV [27, 28], many techniques and tools were proposed in almost two decades. One direction of the research in RV focuses on reducing the time and memory overhead during execution [19, 32, 39, 43, 44] and supporting different formalism [5, 25, 40, 41]. Another direction of research aims to apply RV techniques to various domains such as testing [37] and robot [31]. However, little attention is paid to the correctness of the monitors themselves. Our bounded model checking approach can also be used in runtime verification. All we need to do is to encode the program and the concrete input in φ_{init} and set the search depth to infinity. Our approach will work no matter φ_{init} is concrete or symbolic. The advantage of our approach is that there is no need to trust the monitor code and the instrumentation tools, as a proof object will be generated during the execution.

Most of the RV tools translate the properties into state machines and generate monitor codes based on the state machines. For properties written in extended regular expressions and linear temporal logic, [50, 51] proposed a novel approach which uses *derivatives* to generate automata as monitors. In our approach, unrolling the μ binder is similar to calculating the derivatives. However, we unroll the program and the property on the fly during the execution instead of constructing the automata in advance.

5.2 Program Verification

Program Verification uses formal methods of mathematics to prove the correctness of a program. A popular approach to building program verifiers for real-world language is to translate to an intermediate verification language (IVL) and do verification at the IVL level. This advantage of this approach is that VC generation and reasoning about state properties are implemented only once, at the IVL level. Boogie [3] is a popular IVL integrated with Z3. There are several verifiers built on top of Boogie, such as VCC [15], HAVOC [35], and Dafny [38]. VCDrayd [42] is a separation logic based verifier built on top of VCC. Another approach advocates to develop language-independent formal methods [17,46,47,48,49] and reason about programs directly over *operational semantics*. This approach reduces the burden to specify multiple semantics and prove their soundness with respect to the operational semantics.

Our model checking technique follows the philosophy of the second approach and blurs the boundary between model checking and theorem proving. In fact, all path reachability formula $\varphi \Rightarrow^{\forall} \varphi'$ can be expressed in the matching μ-logic as $\varphi \rightarrow \nu X.\varphi' \vee (\circ X \wedge \bullet \top)$. In other words, our bounded model checking can be seen as a fully *automatic* proof tactics for proving patterns in matching μ-logic.

5.3 Model Checking

Model checking [14] has been used extensively to algorithmically check temporal properties of models. The literature on model checking is rich. We only discuss work on symbolic software model checking.

There are several model checkers for different languages. CBMC [13], ESBMC [16], Saturn [52] are widely used BMC tools for C programs. CBMC has both SAT-based and SMT-based backend. ESBMC is an SMT-based model checker which provide more accurate support for bit-level, pointers, unions and fixed point arithmetic. Saturn improves upon the basic bounded model checking algorithm by computing and memoizing relations between inputs and outputs (âĂIJsummariesâĂİ) for procedures bottom-up in the call graph. This makes bounded model checking scale to large programs. JavaPathFinder [26] is an execution-based model checker for muilti-threaded Java programs that combines exploring schedule choices explicitly with tracking user inputs symbolically. CORRAL [36] is a bounded model checker for Boogie [3] programs where the depth of recursion is bounded by a user-supplied recursion bound. It combines summaries, variable abstraction, and stratified inlining to solve the reachability-modulo-theories problem. The tools mentioned above were developed for a fixed target language and the language semantics is usually hardcoded within the implementation of their algorithms. On the contrary, our approach works for all languages and is faithful to the operational semantics of the target language.

For infinite state programs, symbolic model checking may not terminate, or take an inordinate amount of time or memory to terminate. Various abstraction technique were proposed to trade off precision of the analysis for efficiency. In practice, a well known approach is counterexample-guided abstraction refinement (CEGAR) [12]. The SLAM model checker [2] was the first implementation of CEGAR with predicate abstraction for C programs. Following SLAM project, BLAST [29] implements an optimization of CEGAR called lazy abstraction where the refinement step triggers the re-abstraction of only relevant parts of the original program. CEGAR has been generalized to check properties of heap-manipulating programs [6], as well as concurrent programs [9]. It is interesting to see how these techniques can be incorporated as proof search strategy in our model checker.

6 Conclusion

The goal of my thesis is to demonstrate and improve the scalability and practicality of the language-independent bounded model checker. To support this, I worked on the underlying theories and came up with an algorithm to bounded model check specific forms of modal μ-logic patterns. I implemented the idea in the $\mathbb{K}$ haskell-backend and made it work on simple language semantics such as IMP. I will thoroughly evaluate the model checker on real world language and applications. I believe that this approach reduces the trust base of the tool and would eliminate the need for dedicated model checkers for different languages. $\bullet\bot = \bot \circ\bot$

References

1. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Form. Methods Syst. Des.*, 18(2):97–116, March 2001.
2. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.
3. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 364–387, 2006.
4. Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. In *RV*, 2007.
5. Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *JLC*, 20(3), 2010.
6. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz, and Damien Zufferey. Shape refinement through explicit heap analysis. In David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, pages 263–277, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
7. Eric Bodden. MOPBox: A library approach to runtime verification. In *RV Tool Demo*, 2011.
8. Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with Tracematches. In *RV*, 2007.
9. Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 385–395, Washington, DC, USA, 2003. IEEE Computer Society.

10. Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *RV*, 2003.
11. Xiaohong Chen and Grigore Roşu. Matching mu-logic. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*, 2019.
12. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
13. Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *TACAS*, 2004.
14. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
15. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42, 2009.
16. L. Cordeiro, B. Fischer, and J. Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 137–148, Nov 2009.
17. Andrei Ştefănescu, Stefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *RTA*, volume 8560 of *LNCS*, pages 425–440, July 2014.
18. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.
19. Normann Decker, Jannis Harder, Torben Scheffel, and Daniel Schmitz, Malteand Thoma. Runtime monitoring with union-find structures. In *TACAS*, 2016.
20. Matthew B. Dwyer, Rahul Purandare, and Suzette Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis? In *RV*, 2010.
21. Azadeh Farzan and Jose Meseguer. Partial order reduction for rewriting semantics of programming languages. *Electronic Notes in Theoretical Computer Science*, 176, 07 2007.
22. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.
23. Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors. *Algebra, meaning, and computation*, volume 4060 of *Theoretical Computer Science and General Issues*. Springer, first edition, 2006.
24. Kurt Gödel. *On formally undecidable propositions of principia Mathematica and related systems*. Courier corporation, 1992.
25. Klaus Havelund, Doron Peled, and Dogan Ulus. First order temporal logic monitoring with BDDs. In *FMCAD*, 2017.
26. Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Mar 2000.
27. Klaus Havelund and Grigore Rosu. Monitoring java programs with java pathexplorer. In *RV*, 2001.
28. Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In *ASE*, 2001.
29. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.
30. Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Dwight Guth, Philip Daian, and Grigore Roşu. Kevm: A complete semantics of the ethereum virtual machine.

31. Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. Rosrv: Runtime verification for robots. In *Proceedings of the 14th International Conference on Runtime Verification*, volume 8734 of *LNCS*, pages 247–254. Springer International Publishing, September 2014.
32. Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. Garbage collection for monitoring parametric properties. In *PLDI*, 2011.
33. Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE Demo*, 2012.
34. Dexter Kozen. Results on the propositional μ-calculus. In *Proceedings of the 9th International Colloquium on Automata, Languages and Programming (ICALP'82)*, pages 348–359. Springer, 1982.
35. Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.
36. Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A solver for reachability modulo theories. In *Computer-Aided Verification (CAV)*, July 2012.
37. O. Legunsen, D. Marinov, and G. Roşu. Evolution-aware monitoring-oriented programming. In *ICSE NIER*, 2015.
38. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
39. Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick OâĂŹNeil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *RV*, 2014.
40. Patrick Meredith and Grigore Roşu. Efficient parametric runtime verification with deterministic string rewriting. In *ASE*, 2013.
41. P.O. Meredith, Dongyun Jin, Feng Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *ASE*, 2008.
42. Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI*, pages 440–451. ACM, 2014.
43. Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *OOPSLA*, 2010.
44. Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *ISSTA*, 2013.
45. Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4):1–61, 2017.
46. Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.
47. Grigore Roşu and Andrei Ştefănescu. From Hoare logic to matching logic reachability. In *FM*, volume 7436 of *LNCS*, pages 387–402, 2012.
48. Grigore Roşu and Andrei Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP*, volume 7392 of *LNCS*, pages 351–363, 2012.
49. Grigore Roşu, Andrei Ştefănescu, Stefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS*, pages 358–367. IEEE, 2013.
50. Koushik Sen and Grigore Rosu. Generating optimal monitors for extended regular expressions. In *Proceedings of 3rd International Workshop on Runtime Verification (RV'03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*, pages 226–245. Elsevier, June 2003.
51. Koushik Sen, Grigore Rosu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In *Proceedings of the 8th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of *Lecture Notes in Computer Science (LNCS)*, pages 260–275. Springer-Verlag, December 2003.

52. Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 351–363, 2005.