Software Model Checking via Iterative Abstraction Refinement of Constraint Logic Queries

Cormac Flanagan
Department of Computer Science
University of California, Santa Cruz

Abstract

Existing predicate abstraction tools rely on both theorem provers (to abstract the original program) and model checkers (to check the abstract program). This paper combines these theorem proving and model checking components in a unified algorithm. The correctness of the original, infinite-state program is expressed as a single query in *constraint logic*, which is sufficiently expressive to encode recursion and least fixed-point computations. The satisfiability of this query is decided using a combination of predicate abstraction, counterexample-based predicate inference, and proof-based explication. Our algorithm avoids the Cartesian approximation while reducing the number of theorem prover queries.

1 Introduction

The combination of predicate abstraction and iterative abstraction refinement has emerged as a promising strategy for model checking imperative, infinite-state software [1, 15]. Existing tools leverage a theorem prover to generate an abstraction of the original imperative program, and proceed to model check this abstract, imperative program. This paper presents a unified approach that combines the theorem proving and model checking components in a single architecture that is potentially more efficient.

In this approach, the correctness of the original, infinite-state program is expressed as a single logical query. To accommodate iterative and recursive constructs in the original program, we express this query in an extended *constraint logic* (see [16]) that can directly express recursive and least fixed-point computations. The translation of imperative, infinite-state software into constraint logic is the subject of a previous paper [9]; this paper focuses on determining the satisfiability of the resulting constraint logic query. We present an algorithm for deciding the satisfiability of constraint logic queries using a combination of predicate abstraction, counterexample-based predicate inference, and proof-based explication. Performing the entire reasoning process within the constraint logic framework yields a unified algorithm as well as potential gains in efficiency.

Our algorithm works by iteratively abstracting the given constraint logic query to a decidable boolean query. This abstraction is conservative, in that if the boolean query is unsatisfiable then so is the constraint logic query, and the original program therefore satisfies the desired correctness property. If the boolean query is satisfiable, we compute a *derivation* or *trace* for the boolean query and a corresponding trace for the constraint logic query. If this constraint logic trace is also satisfiable, it corresponds to a feasible execution of the orignal program that violates the correctness property. If the constraint logic trace is unsatisfiable, we use the proof of unsatisfiability to efficiently refine the abstraction so that this trace is excluded on future iterations of the algorithm.

This refinement process infers additional predicates, in a similar fashion to SLAM [1] and BLAST [15]. In addition, the refinement process also *explicates* relevant facts of the underlying theories in a manner reminiscent of explicating theorem provers [10, 2] and the sucessive approximation technique of Das and Dill [5]. This explication achieves a similar result to the (exponentially-many) theorem prover queries performed in predicate abstraction, but explicates this information in a more lazy fashion. This lazy approach avoids both the Cartesian abstraction [15] while reducing the number of theorem prover queries necessary for predicate abstraction.

The presentation of our approach proceeds as follows. Section 2 illustrates the generation of constraint logic queries on an example program. Section 3 contains a brief review of the constraint and boolean logics that we use. Section 4 describes the abstraction of constraint logic queries. Section 5 presents our iterative abstraction algorithm, and Section 6 describes how satisfiable traces are used to refine the current abstraction. Section 7 describes related work, and we conclude with section 8.

2 Software Model Checking via Constraint Logic

To illustrate the use of constraint logic for software model checking, consider the example program Rational shown in Figure 1(a). This class implements rational numbers, where a rational is represented as a pair of integers for the numerator and denominator. The example also contains a test harness, which reads in two integers, `x` and `y`, ensures that `y` is not zero, creates a corresponding rational, and then repeatedly prints out the truncation of the rational. We wish to check that a division-by-zero error never occurs, and we express this property as an assertion in the `trunc` method.

Checking such correctness properties of software is, in general, quite challenging, particularly for imperative, infinite-state software with heap-allocated data structures. In an earlier paper [9], we proposed checking such programs via translation into constraint logic. This approach translates each routine m in the program into two relations in the constraint logic:

1. The *error relation* $\mathbf{E}\mathrm{m}(\mathit{state})$, which describes states from which the execution of m may go wrong by failing an assertion.

2. The *transfer relation* $\mathbf{T}\mathrm{m}(\mathit{state}, \mathit{state}')$, which, when m terminates normally, describes the relation between the pre-state and post-state of m.

Loops in the source program may be accomodated in our framework by desugaring them into tail-recursive routines, which then yield additional relations. The query **E**`main`() is then satisfiable if the original program starting from `main` may go wrong by failing an assertion. In this case, the satisfying derivation for the query corresponds to an erroneous execution trace for the original program.

For the Rational program, the error and transfer relations are shown in Figure 1(b). (We follow Lamport's use of $\wedge$ and $\vee$ as bullets in large formulas for clarity [18].) With respect to these

```
class Rat {

  int n, d;

  Rat(int x, int y) {
    n = x;
    d = y;
  }

  int trunc() {
    assert d != 0;
    return n/d;
  }
}

public void main() {
  int x = readInt();
  int y = readInt();
  if( y == 0 ) {
    return;
  }
  Rat r=new Rat(y,x);
  int i=0;
  while (i<100000) {
    print( r.trunc() );
    i++;
  }
}
```

$$\begin{array}{l}
\mathbf{T}\mathrm{Rat}(x,y,n,d,h,n',d',h',\mathit{this}) \leftarrow \\
\quad \wedge\ \mathit{select}(h,\mathit{this})=0 \\
\quad \wedge\ h'=\mathit{store}(h,\mathit{this},1) \\
\quad \wedge\ n'=\mathit{store}(n,\mathit{this},x) \\
\quad \wedge\ d'=\mathit{store}(d,\mathit{this},y) \\
\\
\mathbf{E}\mathrm{trunc}(\mathit{this},n,d) \leftarrow \\
\quad \mathit{select}(d,\mathit{this})=0 \\
\\
\mathbf{T}\mathrm{readInt}(r) \leftarrow \\
\quad \mathit{isInt}(r) \\
\\
\mathbf{E}\mathrm{loop}(\mathit{this},i,n,d) \leftarrow \\
\quad \wedge\ i<100000 \\
\quad \wedge\ \ \vee\ \mathbf{E}\mathrm{trunc}(\mathit{this},n,d) \\
\qquad \vee\ \mathbf{E}\mathrm{loop}(\mathit{this},i+1, \\
\qquad\qquad\qquad n,d) \\
\\
\mathbf{E}\mathrm{main}() \leftarrow \\
\quad \wedge\ \mathbf{T}\mathrm{readInt}(x) \\
\quad \wedge\ \mathbf{T}\mathrm{readInt}(y) \\
\quad \wedge\ y \neq 0 \\
\quad \wedge\ \mathbf{T}\mathrm{Rat}(y,x,n,d,h, \\
\qquad\qquad\qquad n',d',h',\mathit{this}) \\
\quad \wedge\ \mathbf{E}\mathrm{loop}(\mathit{this},0,n',d')
\end{array}$$

$$\begin{array}{l}
\mathbf{T}\mathrm{Rat}(\llbracket d'=\mathit{store}(d,\mathit{this},y)\rrbracket) \leftarrow \\
\quad \wedge\ \mathit{isTrue}(\llbracket \mathit{select}(h,\mathit{this})=0\rrbracket) \\
\quad \wedge\ \mathit{isTrue}(\llbracket h'=\mathit{store}(h,\mathit{this},1)\rrbracket) \\
\quad \wedge\ \mathit{isTrue}(\llbracket n'=\mathit{store}(n,\mathit{this},x)\rrbracket) \\
\quad \wedge\ \mathit{isTrue}(\llbracket d'=\mathit{store}(d,\mathit{this},y)\rrbracket) \\
\\
\mathbf{E}\mathrm{trunc}(\llbracket \mathit{select}(d,\mathit{this})=0\rrbracket) \leftarrow \\
\quad \mathit{isTrue}(\llbracket \mathit{select}(d,\mathit{this})=0\rrbracket) \\
\\
\mathbf{T}\mathrm{readInt}() \leftarrow \\
\quad \mathit{isTrue}(\llbracket \mathit{isInt}(r)\rrbracket) \\
\\
\mathbf{E}\mathrm{loop}(\llbracket \mathit{select}(d,\mathit{this})=0\rrbracket) \leftarrow \\
\quad \wedge\ \mathit{isTrue}(\llbracket i<100000\rrbracket) \\
\quad \wedge\ \ \vee\ \mathbf{E}\mathrm{trunc}(\llbracket \mathit{select}(d,\mathit{this})=0\rrbracket) \\
\qquad \vee\ \mathbf{E}\mathrm{loop}(\llbracket \mathit{select}(d,\mathit{this})=0\rrbracket) \\
\\
\mathbf{E}\mathrm{main}() \leftarrow \\
\quad \wedge\ \mathbf{T}\mathrm{readInt}() \\
\quad \wedge\ \mathbf{T}\mathrm{readInt}() \\
\quad \wedge\ \mathit{isTrue}(\llbracket y\neq 0\rrbracket) \\
\quad \wedge\ \mathbf{T}\mathrm{Rat}(\llbracket d'=\mathit{store}(d,\mathit{this},y)\rrbracket) \\
\quad \wedge\ \mathbf{E}\mathrm{loop}(\llbracket \mathit{select}(d,\mathit{this})=0\rrbracket) \\
\quad \wedge\ (\quad \mathit{isTrue}(\llbracket \mathit{select}(d',\mathit{this})=0\rrbracket) \\
\qquad\quad \wedge\ \mathit{isTrue}(\llbracket d'=\mathit{store}(d,\mathit{this},y)\rrbracket) \\
\qquad\quad \Rightarrow\ \mathit{isTrue}(\llbracket y=0\rrbracket))
\end{array}$$

Figure 1: The example program `Rational` (column 1), the corresponding constraint logic rule set (column 2), and a boolean abstraction of that rule set.

relations, the query **`Emain`**() is satisfiable, indicating an error in the program. An investigation of the satisfying derivation reveals the source of the error: the arguments are passed to the `Rat` constructor in the wrong order. Note that since both arguments are integers, standard type systems do not catch this error.

After fixing this bug, the query **`Emain`**() is now unsatisfiable, indicating that a division-by-zero error cannot occur. However, a standard (depth-first) constraint logic implementation, such as SICStus Prolog [23], explicitly examines all 100,000 possible execution paths before answering that the query is unsatisfiable. Even worse, a real program typically has infinitely many execution paths, and so a standard depth-first search would diverge on the corresponding constraint logic query. By comparison, our algorithm determines the unsatisfiability of **`Emain`**() in just two iterations.

3 Constraint Logic

This section provides an overview of our notation and syntax. We write $\vec{X}$ for a (possibly empty) sequence $X_1, \ldots, X_n$. We let x, y, z range over variables. A *term* t is either a variable or the application of a function f to a sequence of terms. A *primitive constraint* $p(\vec{t})$ is the application of a predicate p to a term sequence. A *literal* is a primitive constraint or its negation. *Constraints* include literals, conjunctions, and disjunctions.

We let r, s range over user-defined relations. An *atom* $r(\vec{t})$ is the application of a user-defined relation r to a term sequence $\vec{t}$. *Formulas* extend constraints with atoms in positive positions. A *rule* $r(\vec{x}) :\!-\ e$ provides a definition of the relational symbol r. For example, the rule $r(x, y) :\!-\ x = y$ defines r as the identity relation. A *rule set* P is a sequence of rules, where each relation r mentioned in the rule set has a unique defining rule. The operation $e[x := t]$ denotes the substitution of term t for free occurrences of x within the formula e. The function *vars* extracts the free variables of a formula.

Syntax

(*terms*)	t	$::=$	$x \mid f(\vec{t})$	(*constraints*)	c	$::=$	$l \mid c \wedge c \mid c \vee c$
(*prim. constraints*)	h	$::=$	$p(\vec{t})$	(*formulas*)	e	$::=$	$l \mid e \wedge e \mid e \vee e \mid a$
(*literals*)	l	$::=$	$h \mid \neg h$	(*rules*)	d	$::=$	$r(\vec{x}) :\!-\ e$
(*atoms*)	a	$::=$	$r(\vec{t})$	(*rule set*)	P	$::=$	$\vec{d}$

The constraint logic language $CLP(\mathcal{D})$ is parameterized by an underlying constraint domain $\mathcal{D}$. The constraint domain determines the set of function and predicate symbols from which programs may be constructed, and may associate an intended interpretation with those symbols. We require that the set of predicate symbols includes equality, with the usual semantics, in order to express parameter passing. In addition, for model checking many program correctness properties [11], the constraint domain $\mathcal{D}$ may also include linear arithmetic, functional maps (with the *select* and *store* functions), and equality with uninterpreted function symbols (EUF).

Constraint logic rules may be self- or mutually-recursive, and so a rule set may yield multiple models. We are only interested in the least model that is compatible with the intended interpretation $\mathcal{D}$ of the functions and predicates. A $CLP(\mathcal{D})$ *query* is to determine if the least compatible model of the rule set P implies a particular *goal* or nullary relation symbol r, which we write as $P \models_{\mathcal{D}} r(\,)$.

3.1 Traces

Each user-defined relation may be applied multiple times in a rule set. To help distinguish these applications when reasoning about derivations, we associate with each relation symbol r an unbounded number of *variant* relation symbols $s_1, s_2, \ldots$. The function *base* maps each variant back to the original relation symbol: if $base(s) = r$, then s is a variant of r. We require that each relation symbol r in the original rule set is a variant of itself, that is, $base(r) = r$.

The unrolling relation $\cdot \to \cdot$ on formulas replaces each call to a relation r with a call to some variant s of r, and makes a committed choice on each disjunction. It is formalized as the least

$$
\begin{array}{l}
\textbf{E}\text{main}() \leftarrow \\
\quad \wedge \quad \textbf{T}\text{readInt}^1(x) \\
\quad \wedge \quad \textbf{T}\text{readInt}^2(y) \\
\quad \wedge \quad y \neq 0 \\
\quad \wedge \quad \textbf{T}\text{Rat}^3(x, y, n, d, h, n', d', h', \mathit{this}) \\
\quad \wedge \quad \textbf{E}\text{loop}^4(\mathit{this}, 0, n', d') \\
\textbf{T}\text{readInt}^1(r) \leftarrow \\
\qquad \mathit{isInt}(r) \\
\textbf{T}\text{readInt}^2(r) \leftarrow \\
\qquad \mathit{isInt}(r) \\
\textbf{T}\text{Rat}^3(x, y, n, d, h, n', d', h', \mathit{this}) \leftarrow \\
\quad \wedge \quad \mathit{select}(h, \mathit{this}) = 0 \\
\quad \wedge \quad h' = \mathit{store}(h, \mathit{this}, 1) \\
\quad \wedge \quad n' = \mathit{store}(n, \mathit{this}, x) \\
\quad \wedge \quad d' = \mathit{store}(d, \mathit{this}, y) \\
\textbf{E}\text{loop}^4(\mathit{this}, i, n, d) \leftarrow \\
\quad \wedge \quad i < 100000 \\
\quad \wedge \quad \textbf{E}\text{trunc}^5(\mathit{this}, n, d) \\
\textbf{E}\text{trunc}^5(\mathit{this}, n, d) \leftarrow \\
\qquad \mathit{select}(d, \mathit{this}) = 0
\end{array}
$$

Figure 2: A trace for the fixed Rational program.

relation such that, if $e_i \to e'_i$ and $base(s) = r$, then

$$
\begin{array}{rcl}
r(\vec{t}) & \to & s(\vec{t}) \\
l & \to & l \\
e_1 \wedge e_2 & \to & e'_1 \wedge e'_2 \\
e_1 \vee e_2 & \to & e'_i
\end{array}
$$

An *instance* of a rule $r(\vec{x}) :- e$ is a rule $s(\vec{x}) :- e'$ that defines the variant s of r as some unrolling e' of e, where $e \to e'$.

Given a rule set P with goal r, a *trace* is essentially a derivation of why $r()$ is true with respect to P. To facilitate our technical development, a trace of P is formalized as a rule set T that (1) contains a rule for the goal, (2) only contains instances of rules from P, and (3) does not contain recursive invocations. The following lemma states that every satisfiable query has a satisfiable trace.

Lemma 1 *$P \models_{\mathcal{D}} r(\,)$ if and only if there exists a trace T of P such that $T \models_{\mathcal{D}} r(\,)$.*

To illustrate this idea, Figure 2 shows a trace for the Rational program (with the bug removed). We use numeric superscripts to denote variants of relations in the original rule set.

3.2 Boolean Logic Programming

Boolean logic is a particular instance $CLP(\mathcal{B})$ of the constraint logic paradigm, were the only values in the underlying domain $\mathcal{B}$ are the boolean constants *true* and *false*; there are no function symbols;

and there is a single unary predicate symbol called $isTrue$, which only holds on the value *true*. Because of the domain $\mathcal{B}$ is finite, $CLP(\mathcal{B})$ queries are decidable, and thus are a natural target for abstracting $CLP(\mathcal{D})$ queries.

4 Abstracting Constraint Logic Queries

An *abstraction* $\alpha_r = (\alpha_r^a, \alpha_r^e)$ for a constraint logic rule $r(\vec{x}) :- e$ is a pair that describes how to abstract that rule into a corresponding boolean logic rule. The first component α_r^a is a sequence of primitive constraints over $\vec{x}$, and provides an *abstract interface* for r. Essentially, whereas r defines some relation over the formal parameters $\vec{x}$, the abstraction of r defines a coarser or larger relation that is expressible as a boolean combination of the constraints in α_r^a. We require that α_r^a is empty whenever $\vec{x}$ is empty.

The second component α_r^e of the abstraction is a constraint that explicates various axioms and properties of the semantic domain $\mathcal{D}$, and should be a tautology with respect to $\mathcal{D}$. Eventually, the explicated constraints of the various rules should contain enough information about $\mathcal{D}$ that the unsatisfiability of the query follows from these explicated constraints by purely propositional reasoning.

An abstraction α for a rule set P provides an abstraction for each relation symbol r defined in the rule set. We use the notation α_r^a to denote the abstract interface for the relation r under this abstraction; similarly α_r^e denotes the explicated constraints for r; and α_r denotes the pair (α_r^a, α_r^e).

A suitable abstraction δ for the fixed Rational program is given by:

$$
\begin{array}{lcl}
\delta_{\mathbf{E}\text{trunc}} & = & \langle\{select(d, this) = 0\},\ true\rangle \\
\delta_{\mathbf{E}\text{loop}} & = & \langle\{select(d, this) = 0\},\ true\rangle \\
\delta_{\mathbf{T}\text{Rat}} & = & \langle\{d' = store(d, this, y)\},\ true\rangle \\
\delta_{\mathbf{E}\text{main}} & = & \langle\emptyset,\ (select(d', this) = 0 \wedge d' = store(d, this, y)) \Rightarrow y = 0\rangle \\
\delta_{\mathbf{T}\text{readInt}} & = & \langle\emptyset,\ true\rangle
\end{array}
$$

This abstraction states that the behavior of the relations **E**trunc and **E**loop crucially depends on whether $select(d, this) = 0$. In particular, these relations hold whenever their arguments satisfy this constraint. Similarly, the primitive constraint $d' = store(d, this, y)$ is relevant to the behavior of **T**Rat. Finally, the functions *select* and *store* have a particular intended interpretation satisfying the axiom

$$(select(d', this) = 0 \wedge d' = store(d, this, y)) \Rightarrow y = 0$$

and this axiom is included in the explicated constraint for **E**main, since it is useful when reasoning about **E**main.

4.1 Performing Abstractions

Let $[\![\cdot]\!]$ map primitive constraints to boolean logic variables in an injective manner. Each primitive constraint h is mapped to the boolean variable $[\![h]\!]$ that represents the truth value of h. Given an abstraction α, we extend $[\![\cdot]\!]$ to a function $[\![\cdot]\!]^\alpha$ that maps constraint logic formulas and rules to corresponding formulas and rules in boolean logic: see Figure 3. The map $[\![\cdot]\!]^\alpha$ preserves the propositional structure of formulas, and maps each primitive constraint h to a boolean constraint $isTrue([\![h]\!])$. For a relation invocation $r(\vec{t})$, instead of passing the (infinite state) values described

$$
\begin{array}{rcl}
[\![h]\!]^\alpha & = & isTrue([\![h]\!]) \\
[\![\neg h]\!]^\alpha & = & \neg isTrue([\![h]\!]) \\
[\![e_1 \wedge e_2]\!]^\alpha & = & [\![e_1]\!]^\alpha \wedge [\![e_2]\!]^\alpha \\
[\![e_1 \vee e_2]\!]^\alpha & = & [\![e_1]\!]^\alpha \vee [\![e_2]\!]^\alpha \\
[\![r(\vec{t})]\!]^\alpha & = & r([\![h'_1]\!] \dots [\![h'_n]\!]) \\
 & & \text{where} \quad r \text{ is defined as } r(\vec{x}) :- e \\
 & & \qquad\quad \alpha_r^a = h_1 \dots h_n \\
 & & \qquad\quad h'_i = h_i[\vec{x} := \vec{t}] \\
[\![r(\vec{x}) :- e]\!]^\alpha & = & r([\![h_1]\!] \dots [\![h_n]\!]) :- [\![e]\!]^\alpha \wedge [\![\alpha_r^e]\!]^\alpha \\
 & & \text{where} \quad \alpha_r^a = h_1 \dots h_n \\
[\![d_1 \dots d_n]\!]^\alpha & = & [\![d_1]\!]^\alpha \dots [\![d_n]\!]^\alpha
\end{array}
$$

Figure 3: The abstraction function $[\![\cdot]\!]^\alpha$.

by the terms $\vec{t}$, the abstract interface α_r^a of r is inspected to see what properties $h_1 \dots h_n$ of the formal parameters $\vec{x}$ are considered relevant, the corresponding properties $h'_i = h_i[\vec{x} := \vec{t}]$ of the argument terms $\vec{t}$ are computed, and the boolean variables $[\![h'_1]\!] \dots [\![h'_n]\!]$ are passed instead.

When abstracting a rule definition $r(\vec{x}) :- e$, the abstract interface α_r^a is consulted to determine the relevant properties $h_1 \dots h_n$ of $\vec{x}$, and the $CLP(\mathcal{B})$ variables $[\![h_1]\!] \dots [\![h_n]\!]$ are used as formal parameters of the abstract rule for r. The boolean abstraction of each $CLP(\mathcal{D})$ primitive constraint h to the $CLP(\mathcal{B})$ variable $[\![h]\!]$ ignores the semantics of $CLP(\mathcal{D})$ constraints. The explicated constraint α_r^e compensates for this loss of information by including properties or instantiated axioms about the domain $\mathcal{D}$, and its abstraction $[\![\alpha_r^e]\!]^\alpha$ is included in the body of the abstract rule for r.

To illustrate this translation, Figure 1(c) shows the boolean rule set generated from the fixed Rational example, using the abstraction δ of Section 4. This boolean query is unsatisfiable, indicating that the abstraction δ is precise enough to verify the correctness of the fixed Rational example.

The abstraction mapping is conservative in that it may make additional queries satisfiable, but will never make a satisfiable query unsatisfiable.

Lemma 2 *If* $P \models_\mathcal{D} r()$ *then* $[\![P]\!]^\alpha \models_\mathcal{B} r()$.

4.2 Ordering Abstractions

Abstractions enjoy a partial ordering: $\alpha \sqsubseteq \beta$ if and only if for all relations r, $\alpha_r^a \subseteq \beta_r^a$ and $\alpha_r^e \Leftarrow \beta_r^e$. This ordering has minimal element $\bot$ (that describes the coarsest abstraction) and join operation $\alpha \wedge \beta$ defined by:

$$
\begin{array}{rcl}
\bot_r & = & \langle \emptyset, true \rangle \\
(\alpha \wedge \beta)_r^a & = & \alpha_r^a \cup \beta_r^a \\
(\alpha \wedge \beta)_r^e & = & \alpha_r^e \wedge \beta_r^e
\end{array}
$$

The abstraction mapping is anti-monotonic: if a query is satisfiable under some abstraction α, then it is also satisfiable under any smaller (coarser) abstraction β.

Lemma 3 *If* $[\![P]\!]^\alpha \models_\mathcal{B} r()$ *and* $\beta \sqsubseteq \alpha$ *then* $[\![P]\!]^\beta \models_\mathcal{B} r()$.

Algorithm CAR
Input query: $P \models_{\mathcal{D}} r(\,)$
Output: "satisfiable" or "unsatisfiable"

```
α := ⊥;
while true {
    P′ := [[P]]^α;
    if P′ ⊭_B r then return "unsatisfiable";        // Lemma 2
    T′ := satisfiable trace for P′;                  // Lemma 1
    T := corresponding trace for P;                  // Lemma 4
    γ := ETC(T);
    if γ = "satisfiable" then return "satisfiable";
    α := α ∧ collapse(γ);
}
```

Figure 4: The algorithm CAR.

Conversly, any more precise abstraction $\delta' \sqsupseteq \delta$ would suffice to verify that the query for the fixed Rational program is unsatisfiable, and hence that that program does not go wrong.

4.3 Abstractions for Traces

The operation $replicate(\alpha)$ replicates the abstraction for a relation r to all of its variants. The inverse operation $collapse(\alpha)$ generates an abstraction for r as the join of the abstractions of its variants.

$$\begin{array}{rcl} replicate(\alpha)_s & = & \alpha_{base(s)} \\ collapse(\alpha)_r & = & \wedge\,\{\alpha_s \mid base(s) = r\} \end{array}$$

These operations enjoy the following properties: For any abstraction α,

1. $\alpha = collapse(replicate(\alpha))$,
2. $\alpha \sqsubseteq replicate(collapse(\alpha))$, and
3. *collapse* and *replicate* are monotonic.

In addition, given a trace for the abstraction of a rule set P, we can generate a corresponding trace for P.

Lemma 4 (Corresponding traces) *Suppose α is an abstraction for P and T is a trace for $[\![P]\!]^\alpha$. Then there exists a trace T' for P such that T is also a trace for $[\![T']\!]^{replicate(\alpha)}$.*

5 Deciding Constraint Logic Queries

We now present a semi-algorithm CAR (constraint abstraction refinement) that decides constraint logic queries by iterative abstraction refinement. The algorithm, shown in Figure 4, abstracts the given constraint logic query to a decidable boolean query. If the boolean query is unsatisfiable, then so is the original query, by Lemma 2. Otherwise we use a satisfiable trace from the boolean query to construct (by Lemma 4) a corresponding trace of the constraint logic query, which may or may not be satisfiable. If the constraint logic trace is satisfiable, then so is the original constraint logic query, by Lemma 1. Otherwise we use the unsatisfiable trace to refine the abstraction so that this trace is excluded the from future consideration, and repeat the process.

Our CAR algorithm leverages the Explicating Trace Checker (ETC) algorithm for constraint logic traces described in the following section. Given a constraint logic trace T and goal r such that $T \models_{\mathcal{D}} r(\,)$ is unsatisfiable, the ETC algorithm generates an abstraction γ such that the abstract query $[\![T]\!]^\gamma \models_{\mathcal{D}} r(\,)$ is unsatisfiable. If $T \models_{\mathcal{D}} r(\,)$ is satisfiable, the ETC algorithm returns "satisfiable".

Lemma 5 (Correctness) *The* CAR *algorithm only reports correct answers.*

Proof: If the algorithm reports unsatisfiable, then $[\![P]\!]^\alpha \not\models_{\mathcal{B}} r$, and hence by Lemma 2, $P \not\models_{\mathcal{D}} r(\,)$. If the algorithm reports satisfiable, then the query has satisfying trace, and hence by Lemma 1, $P \models_{\mathcal{D}} r(\,)$.

Lemma 6 (Progress) *The* CAR *algorithm never applies the* ETC *algorithm to a particular trace more than once.*

Proof: Suppose the algorithm considers a concrete trace T. If T is unsatisfiable, then ETC returns an abstraction γ such that $[\![T]\!]^\gamma$ is unsatisfiable, and the abstraction α is joined with $\mathit{collapse}(\gamma)$. Since α only increases, at any later stage we have $\alpha \sqsupseteq \mathit{collapse}(\gamma)$.

If the trace T is later reconsidered, then T comes from some satisfiable trace $[\![T']\!]^{\mathit{replicate}(\alpha)}$ for $[\![\alpha]\!]^\alpha$. But $\mathit{replicate}(\alpha) \sqsupseteq \mathit{replicate}(\mathit{collapse}(\gamma)) \sqsupseteq \gamma$, and hence $[\![T]\!]^\gamma$ is satisfiable. From this contradiction, we infer that the concrete trace T is never reconsidered.

For the fixed Rational program, the CAR algorithm requires just two iterations. The first iteration computes $[\![P]\!]^\perp$, which yields the trace of Figure 2, and the ETC algorithm yields the abstraction δ for this trace. On the second iteration, the CAR algorithm computes $[\![P]\!]^\delta$, yielding the boolean query of Figure 1(c). This boolean query is unsatisfiable, and hence the algorithm concludes that the fixed Rational program is correct. In general, the CAR algorithm may require multiple iterations to infer all the interface predicates and explicated clauses necessary to verify the program.

6 An Explicating Theory for Constraints

This section presents the ETC algorithm (explicating theory for constraints) for determining the satisfiability of traces. Given a trace T, the ETC algorithm decides if $T \models_{\mathcal{D}} r(\,)$ is satisfiable. In addition, if $T \models_{\mathcal{D}} r(\,)$ is unsatisfiable, the algorithm returns an abstraction γ such that $[\![T]\!]^\gamma \models_{\mathcal{B}} r(\,)$ is unsatisfiable, *i.e.*, γ explicates (at the boolean level) why T is unsatisfiable.

The ETC proceeds by first flattening the given query into a constraint. Since a particular variable name x may appear multiple times in a trace, during the flattening process we label variables with relation symbols, as in x^r. The operation t^r applies the relation symbol r to each variable in a term t. For example, $(f(x, y))^r \equiv f(x^r, y^r)$. When flattening a trace, each relation symbol is used to annotate variables mentioned within that definition, thus avoiding collision between distinct variables. The operation $\mathit{flat}_r^T(c)$ flattens a constraint c appearing within trace T, where the relation symbol r is applied to each variable in c:

$$\begin{array}{rcl}
\mathit{flat}_r^T(s(t_1,\ldots,t_n)) & = & (\ \wedge_i\ x_i^s = t_i^r) \wedge \mathit{flat}_s^T(e) \\
 & & \text{if } T \text{ contains } s(x_1,\ldots,x_n) \text{ :- } e \\
\mathit{flat}_r^T(p(t_1,\ldots,t_n)) & = & p(t_1^r,\ldots,t_n^r) \\
\mathit{flat}_r^T(\neg p(t_1,\ldots,t_n)) & = & \neg p(t_1^r,\ldots,t_n^r) \\
\mathit{flat}_r^T(e_1 \wedge e_2) & = & \mathit{flat}_r^T(e_1) \wedge \mathit{flat}_r^T(e_2)
\end{array}$$

The flattened constraint is equi-satisfiable to the trace that produced it.

Lemma 7 *For any trace T, $T \models_{\mathcal{D}} r(\,)$ if and only if $\models_{\mathcal{D}} \mathit{flat}_r^T(r())$.*

The ETC algorithm relies on a proof-generating decision procedure for $\mathcal{D}$. As mentioned earlier, for our intended application of software model checking, the domain $\mathcal{D}$ should include equality with uninterpreted function symbols (EUF), linear arithmetic, and functional maps (via the *select* and *store* functions). Several proof-generating decision procedures or theorem provers for this domain has been developed using the Nelson-Oppen framework of cooperating decision procedures for individual theories [21]; examples include Verifun [10], CVC [2] and Touchstone [20]. Given a constraint c, the proof-generating decision procedure either reports that the constraint is satisfiable, or generates a proof of its unsatisfiability. We illustrate this idea with the following proof that the trace of Figure 2 is unsatisfiable. In this proof, we abbreviate $d^{\mathbf{E}\mathrm{trunc}^5}$ by d^5, etc.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathit{select}(d^5,\mathit{this}^5)=0 \quad d^5=d^4 \quad \mathit{this}^5=\mathit{this}^4}{\mathit{select}(d^4,\mathit{this}^4)=0} \quad d^4=d' \quad \mathit{this}^4=\mathit{this}}{\mathit{select}(d',\mathit{this})=0} \qquad \dfrac{d'^3=\mathit{store}(d^3,\mathit{this}^3,y^3) \quad d'^3=d' \quad d^3=d \quad \mathit{this}^3=\mathit{this} \quad y^3=y}{d'=\mathit{store}(d,\mathit{this},y)}}{y=0} \qquad y\neq 0}{\mathit{false}}$$

To simplify our development, instead of deadling with such proof trees, we assume the proof is represented as a conjunction of clauses $d_1 \wedge \ldots \wedge d_n$, where each clause is a tautology with respect to $\mathcal{D}$, and where the unsatisfiability of $c \wedge d_1 \wedge \ldots \wedge d_n$ follows by purely propositional reasoning. Thus, we represent the above proof as the conjunction of the following clauses, where each clause is written as an implication, for clarity:

$$\begin{array}{l}
(\mathit{select}(d^5,\mathit{this}^5)=0 \wedge d^5=d^4 \wedge \mathit{this}^5=\mathit{this}^4) \Rightarrow \mathit{select}(d^4,\mathit{this}^4)=0 \\
(\mathit{select}(d^4,\mathit{this}^4)=0 \wedge d^4=d' \wedge \mathit{this}^4=\mathit{this}) \Rightarrow \mathit{select}(d',\mathit{this})=0 \\
(d'^3=\mathit{store}(d^3,\mathit{this}^3,y^3) \wedge d'^3=d' \wedge d^3=d \wedge \mathit{this}^3=\mathit{this} \wedge y^3=y) \\
\qquad \Rightarrow d'=\mathit{store}(d,\mathit{this},y) \\
(\mathit{select}(d',\mathit{this})=0 \wedge d'=\mathit{store}(d,\mathit{this},y)) \Rightarrow y=0
\end{array}$$

We require in addition that the proof satisfy certain *hygiene* conditions. For each rule $r(\vec{x})$:- e in the rule set, let $\mathit{args}(r)=\vec{x}$ and $\mathit{vars}(r)=\mathit{vars}(e)$. A clause is *hygenic* if there exists some relation r such that for each literal l in the clause, either:

1. $\mathit{vars}(l) \subseteq \mathit{vars}(r)$,

2. l is a *variable binding equality* $x_i^s = t_i^r$ for some callee s of r, or

3. $vars(l) \subseteq args(s)$ for some callee s of r.

The above proof satisfies this hygine requirement.

We compute the desired abstraction γ from the proof, starting with $\gamma = \perp$ and processing each clause in the proof in turn. For each clause d_i, let d_i be hygienic with respect to r. We first remove from d_i each variable binding equality. We next consider in turn each remaining literal l in d_i. If $vars(l) \subseteq args(s)$ where r includes a call $s(t_1, \ldots, t_n)$ and $args(s) = x_1, \ldots, x_n$, then the literal l expresses a property of the call of s that is relevant in proving that the current trace is unsatisfiable. Hence we record the positive form of l as an interface predicate by adding it to γ_s^a. In addition, we replace l in d_i by $l[x_i := t_1]$. After these operations, the modified clause d_i satisfies $vars(d_i) \subseteq vars(r)$, and we add d_i to γ_r^e (unless d_i is trivially true via propositional reasoning).

Lemma 8 *The computed abstraction* γ *is such that* $[\![T]\!]^\gamma \not\models_{\mathcal{B}} r$.

When applied to the above proof, this process yields the abstraction δ (shown in Section 4), which is precise enough to refute the Rational trace of Figure 2.

For the fixed Rational program, the CAR algorithm requires just two iterations. The first iteration computes $[\![P]\!]^\perp$, which yields the trace of Figure 2, and the ETC algorithm yields the abstraction δ for this trace. On the second iteration, the CAR algorithm computes $[\![P]\!]^\delta$, yielding the boolean query of Figure 1, column 3. This boolean query is unsatisfiable, and hence the algorithm concludes that the fixed Rational program is correct. In general, the CAR algorithm may require multiple iterations to infer all the interface predicates and explicated clauses necessary to verify the program.

7 Related Work

This paper is a synthesis of ideas from extended static checking [7, 11], software model checking [1, 15], and explicating theorem provers [10, 2]. An extended static checker translates the given program into a logical formula. However, the translation of (recursive) procedure calls requires programmer-supplied specifications. We build on top of the ESC approach, but avoid the need for procedure specifications by targeting *constraint logic*, which can express recursion directly.

The software model checkers SLAM [1] and BLAST [15] use a combination of predicate abstraction [14] and automatic predicate inference. This paper shows that these ideas also apply in a natural and efficient manner to constraint logic, which provides a natural means for expressing correctness properties of the imperative programs.

Lahiri *et al* present an alternative approach for avoiding many theorem proving queries during predicate abstraction [17]. Instead, they formulate a symbolic representation of predicate abstraction step, reduce it to be quantified Boolean formula, and use Boolean reasoning to extract the abstract transition relation. Their experimental results are quite promising. McMillan and Amla present a technique for automatic abstraction of finite state systems based on a proof of unsatisfiability for all traces up to a given bound [19]. In contrast, our approach performs abstraction refinement based on single traces of infinite state systems. Developing a synthesis of these approaches is an interesting area for future work.

The explicating theorem provers Verifun [10] and CVC [2] iteratively conjoin the given query with explicated axiom instantiations until the conjoined query is unsatisfiable by boolean reasoning. This paper adapts these ideas to constraint logic to infer explicated clauses, thus avoiding the need for a exponential number of queries to the decision procedures.

The depth-first search of standard constraint logic implementations [23] corresponds to explicit path exploration, much like that performed by software model checkers, such as Bandera [8]. However, whereas Bandera relies on programmer-supplied abstractions to abstract (infinite-state) data variables, the constraint logic implementation reasons about data values using collections of constraints, thus providing a form of automatic data abstraction. The programmer-supplied abstractions of Bandera do provide stronger termination guarantees, but may yield false alarms. Delzanno and Podelski [6] also explore the use of constraint logic for model checking. They focus on concurrent systems expressed in the guarded-command specification language proposed by Shankar [22], which does not provide explicit support for dynamic allocation or recursion. The performance of their constraint logic-based model checking approach is promising.

Bruening [3] has built a dynamic assertion checker based on state-space exploration for multithreaded Java programs. Stoller [24] provides a generalization of Bruening's method to allow model checking of programs with either message-passing or shared-memory communication. Both of these approaches operate on the concrete program without any abstraction. Abstract interpretation [4] is the standard framework for developing and describing program analyses, and provides the semantics basis for the abstractions in our work.

8 Conclusion

This paper proposes model checking infinite state, imperative software via translation to constraint logic queries. The translation into constraint logic is described in a previous paper [9]; this paper focuses on determining the satisfiability of the generated queries. Standard depth-first search techniques are inadequate, since realistic software typically admits infinitely many execution paths. This paper presents the CAR algorithm for deciding the satisfiability of the generated queries via a combination of predicate abstraction, iterative abstraction refinement, and proof-based explication. While more practical experience is needed, our prototype implementation of this algorithm has performed well on the number of small example programs, and has better termination properties than standard depth-first search.

An important direction for future work is applying the CAR algorithm to model checking multithreaded software systems. For such systems, techniques such as thread-modular reasoning are necessary to combat state explosion. Thread-modular reasoning is naturally expressed as a (possibly infinite state) least fixed point computation [12], which is turn easily expressed as a constraint logic query. Given such a query, our CAR algorithm can potentially infer both the reachable states and the environment assumptions and guarantees for each thread, using iterative abstraction refinement an unified manner to infer all of these relations. Similarly, reduction-based model checking can be also formulated as a least fixed point constraint logic query [13], again, potentially allowing the CAR algorithm to infer both the access predicates of program variables and the reachable states of the reduced program.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, May 2001.

[2] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer, July 2002.

[3] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analyses of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[5] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 51–58, June 2001.

[6] G. Delzanno and A. Podelski. Model checking in CLP. *Lecture Notes in Computer Science*, 1579:223–239, 1999.

[7] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Dec. 1998.

[8] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[9] C. Flanagan. Automatic software model checking using CLP. In *European Symposium on Programming*, 2003.

[10] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *Computer Aided Verification, 15th International Conference, CAV '03*, 2003.

[11] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[12] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN Workshop*, 2003.

[13] C. Flanagan and S. Qadeer. Transations: A new approach to the state-explosion problem for software model checking. In *Submission to the Software Model Checking Workshop*, 2003.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs via PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[15] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, January 2001.

[16] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[17] S. K. Lahiri, R. E. Bryand, and B. Cook. A symbolic approach to predicate abstraction. In *Proceedings of the 15th International Conference on Computer Aided Verification*, 2003.

[18] L. Lamport. How to write a long formula. Technical Note 1993-119, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Dec. 1993.

[19] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[20] G. C. Necula and P. Lee. Proof generation in the Touchstone theorem prover. In *Proceedings 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 25–44. Springer, June 2000.

[21] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Syst.*, 1(2):245–257, Oct. 1979.

[22] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *Computing Surveys*, 25(3):225–302, 1993.

[23] SICStus Prolog. On the web at `http://www.sics.se/sicstus/`.

[24] S. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, pages 224–244. Springer-Verlag, 2000.