ARTICLE IN PRESS

Available at www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Science of Computer Programming ▮▮▮ (▮▮▮▮) ▮▮▮–▮▮▮

Science of Computer Programming

www.elsevier.com/locate/scico

Automatic software model checking via constraint logic

Cormac Flanagan

Computer Science Department, University of California, Santa Cruz, CA, USA

Received 4 July 2003; received in revised form 11 December 2003; accepted 18 December 2003

Abstract

This paper proposes the use of constraint logic to perform model checking of imperative, infinite-state programs. We present a semantics-preserving translation from an imperative language with recursive procedures and heap-allocated mutable data structures into constraint logic. The constraint logic formulation provides a clean way to reason about the behavior and correctness of the original program. In addition, it enables the use of existing constraint logic implementations to perform bounded software model checking, using a combination of symbolic reasoning and explicit path exploration.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Model checking; Constraint logic; Program verification

1. Introduction

Ensuring the reliability of software systems is an important but challenging problem. Achieving reliability through testing alone is difficult, due to the test coverage problem. For finite-state systems, model checking techniques that explore all paths have been very successful. However, verifying software systems is a harder problem because such systems are inherently infinite-state: many variables are (essentially) infinite-domain and the heap is of unbounded size.

A natural method for describing and reasoning about infinite-state systems is to use *constraints*. For example, the constraint $a[i] > y$ describes states in which the ith component of a is greater than y. The close connection between constraints and program semantics is illustrated by Dijkstra's weakest precondition translation [10]. This translation expresses the behavior of a code fragment that does not use iteration or recursion

E-mail address: cormac@cs.ucsc.edu (C. Flanagan).

0167-6423/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.01.006

as a boolean combination of constraints. Fully automatic theorem provers, such as Simplify [9], provide an efficient means for reasoning about the validity of such combinations of constraints. These techniques provide the foundation of the extended static checkers ESC/Modula-3 [8] and ESC/Java [15].

Unfortunately, iterative and recursive constructs, such as `while` loops, `for` loops, and recursive procedure calls, cannot be directly translated into boolean combinations of constraints. Instead, extended static checkers rely on the programmer to supply loop invariants and procedure specifications to aid in this translation.[1] The need for invariants and specifications places a significant burden on programmer and is perhaps the main reason these checkers are not more widely used, even though they catch defects and improve software quality [15].

This paper presents a variant of the extended static checking approach that avoids the need for programmer-supplied invariants and specifications. Instead, we start with an unannotated program, which may include iterative and recursive constructs as well as assertions. We translate this program into in an extended logic called *constraint logic*. Essentially, a constraint logic rule set [20–22,24] consists of a sequence of rules, each of which defines a particular relation symbol as a boolean combination of constraints. Since constraints may refer to relation symbols, these rules can be self- and mutually recursive. By expressing iterative and recursive constructs of the original imperative program as recursive constraint logic rules, we avoid the need for programmer-supplied invariants and specifications.

This paper presents a semantics-preserving translation into constraint logic from an imperative language that is infinite-state and that supports global and local variables, heap-allocated mutable data structures, and recursive procedure calls. We use this translation to illustrate the connection between imperative programs and constraint logic rule sets, between erroneous program executions and satisfiable constraint logic queries, and between erroneous program traces and satisfiable constraint logic derivations.

Our translation enables the use of efficient constraint logic implementations, such as SICStus Prolog [28], to check correctness properties of software. This implementation performs a depth-first search for a satisfying assignment, using efficient constraint solvers to symbolically reason about boolean variables, linear arithmetic, and functional maps. This search strategy corresponds to *explicitly* exploring all program execution paths, but *symbolically* reasoning about data values. That is, instead of explicitly enumerating all possible values for an integer variable x, the constraint logic implementation symbolically reasons about the consistency of a collection of constraints or linear inequalities on x. This symbolic analysis provides greater coverage and efficient checking.

The depth-first search strategy may diverge on software with infinitely long or infinitely many execution paths. To cope with such systems, we can bound the depth of the search, thus producing a bounded software model checker. Our translation also

[1] These extended static checkers also support loops without invariants, which are handled in a manner that is unsound but still useful.

facilitates software model checking using other constraint logic implementation techniques, such as breadth-first search, tableaux methods, or subsumption, which may provide stronger termination and error detection properties.

The remainder of the paper proceeds as follows. Section 2 provides a review of constraint logic. Section 3 illustrates our translation by applying it to an example program, and uses the constraint logic representation to detect defects in the program. Section 4 presents the imperative language that is the basis for our formal development, and Section 5 translates this language into constraint logic. Section 6 uses the constraint logic representation for program checking and defect detection. Section 7 discusses related work, and we conclude in Section 8.

2. A review of constraint logic

In this section, we provide a brief review of constraint logic. We write $\vec{X}$ for a (possibly empty) sequence $X_1, \ldots, X_n$. A *term* t is either a variable or the application of a primitive function f to a sequence of terms. An *atom* $r(\vec{t})$ is the application of a user-defined relation r to a term sequence $\vec{t}$. A *primitive constraint* $p(\vec{t})$ is the application of a primitive predicate p to a term sequence. *Constraints* include primitive constraints and their negations, conjunction, disjunction, and atoms. A *rule* $r(\vec{t}) :\!- c$ provides a definition of the relational symbol r. For example, the rule $r(x, y) :\!- x = y$ defines r as the identity relation. Following Lamport [23], we sometimes use bullet-style notation for $\wedge$ and $\vee$ in large formulas for clarity.

Constraint logic syntax

(*terms*)	$t ::= x \mid f(\vec{t})$
(*constraints*)	$c ::= p(\vec{t}) \mid \neg p(\vec{t}) \mid c \wedge c \mid c \vee c \mid r(\vec{t})$
(*rules*)	$R ::= r(\vec{t}) :\!- c$
(*variables*)	x
(*constants*)	$k \in \{0, 1, 2, \ldots\}$
(*primitive fns*)	$f \in \{k, +, -, \texttt{select}, \texttt{store}\}$
(*primitive preds*)	$p \in \{\mathit{true}, \mathit{false}, =, \neq, <, \ldots\}$
(*relation names*)	r

Primitive functions include binary functions for addition and subtraction, nullary constants, and the `select` and `store` operations, which are explained in Section 5. Primitive predicates include equality, disequality, inequalities, and the nullary predicates *true* and *false*. We sometimes write binary function and predicate applications using infix instead of prefix notation.

A constraint logic rule set $\vec{R}$ is a collection of rules. These rules may be self- or mutually recursive, and so the rule set $\vec{R}$ may yield multiple models. We are only interested in the least model of $\vec{R}$ that is compatible with the intended interpretation $\mathscr{D}$ of the primitive functions and predicates. In particular, we are interested in the question of whether this least $\mathscr{D}$-compatible model of $\vec{R}$ implies a particular *goal* or atom $r(\vec{t})$, which we write as $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, r(\vec{t}\,)$, where the symbol $\tilde{\exists}$ existentially quantifies over all free variables in $r(\vec{t}\,)$.

Much work on the implementation and optimization of constraint logic queries has focused on answering such queries efficiently. In the following section, we exploit this effort to check correctness properties of an example program, without the need for procedure specifications or loop invariants.

3. Overview

To illustrate our method, consider the example program shown in Fig. 1, column 1, which is a variant of the locking example used to illustrate the BLAST checker [19]. The procedures `lock` and `unlock` acquire and release the lock l, respectively, where $l = 1$ if the lock is held, and $l = 0$ if the lock is not held. The correctness properties

Program	Transfer relations	Error relations

`lock() {`
`    assert l = 0;`
`    l := 1;`
`}`
	$\mathbf{T}_{\text{lock}}(l,n,d,l_1,n,d)$:–
	$\wedge\, l = 0$
	$\wedge\, l_1 = 1$
		$\mathbf{E}_{\text{lock}}(l,n,d)$:–
		$l \neq 0$

`unlock() {`
`    assert l = 1;`
`    l := 0;`
`}`
	$\mathbf{T}_{\text{unlock}}(l,n,d,l_1,n,d)$:–
	$\wedge\, l = 1$
	$\wedge\, l_1 = 0$
		$\mathbf{E}_{\text{unlock}}(l,n,d)$:–
		$l \neq 1$

`main() {`
`    loop();`
`    unlock();`
`}`
	$\mathbf{T}_{\text{main}}(l,n,d,l_2,n_2,d_2)$:–
	$\wedge\, \mathbf{T}_{\text{loop}}(l,n,d,l_1,n_1,d_1)$
	$\wedge\, \mathbf{T}_{\text{unlock}}(l_1,n_1,d_1,l_2,n_2,d_2)$
		$\mathbf{E}_{\text{main}}(l,n,d)$:–
		$\vee\, \mathbf{E}_{\text{loop}}(l,n,d)$
		$\vee \wedge\, \mathbf{T}_{\text{loop}}(l,n,d,l_1,n_1,d_1)$
		$\quad \wedge\, \mathbf{E}_{\text{unlock}}(l_1,n_1,d_1)$

`loop() {`
`    lock();`
`    d := n;`
`    unl();`
`    if (n != d) {`
`        loop();`
`    } else {`
`        // skip`
`    }`
`}`
	$\mathbf{T}_{\text{loop}}(l,n,d,l_4,n_4,d_4)$:–
	$\wedge\, \mathbf{T}_{\text{lock}}(l,n,d,l_1,n_1,d_1)$
	$\wedge\, d_2 = n_1$
	$\wedge\, \mathbf{T}_{\text{unl}}(l_1,n_1,d_2,l_3,n_3,d_3)$
	$\wedge \vee \wedge\, n_3 \neq d_3$
	$\qquad \wedge\, \mathbf{T}_{\text{loop}}(l_3,n_3,d_3,l_4,n_4,d_4)$
	$\quad \vee \wedge\, n_3 = d_3$
	$\qquad \wedge\, l_4 = l_3$
	$\qquad \wedge\, n_4 = n_3$
	$\qquad \wedge\, d_4 = d_3$
		$\mathbf{E}_{\text{loop}}(l,n,d)$:–
		$\vee\, \mathbf{E}_{\text{lock}}(l,n,d)$
		$\vee \wedge\, \mathbf{T}_{\text{lock}}(l,n,d,l_1,n_1,d_1)$
		$\quad \wedge\, d_2 = n_1$
		$\quad \wedge \vee\, \mathbf{E}_{\text{unl}}(l_1,n_1,d_2)$
		$\qquad \vee \wedge\, \mathbf{T}_{\text{unl}}(l_1,n_1,d_2,l_3,n_3,d_3)$
		$\qquad\quad \wedge\, n_3 \neq d_3$
		$\qquad\quad \wedge\, \mathbf{E}_{\text{loop}}(l_3,n_3,d_3)$

`unl() {`
`    if (*) {`
`        unlock();`
`        // n++;`
`    }`
`}`
	$\mathbf{T}_{\text{unl}}(l,n,d,l_1,n_1,d_1)$:–
	$\vee\, \mathbf{T}_{\text{unlock}}(l,n,d,l_1,n_1,d_1)$
	$\vee \wedge\, l_1 = l$
	$\quad \wedge\, n_1 = n$
	$\quad \wedge\, d_1 = d$
		$\mathbf{E}_{\text{unl}}(l,n,d)$:–
		$\mathbf{E}_{\text{unlock}}(l,n,d)$

Fig. 1. The example program and the corresponding error and transfer relations.

we wish to check are that:

(1) the procedure `lock` is never called when the lock is already held, and
(2) the procedure `unlock` is never called unless the lock is already held.

These correctness properties are expressed as assertions in the `lock` and `unlock` procedures. Hence, checking these properties reduces to checking whether the example program *goes wrong* by violating either of these assertions.

The example contains three other routines, which manipulate two additional variables, n and d. Thus, the state of the store is captured by the triple $\langle l, n, d\rangle$. The example uses the notation `if (*) ...` to express nondeterministic choice.

Our method translates each procedure m into two constraint logic relations:

(1) the *error relation* $\mathbf{E}_m(l, n, d)$, which describes pre-states $\langle l, n, d\rangle$ from which the execution of m goes wrong by failing an assertion; and
(2) the *transfer relation* $\mathbf{T}_m(l, n, d, l', n', d')$, which describes the relation between pre-states $\langle l, n, d\rangle$ and post-states $\langle l', n', d'\rangle$ of executions of m that terminate normally (without failing an assertion).

The transfer and error relations for the example program are shown in Fig. 1, columns 2 and 3, respectively. The relation $\mathbf{E}_{\text{lock}}$ says that `lock` goes wrong if l is not initially 0, and $\mathbf{T}_{\text{lock}}$ says that `lock` terminates normally if l is initially 0, where $l = 1$ and n and d are unchanged the post-state. The relation $\mathbf{E}_{\text{main}}$ says that `main` goes wrong if either `loop` goes wrong or `loop` terminates normally and `unlock` goes wrong in the post-state of `loop`. The definitions of the other relations are similarly intuitive. Automatically generating these definitions from the program source code is straightforward.

We use these relation definitions to check if an invocation of `main` may go wrong by checking the satisfiability of the constraint logic query $\mathbf{E}_{\text{main}}(l, n, d)$. This query is satisfiable in the case where $l = 1$, indicating that the program may go wrong if the lock is held initially; an inspection of the source code reveals that this is indeed the case.

If we provide the additional precondition that the lock is not initially held, then the corresponding constraint logic query $l = 0 \wedge \mathbf{E}_{\text{main}}(l, n, d)$ is still satisfiable via the following derivation:

```
$\mathbf{E}_{\text{main}}(0, n, d)$
  $\mathbf{T}_{\text{loop}}(0, n, d, 0, n, d)$
    $\mathbf{T}_{\text{lock}}(0, n, d, 1, n, d)$
    $\mathbf{T}_{\text{unl}}(1, n, d, 0, n, d)$
      $\mathbf{T}_{\text{unlock}}(1, n, d, 0, n, d)$
  $\mathbf{E}_{\text{unlock}}(0, n, d)$
```

This derivation corresponds to the following execution trace: `main` calls `loop`; `loop` calls `lock`; `lock` returns to `loop`; `loop` calls `unl`; `unl` calls `unlock`; `unlock` returns to `unl`; `unl` returns to `loop`; `loop` returns to `main`; `main` calls `unlock`; and `unlock` fails its assertion, since there are two calls to `unlock` without an intervening call to `lock`.

The reason for this bug is that the increment operation `n++` in `unl` is commented out. After uncommenting this increment operation, the modified transfer relation for `unl` is:

$$
\begin{array}{l}
\mathbf{T}_{\mathrm{unl}}(l,n,d,l_1,n_2,d_1) :- \\
\qquad\qquad \vee \wedge \mathbf{T}_{\mathrm{unlock}}(l,n,d,l_1,n_1,d_1) \\
\qquad\qquad \quad \wedge\, n_2 = n_1 + 1 \\
\qquad\qquad \vee \wedge\, l_1 = l \\
\qquad\qquad \quad \wedge\, n_2 = n \\
\qquad\qquad \quad \wedge\, d_1 = d
\end{array}
$$

The query $l=0 \wedge \mathbf{E}_{\mathrm{main}}(l,n,d)$ is now unsatisfiable, indicating that the fixed program satisfies the desired correctness properties.

4. The source language: syntax and semantics

We now formalize the approach outlined above, and begin by presenting the syntax and semantics of the imperative language that we use as the basis for our formal development.

4.1. Syntax

A program is a sequence of procedure definitions. Each procedure definition consists of a procedure name m and a sequence of formal parameters, which are bound in the procedure body, and can be α-renamed in the usual fashion. The procedure body is an expression. Expressions include variable reference and assignment, `let`-expressions, application of primitive functions f and user-defined procedures m, conditionals, and assertions. To illustrate the handling of heap-allocated data structures, the language includes mutable pairs, and provides operations to create pairs and to access and update each field i of a pair, for $i=1,2$. Although our language does not include iterative constructs such as `while` or `for` loops, they can be encoded as tail-recursive procedures. In addition to local variables bound by `let`-expressions and parameter lists, programs may also manipulate the global variables $\vec{g}$. For simplicity, the language is untyped, although we syntactically distinguish boolean expressions, which are formed by applying a primitive predicate p to an argument sequence.

Programming language syntax

$$
\begin{array}{lll}
(\textit{programs}) & P & ::= \vec{D} \\
(\textit{definitions}) & D & ::= m(\vec{x}\,)\ \{e\} \\
(\textit{expressions}) & e & ::= x \mid x := e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \quad \mid\ f(\vec{e}\,) \mid m(\vec{e}\,) \mid \texttt{if}\ p(\vec{e}\,)\ e\ e \mid \texttt{assert}\ p(\vec{e}\,) \\
 & & \quad \mid\ \langle e,e\rangle \mid e.i \mid e.i := e \\
(\textit{procedure names}) & m & \\
(\textit{global variables}) & \vec{g} & \\
(\textit{special variables}) & \vec{h} & = \ h.h_1.h_2
\end{array}
$$

Throughout this paper, we assume the original program and the desired correctness property have already been combined into an *instrumented program*, which includes `assert` statements that check that the correctness property is respected by the program. We say an execution of the instrumented program *goes wrong* if it fails an assertion because the original program fails the desired correctness property. The focus of this paper is to statically determine if the instrumented program can go wrong.

Notation. We use $\vec{X} \cdot \vec{Y}$ to denote sequence concatenation and ε to denote the empty sequence. We sometimes interpret sequences as sets, and vice versa. If M is a (partial) map, then the map $M[X := Y]$ maps X to Y and is otherwise identical to M, and the map $M[-X]$ is undefined on X and is otherwise identical to M. The operations $M[\vec{X} := \vec{Y}]$ and $M[-\vec{X}]$ are defined analogously. We use $\vec{X} = \vec{Y}$ to abbreviate $X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n$. We use e_1 ; e_2 to abbreviate `let` $x = e_1$ `in` e_2, where x is not free in e_2. We sometimes enclose expressions in parentheses for clarity.

4.2. Semantics

We formalize the meaning of programs using a “big step” operation semantics. A store σ is a partial mapping from variables to values. The judgment $e, \sigma \xrightarrow{P} v, \sigma'$ states that, when started from a store σ, the evaluation of expression e may terminate normally yielding a result value v and resulting store σ'. In this judgment, program P provides the definitions of procedures that may be called by e. The related judgment $\vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma'$ extends this semantics to expression sequences, whose evaluation yields a sequence of values.

The rules defining these judgments are shown in Fig. 2. The rule [EVAL VAR] retrieves the value of a variable x from the store. The rule [EVAL ASSIGN] for $x := e$ first evaluates e to yield a new store σ' and a value v, and then produces an updated store $\sigma'[x := v]$ that records v as the current value for x. The rule [EVAL FN] relies on the functions $\mathscr{M}_f : Value^* \to Value$ to define the meaning of each primitive function f. The rules [EVAL IF] and [EVAL ASSERT] rely on the relations $\mathscr{M}_p \subseteq Value^*$ to define the meaning of each primitive predicate p. The rule [EVAL CALL] for a procedure call $m(\vec{e})$ ensures that the formal parameters are not already bound in the store (using implicit α-renaming, if necessary), evaluates the actual parameter list $\vec{e}$, updates the store to map the formal parameters to the resulting argument values, and then evaluates the procedure body in the extended store.

To represent pairs, the store σ also maps three special variables, h, h_1, and h_2, to maps. The map $\sigma(h)$ describes which locations have been allocated, and $\sigma(h_1)$ and $\sigma(h_2)$ describe the components of allocated pairs. For any heap location l, if $\sigma(h)(l) = 0$ then the location l is not allocated, otherwise the components of the pair at location l are given by $\sigma(h_1)(l)$ and $\sigma(h_2)(l)$, respectively. This representation of pairs significantly simplifies the correspondence proof between imperative programs and constraint logic rule sets. The rule [EVAL PAIR] for $\langle e_1, e_2 \rangle$ picks an unallocated location l with $\sigma(h)(l) = 0$, and updates the store to indicate that the location is allocated and to record the two fields values of the pair. The rules [EVAL FIELD REF] and [EVAL FIELD ASSIGN] retrieve and update a field of a pair, respectively.

$$\boxed{e, \sigma \xrightarrow{P} v, \sigma'}$$

[EVAL VAR]
$$\frac{}{x, \sigma \xrightarrow{P} \sigma(x), \sigma}$$

[EVAL ASSIGN]
$$\frac{e, \sigma \xrightarrow{P} v, \sigma'}{(x := e), \sigma \xrightarrow{P} v, \sigma'[x := v]}$$

[EVAL LET]
$$\frac{\begin{array}{c} e_1, \sigma \xrightarrow{P} v_1, \sigma' \\ x \notin dom(\sigma') \\ e_2, \sigma'[x := v_1] \xrightarrow{P} v_2, \sigma'' \end{array}}{(\texttt{let } x = e_1 \texttt{ in } e_2), \sigma \xrightarrow{P} v_2, \sigma''[-x]}$$

[EVAL FN]
$$\frac{\vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma'}{f(\vec{e}), \sigma \xrightarrow{P} \mathscr{M}_f(\vec{v}), \sigma'}$$

[EVAL CALL]
$$\frac{\begin{array}{c} \vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma' \\ m(\vec{x})\ \{e\} \in P \\ \vec{x} \cap dom(\sigma') = \emptyset \\ e, \sigma'[\vec{x} := \vec{v}] \xrightarrow{P} v, \sigma'' \end{array}}{m(\vec{e}), \sigma \xrightarrow{P} v, \sigma''[-\vec{x}]}$$

[EVAL IF]
$$\frac{\begin{array}{c} \vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma' \\ i = (\text{if } \mathscr{M}_p(\vec{v}) \text{ then } 1 \text{ else } 2) \\ e_i, \sigma' \xrightarrow{P} v, \sigma'' \end{array}}{(\texttt{if } p(\vec{e})\ e_1\ e_2), \sigma \xrightarrow{P} v, \sigma''}$$

[EVAL ASSERT]
$$\frac{\begin{array}{c} \vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma' \\ \mathscr{M}_p(\vec{v}) = true \end{array}}{(\texttt{assert } p(\vec{e})), \sigma \xrightarrow{P} 0, \sigma}$$

[EVAL PAIR]
$$\frac{\begin{array}{c} (e_1.e_2), \sigma \xrightarrow{P} (v_1.v_2), \sigma' \qquad \sigma'(h)(l) = 0 \\ \sigma'' = \sigma'[h := \sigma'(h)[l := 1], h_i := \sigma'(h_i)[l := v_i]^{i \in 1,2}] \end{array}}{\langle e_1, e_2 \rangle, \sigma \xrightarrow{P} l, \sigma''}$$

[EVAL FIELD REF]
$$\frac{e, \sigma \xrightarrow{P} l, \sigma'}{e.i, \sigma \xrightarrow{P} \sigma(h_i)(l), \sigma'}$$

[EVAL FIELD ASSIGN]
$$\frac{\begin{array}{c} (e_1.e_2), \sigma \xrightarrow{P} (v_1.v_2), \sigma' \\ \sigma'' = \sigma'[h_i := \sigma'(h_i)[v_1 := v_2]] \end{array}}{(e_1.i := e_2), \sigma \xrightarrow{P} v_2, \sigma''}$$

$$\boxed{\vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma'}$$

[EVAL NONE]
$$\frac{}{\varepsilon, \sigma \xrightarrow{P} \varepsilon, \sigma}$$

[eval some]
$$\frac{e, \sigma \xrightarrow{P} v, \sigma' \qquad \vec{e}, \sigma' \xrightarrow{P} \vec{v}, \sigma''}{(e.\vec{e}), \sigma \xrightarrow{P} (v.\vec{v}), \sigma''}$$

Fig. 2. Evaluation rules: normal semantics.

In addition to terminating normally, the evaluation of an expression may also go wrong by failing an assertion. The judgment $e, \sigma \xrightarrow{P} \texttt{wrong}$ states that, when started from a store σ, the evaluation of expression e may go wrong. Similarly, the judgment

$\vec{e}, \sigma \xrightarrow{P}$ `wrong` states that the evaluation of the expression sequence $\vec{e}$ from store σ may go wrong. The rules defining the error semantics of the language are shown in Fig. 3.

5. Translating imperative programs into constraint logic

We now describe the translation of imperative programs into constraint logic rule sets via symbolic forward execution. At each step in the translation, the environment Γ maps each program variable x into a constraint logic term that provides a symbolic representation of the value of x. Given the initial environment Γ for an expression e, the judgment

$$\Gamma \vdash e : \langle w, n\Gamma', t\rangle$$

describes the behavior of e. The *wrong condition* w is a constraint describing initial states from which e may go wrong by failing an assertion. For example, the wrong condition of $\texttt{assert}\ x = 0$ is $\Gamma(x) \neq 0$, i.e., the assertion goes wrong if x is not initially 0. Similarly, the *normal condition* n describes the initial states from which e may terminate normally. In this case, the environment Γ' describes values of variables in the post-state, and the term t is a symbolic representation of the result of e. The judgment $\Gamma \vdash \vec{e} : \langle w, n, \Gamma', \vec{t}\rangle$ behaves in a similar manner on expression sequences, which may go wrong or may terminate normally producing a value sequence represented by $\vec{t}$.

The rules defining these judgments are shown in Fig. 4. The rule [TR VAR] states that a variable access x never goes wrong and always terminates normally without changing the program state. The rule retrieves a symbolic representation $\Gamma(x)$ for the value of x from the environment. The rule [TR ASSIGN] for an assignment $x := e$ determines a symbolic representation t for e, and updates the environment to record that t represents of the current value of x. The rule [TR LET] states that $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ goes wrong if either e_1 goes wrong or if e_1 terminates normally and e_2 goes wrong.

Some translation rules are more complicated. For example, the rule [TR IF] for the conditional $\texttt{if}\ p(\vec{e}\,)\ e_1\ e_2$ needs to merge the environments Γ_i produced by the translation of e_i, for $i = 1, 2$. To accomplish this merge, the rule determines the set $\vec{y}$ of variables assigned in either e_1 or e_2, and introduces an environment Γ'' that maps $\vec{y}$ to fresh variables. Then, having determined that the branch e_i of the conditional is executed, the rule asserts that the $\Gamma''(\vec{y}\,) = \Gamma_i(\vec{y}\,)$, thus recording that the representation of $\vec{y}$ in the resulting environment Γ'' comes from the branch e_i. This translation of conditionals avoids the exponential blow-up of traditional VC generation algorithms [10], and is analogous to the compact VC generation algorithm of ESC/Java [16].

Our translation for pairs relies on the primitive functions `select` and `store`, where $\texttt{store}(a, i, v)$ extends a functional map a at index i with value v, and $\texttt{select}(a, i)$ selects the element at index i from map a. These two functions satisfy the select-of-store axioms:

$$\begin{aligned} & \texttt{select}(\texttt{store}(a,i,v),i) = v \\ i \neq j \;\Rightarrow\; & \texttt{select}(\texttt{store}(a,i,v),j) = \texttt{select}(a,j) \end{aligned}$$

$$\boxed{e, \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG ASSIGN]
$$\frac{e, \sigma \xrightarrow{P} \texttt{wrong}}{(x := e), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG FN]
$$\frac{\vec{e}, \sigma \xrightarrow{P} \texttt{wrong}}{f(\vec{e}), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG LET 1]
$$\frac{e_1, \sigma \xrightarrow{P} \texttt{wrong}}{(\texttt{let } x = e_1 \texttt{ in } e_2), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG LET 2]
$$\frac{\begin{array}{c} e_1, \sigma \xrightarrow{P} v_1, \sigma' \qquad x \notin dom(\sigma') \\ 1e_2, \sigma'[x := v_1] \xrightarrow{P} \texttt{wrong} \end{array}}{(\texttt{let } x = e_1 \texttt{ in } e_2), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG CALL 1]
$$\frac{\vec{e}, \sigma \xrightarrow{P} \texttt{wrong}}{m(\vec{e}), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG IF 1]
$$\frac{\vec{e}, \sigma \xrightarrow{P} \texttt{wrong}}{(\texttt{if } p(\vec{e})\ e_1\ e_2), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG CALL 2]
$$\frac{\begin{array}{c} \vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma' \\ m(\vec{x})\ \{e\} \in P \qquad \vec{x} \cap dom(\sigma') = \emptyset \\ e, \sigma'[\vec{x} := \vec{v}] \xrightarrow{P} \texttt{wrong} \end{array}}{m(\vec{e}), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG IF 2]
$$\frac{\begin{array}{c} \vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma' \\ i = (\text{if } \mathscr{M}_p(\vec{v}) \text{ then } 1 \text{ else } 2) \\ e_i, \sigma' \xrightarrow{P} \texttt{wrong} \end{array}}{(\texttt{if } p(\vec{e})\ e_1\ e_2), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG ASSERT 1]
$$\frac{\vec{e}, \sigma \xrightarrow{P} \texttt{wrong}}{(\texttt{assert } p(\vec{e})), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG ASSERT 2]
$$\frac{\vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma' \qquad \mathscr{M}_p(\vec{v}) = \mathit{false}}{(\texttt{assert } p(\vec{e})), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG PAIR]
$$\frac{(e_1.e_2), \sigma \xrightarrow{P} \texttt{wrong}}{\langle e_1, e_2 \rangle, \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG FIELD REF]
$$\frac{e, \sigma \xrightarrow{P} \texttt{wrong}}{e.i, \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG FIELD ASSIGN]
$$\frac{(e_1.e_2), \sigma \xrightarrow{P} \texttt{wrong}}{(e_1.i := e_2), \sigma \xrightarrow{P} \texttt{wrong}}$$

$$\boxed{\vec{e}, \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG FIRST]
$$\frac{e, \sigma \xrightarrow{P} \texttt{wrong}}{(e.\vec{e}\ ), \sigma \xrightarrow{P} \texttt{wrong}}$$

[WRONG REST]
$$\frac{e, \sigma \xrightarrow{P} v, \sigma' \qquad \vec{e}, \sigma' \xrightarrow{P} \texttt{wrong}}{(e.\vec{e}\ ), \sigma \xrightarrow{P} \texttt{wrong}}$$

Fig. 3. Evaluation rules: error semantics.

$$\boxed{\Gamma \vdash e \,:\, \langle w, n, \Gamma', t\rangle}$$

[TR VAR]

$$\frac{}{\Gamma \vdash x \,:\, \langle \mathit{false}, \mathit{true}, \Gamma, \Gamma(x)\rangle}$$

[TR ASSIGN]

$$\frac{\Gamma \vdash e \,:\, \langle w, n, \Gamma', t\rangle}{\Gamma \vdash x := e \,:\, \langle w, n, \Gamma'[x := t], t\rangle}$$

[TR LET]

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \,:\, \langle w_1, n_1, \Gamma_1, t_1\rangle \\ \Gamma_1[x := t_1] \vdash e_2 \,:\, \langle w_2, n_2, \Gamma_2, t_2\rangle\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \,:\, \langle w_1 \vee (n_1 \wedge w_2), n_1 \wedge n_2, \Gamma_2[-x], t_2\rangle}$$

[TR CALL]

$$\frac{\begin{array}{c}\Gamma \vdash \vec{e} \,:\, \langle w, n, \Gamma', \vec{t}\rangle \\ z, \vec{g'}, \vec{h'}\ \text{fresh} \\ w' \equiv w \vee (n \wedge E_m(\vec{t}, \Gamma'(\vec{g}), \Gamma'(\vec{h}))) \\ n' \equiv n \wedge T_m(\vec{t}, \Gamma'(\vec{g}), \Gamma'(\vec{h}), \vec{g'}, \vec{h'}, z) \\ \Gamma'' \equiv \Gamma'[\vec{g} := \vec{g'}, \vec{h} := \vec{h'}]\end{array}}{\Gamma \vdash m(\vec{e}\,) \,:\, \langle w', n', \Gamma'', z\rangle}$$

[TR IF]

$$\frac{\begin{array}{c}\Gamma \vdash \vec{e} \,:\, \langle w, n, \Gamma', \vec{t}\rangle \qquad \Gamma' \vdash e_i \,:\, \langle w_i, n_i, \Gamma_i, t_i\rangle \\ z\ \text{fresh} \qquad \vec{y} = \{y \mid \Gamma_1(y) \neq \Gamma_2(y)\} \\ \Gamma''(x) = (\text{if } x \in \vec{y}\ \text{then fresh var else } \Gamma_1(x)) \\ w' \equiv w \vee (n \wedge p(\vec{t}\,) \wedge w_1) \vee (n \wedge \neg p(\vec{t}\,) \wedge w_2) \\ n'_1 \equiv n \wedge\ \ p(\vec{t}\,) \wedge n_1 \wedge z = t_1 \wedge \Gamma''(\vec{y}\,) = \Gamma_1(\vec{y}\,) \\ n'_2 \equiv n \wedge \neg p(\vec{t}\,) \wedge n_2 \wedge z = t_2 \wedge \Gamma''(\vec{y}\,) = \Gamma_2(\vec{y}\,)\end{array}}{\Gamma \vdash \texttt{if}\ \ p(\vec{e}\,)\ e_1\ e_2 \,:\, \langle w', (n'_1 \vee n'_2), \Gamma'', z\rangle}$$

[TR ASSERT]

$$\frac{\Gamma \vdash \vec{e} \,:\, \langle w, n, \Gamma', \vec{t}\rangle}{\Gamma \vdash \texttt{assert}\ \ p(\vec{e}\,) \,:\, \langle w \vee (n \wedge \neg p(\vec{t}\,)), n \wedge\ p(\vec{t}\,), \Gamma', 0\rangle}$$

[TR FN]

$$\frac{\Gamma \vdash \vec{e} \,:\, \langle w, n, \Gamma', \vec{t}\rangle}{\Gamma \vdash f(\vec{e}\,) \,:\, \langle w, n, \Gamma', f(\vec{t}\,)\rangle}$$

[TR PAIR]

$$\frac{\begin{array}{c}\Gamma \vdash (e_1.e_2) \,:\, \langle w, n, \Gamma', (t_1.t_2)\rangle \\ \Gamma'' \equiv \Gamma'[h := \texttt{store}(\Gamma'(h), l, 1), h_i := \texttt{store}(\Gamma'(h_i), l, t_i)^{i \in 1,2}] \\ l\ \text{fresh} \qquad n' \equiv n \wedge \texttt{select}(\Gamma'(h), l) = 0\end{array}}{\Gamma \vdash \langle e_1, e_2\rangle \,:\, \langle w, n', \Gamma'', l\rangle}$$

[TR FIELD REF]

$$\frac{\Gamma \vdash e \,:\, \langle w, n, \Gamma', t\rangle}{\Gamma \vdash e.i \,:\, \langle w, n, \Gamma', \texttt{select}(\Gamma'(h_i), t)\rangle}$$

[TR FIELD ASSIGN]

$$\frac{\begin{array}{c}\Gamma \vdash e_1.e_2 \,:\, \langle w, n, \Gamma', t_1.t_2\rangle \\ \Gamma'' \equiv \Gamma'[h_i := \texttt{store}(\Gamma'(h_i), t_1, t_2)]\end{array}}{\Gamma \vdash e_1.i := e_2 \,:\, \langle w, n, \Gamma'', t_2\rangle}$$

$$\boxed{\Gamma \vdash \vec{e} \,:\, \langle w, n, \Gamma', \vec{t}\rangle}$$

[TR NONE]

$$\frac{}{\Gamma \vdash \varepsilon \,:\, \langle \mathit{false}, \mathit{true}, \Gamma, \varepsilon\rangle}$$

[TR SOME]

$$\frac{\Gamma \vdash e \,:\, \langle w, n, \Gamma', t\rangle \qquad \Gamma' \vdash \vec{e} \,:\, \langle w', n', \Gamma'', \vec{t}\rangle}{\Gamma \vdash (e.\vec{e}\,) \,:\, \langle w \vee (n \wedge w'), n \wedge n', \Gamma'', (t.\vec{t}\,)\rangle}$$

Fig. 4. Translation rules for expressions.

To aid in the translation, the environment Γ maps the special variables h, h_1, h_2 into constraint logic terms that symbolically model of the current state of the heap. The rule [TR PAIR] for the pair creation expression $\langle e_1, e_2\rangle$ introduces a fresh variable l and asserts that $\texttt{select}(\Gamma(h), l) = 0$, which means that the location l is not yet allocated.

$$\boxed{\vdash D \,:\, R^e, R^t}$$

[TR DEF]

$$\frac{\begin{array}{c} dom(\Gamma) = \{\vec{x}, \vec{g}, \vec{h}\} \\ rng(\Gamma) = \text{fresh variables} \\ \Gamma \vdash e \,:\, \langle w, n, \Gamma', t\rangle \end{array}}{\vdash m(\vec{x})\ \{e\} \,:\, \begin{array}{l} E_m(\Gamma(\vec{x}), \Gamma(\vec{g}), \Gamma(\vec{h})) :\!-\, w, \\ T_m(\Gamma(\vec{x}), \Gamma(\vec{g}), \Gamma(\vec{h}), \Gamma'(\vec{g}), \Gamma'(\vec{h}), t) :\!-\, n \end{array}}$$

$$\boxed{\vdash P \,:\, \vec{R}}$$

[TR DEFS]

$$\frac{P = D_1. \cdots .D_n \quad \vdash D_i \,:\, R^e_i, R^t_i}{\vdash P \,:\, R^e_1, \ldots, R^e_n, R^t_1, \ldots, R^t_n}$$

Fig. 5. Translation rules for procedure definitions.

The rule then updates the environment (1) to map h to $\mathtt{store}(\Gamma(h), l, 1)$, indicating that location l is now allocated, and (2) to map each h_i to $\mathtt{store}(\Gamma(h_i), l, t_i)$, where the term t_i represents the value of e_i, for $i = 1, 2$. Thus, the rule records the contents of the pair in the new terms for h_1 and h_2. The rules for accessing and updating pairs operate in a similar manner.

The most novel aspect of our translation concerns its handling of procedure calls. Earlier approaches translated procedure calls using user-supplied specifications. However, since writing specifications for all procedures imposes a significant burden on the programmer, we use a different approach that exploits the ability to define relation symbols recursively in constraint logic.

We translate each procedure definition $m(\vec{x})\ \{e\}$ into two constraint logic rules according to the rule [TR DEF] in Fig. 5. The first rule defines an *error relation* $\mathbf{E}_m$ that describes pre-states from which an invocation of m may go wrong; the second rule defines a *transfer relation* $\mathbf{T}_m$ that, in situations where m terminates normally, describes the pre-state/post-state relation of m. The arguments to the error relation E_m are the formal parameters $\vec{x}$, the global variables $\vec{g}$, plus the three special variables $\vec{h} = h \cdot h_1 \cdot h_2$ that model the heap. The arguments to T_m are again the formal parameters $\vec{x}$, the globals $\vec{g}$, the special variables $\vec{h}$, followed by $\vec{g'}$, which represents the post-state of the global variables, followed by $\vec{h'} = h' \cdot {h_1}' \cdot {h_2}'$, which represents the post heap state, followed by a term t representing the return value of m. The rule [TR CALL] for a procedure call $m(\vec{e})$ generates a wrong condition that uses E_m to express states from which the execution of $m(\vec{e})$ may go wrong, and generates a normal condition that uses T_m to describe how $m(\vec{e})$ may terminate normally.

Since a program is a sequence of procedure definitions, the rule [TR DEFS] translates a program into a sequence of constraint logic rules simply by combining the constraint logic rules for each procedure in the program.

To motivate our least fixpoint interpretation of constraint logic rule sets, consider following the self-recursive, divergent procedure:

```
void m() { m(); }
```

whose corresponding transfer relation is

$$\mathbf{T}_{\mathrm{m}}(\vec{g},\vec{h'},\vec{g'},\vec{h'},r) : -\mathbf{T}_{\mathrm{m}}(\vec{g},\vec{h'},\vec{g'},\vec{h'},r)$$

This relation definition is also self-recursive, and therefore yields different meanings under the least fixpoint and greatest fixpoint interpretations. Under our least fixpoint interpretation, this transfer relation is false for all argument tuples, correctly reflecting the fact that there are no terminating executions of the procedure `m()`.

5.1. Correctness of the translation

Given an imperative program P, we translate it into error and transfer relations $\vec{R}$ according to the translation rule $\vdash P : \vec{R}$. For any expression e, the judgment:

$$\Gamma \vdash e : \langle w, n, \Gamma' t\rangle$$

describes the behavior of that expression from any initial state σ that is compatible with Γ, i.e., where $dom(\Gamma) \subseteq dom(\sigma)$ and $\models_{\mathscr{D}} \tilde{\exists}\, (\sigma \sim \Gamma)$. Here, the symbol $\tilde{\exists}$ existentially quantifies over the free variables of $(\sigma \sim \Gamma)$, and we use the notation $(\sigma \sim \Gamma)$ to abbreviate $\bigwedge_{x \in dom(\Gamma)} \sigma(x) = \Gamma(x)$, where $\sigma(x)$ means the ground term representing the value $\sigma(x)$. The following theorem formalizes the connection between program evaluations and constraint logic queries.

Theorem 1. *Suppose* $\vdash P : \vec{R}$ *and* $\Gamma \vdash e : \langle w, n, \Gamma', t\rangle$ *and* $\models_{\mathscr{D}} \tilde{\exists}\, (\sigma \sim \Gamma)$.
(1) *If* $e, \sigma \xrightarrow{P} v, \sigma'$ *then* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, ((\sigma \sim \Gamma) \wedge n \wedge (\sigma' \sim \Gamma') \wedge v = t)$.
(2) *If* $e, \sigma \xrightarrow{P}$ `wrong` *then* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, ((\sigma \sim \Gamma) \wedge w)$.

The correctness of the above theorem follows from the following theorem, which introduces a stronger hypothesis to facilitate an inductive proof.

Theorem 2. *Let X be any constraint logic formula.*
(1) *Suppose* $\vdash P : \vec{R}$ *and* $\Gamma \vdash e : \langle w, n, \Gamma', t\rangle$ *and* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, (X \wedge (\sigma \sim \Gamma))$.
  (a) *If* $e, \sigma \xrightarrow{P} v, \sigma'$ *then* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, (X \wedge (\sigma \sim \Gamma) \wedge n \wedge (\sigma' \sim \Gamma') \wedge v = t)$.
  (b) *If* $e, \sigma \xrightarrow{P}$ `wrong` *then* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, (X \wedge (\sigma \sim \Gamma) \wedge w)$.
(2) *Suppose* $\vdash P : \vec{R}$ *and* $\Gamma \vdash \vec{e} : \langle w, n, \Gamma', \vec{t}\rangle$ *and* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, (X \wedge (\sigma \sim \Gamma))$.
  (a) *If* $\vec{e}, \sigma \xrightarrow{P} \vec{v}, \sigma'$ *then* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, (X \wedge (\sigma \sim \Gamma) \wedge n \wedge (\sigma' \sim \Gamma') \wedge \vec{v} = \vec{t})$.
  (b) *If* $\vec{e}, \sigma \xrightarrow{P}$ `wrong` *then* $\vec{R} \models_{\mathscr{D}} \tilde{\exists}\, (X \wedge (\sigma \sim \Gamma) \wedge w)$.

Proof. The proof of all four cases is by simultaneous structural induction on the derivation for the evaluation of e (or $\vec{e}$) and by case analysis on the first rule used in that derivation. □

Thus, to check if the initial procedure *main* of a program P goes wrong, we first generate the corresponding relations $\vec{R}$ according to the rule $\vdash P : \vec{R}$, and then consider the constraint logic query

$$\vec{R} \models_{\mathscr{D}} \models \tilde{\exists} E_{\text{main}}(\vec{g}, \vec{h}).$$

If this query is satisfiable, the constraint logic implementation returns a satisfying assignment for $\vec{g}$ and $\vec{h}$, describing initial values for the global variables and the heap that yields an erroneous execution. If the implementation also returns a constraint logic derivation, then this derivation corresponds in a fairly direct manner to a trace of this erroneous execution.

6. An application

We next consider the example program shown in Fig. 6, which, for clarity, is presented using Java syntax. This class implements rational numbers, where a rational is represented as a pair of integers for the numerator and denominator. The class contains a constructor for creating rationals and a method `trunc` for converting a rational to an integer. The example also contains a test harness, which reads in two integers, n and d, ensures that d is not zero, creates a corresponding rational, and then repeatedly prints out the truncation of the rational.

```
class Rational {

    int num, den;

    Rational(int n, int d) {
        num = n;
        den = d;
    }

    int trunc() {
        assert den != 0;
        return num/den;
    }

    public static void main() {
        int n = readInt(), d = readInt();
        if( d == 0 ) return;
        Rational r = new Rational(d,n);
        for(int i=0; i<10000; i++) {
            print( r.trunc() );
        }
    }
}
```

Fig. 6. The example program `Rational`.

We wish to check that a division-by-zero error never occurs. We express this correctness property as an assertion in the `trunc` method, and translate the instrumented program into constraint logic rules. The constraint logic query $E_{\texttt{main}}()$ 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 `Rational` constructor in the wrong order. Note that since both arguments are integers, Java's type system does not catch this error.

After fixing this bug, the query $E_{\texttt{main}}()$ is now unsatisfiable, indicating that a division-by-zero error cannot occur. However, the constraint logic implementation that we use, SICStus Prolog [28], requires several seconds to answer this query, since its depth-first search strategy explicitly iterates through the loop in `main` 10,000 times.

To avoid this inefficiency, we are currently developing a constraint logic implementation optimized towards software model checking. This implementation uses lazy predicate abstraction and counter-example driven abstraction refinement. Our prototype implementation determines the unsatisfiability of the `Rational` example in just two iterations. We are currently extending this implementation to handle more realistic benchmarks.

7. Related work

This paper can be viewed as a synthesis of ideas from extended static checking [8,15] and model checking [5,26,3,25]. An extended static checker translates the given program into a combination of constraints over program variables, and uses sophisticated decision procedures to reason about the validity of these constraints, thus performing a precise, goal-directed analysis. 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, in which we can express recursion directly.

The software checkers SLAM [1] and BLAST [19] use a combination of predicate abstraction [17] and automatic predicate inference to avoid false alarms and the need for programmer-supplied abstractions, though they may not terminate. These tools have been successfully applied to a number of device drivers. Both tools abstract the given imperative program to a finite-state boolean program, which is then model-checked. This paper suggests that constraint logic may also provide a suitable framework for such tools.

Delzanno and Podelski [7] explore the use of constraint logic for model checking, and the performance of their approach is promising. They focus on concurrent systems expressed in the guarded-command specification language proposed by Shankar [27], which does not provide explicit support for dynamic allocation or recursion. The connection between constraint logic rule sets and imperative programs has also been explored by Gotlieb et al. [18], although the main focus of their work has been to infer appropriate inputs to test the program's execution over certain control-flow paths.

The depth-first search of standard constraint logic implementations [28] corresponds to explicit path exploration, much like that performed by software model checkers such as Bandera [11]. 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. Other approaches based on model checking include [2,29,30].

De Moura et al. [6] explore efficient methods for bounded model checking of infinite-state transition systems based on lazy theorem proving. Since these transition systems are a subset of constraint logic rule sets, their approach may suggest efficient techniques for checking the constraint logic rule sets generated by our work.

Instead of avoiding the need for loop invariants and specifications, another approach is to infer such annotations automatically. The Houdini annotation inference system [14,13] re-uses ESC/Java as a subroutine in a generate-and-test approach to annotation inference. Daikon uses an empirical approach to find probable invariants [12].

Symbolic execution is the underlying technique of the successful bug-finding tool PREfix for C and C++ programs [4]. For each procedure in the given program, PREfix synthesizes a set of execution paths, called a *model*. Models are used to reason about calls, which makes the process somewhat modular, except that fixpoints of models are approximated iteratively for recursive and mutually recursive calls.

8. Conclusion

This paper explores the connection between two programming paradigms: the traditional imperative paradigm and the constraint logic programming paradigm. We express the correctness of imperative programs in terms of constraint logic satisfiability, based on a semantics-preserving translation from imperative programs to constraint logic rule sets. The constraint logic formulation provides a clean way to reason about the behavior and correctness of the original imperative program.

This connection has immediate practical applications: it enables us to use existing constraint logic implementations to check correctness properties of imperative programs. For depth-first constraint logic implementations, this approach yields an efficient method for bounded model checking of software, using a combination of symbolic reasoning for data values and explicit path exploration.

In addition, constraint logic is well-studied [20–22,24], and provides optimizations and implementation techniques such as tableaux methods and subsumption [24], which offer the promise of complete model checking on certain classes of infinite-state programs. More experience on practical examples is certainly necessary, and may provide insight and motivation to develop specialized constraint logic implementations optimized for software model checking.

Acknowledgements

This work was started at HP Systems Research Center; it was also partly supported by faculty research funds granted by the University of California, Santa Cruz.

References

[1] T. Ball, S.K. Rajamani, Automatically validating temporal safety properties of interfaces, in: Model Checking Software, 8th Internat. SPIN Workshop, Lecture Notes in Computer Science, vol. 2057, Springer, Berlin, 2001, pp. 103–122.

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

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inform. Comput. 98 (2) (1992) 142–170.

[4] W.R. Bush, J.D. Pincus, D.J. Sielaff, A static analyzer for finding dynamic programming errors, Software—Practice & Experience 30 (7) (2000) 775–802.

[5] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Workshop on Logic of Programs, Lecture Notes in Computer Science, vol. 131, Springer, Berlin, 1981, pp. 52–71.

[6] L. de Moura, H. Rueß, M. Sorea, Lazy theorem proving for bounded model checking over infinite domains, in: Proc. 18th Internat. Conf. Automated Deduction, Lecture Notes in Computer Science, vol. 2392, Springer, Berlin, 2002, pp. 438–455.

[7] G. Delzanno, A. Podelski, Model checking in CLP, in: Proc. 5th Internat. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 1579, Springer, Berlin, 1999, pp. 223–239.

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

[9] D.L. Detlefs, G. Nelson, J.B. Saxe, Simplify: a theorem prover for program checking, Technical Report HPL-2003-148, HP Labs, 2003.

[10] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[11] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, H. Zheng, Tool-supported program abstraction for finite-state verification, in: Proc. 23rd Internat. Conf. Software Engineering, Toronto, Canada, 2001.

[12] M.D. Ernst, A. Czeisler, W.G. Griswold, D. Notkin, Quickly detecting relevant program invariants, in: Proc. 22nd Internat. Conf. Software Engineering, Limerick, Ireland, June 2000.

[13] C. Flanagan, R. Joshi, K.R.M. Leino, Annotation inference for modular checkers, Inform. Process. Lett. 77 (2–4) (2001) 97–108.

[14] C. Flanagan, K.R.M. Leino, Houdini, an annotation assistant for ESC/Java, in: J.N. Oliveira, P. Zave (Eds.), FME 2001: Formal Methods for Increasing Software Productivity, Berlin, Germany, Lecture Notes in Computer Science, vol. 2021, Springer, Berlin, 2001, pp. 500–517.

[15] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for Java, in: Proc. Conf. Programming Language Design and Implementation, June 2002, pp. 234–245.

[16] C. Flanagan, J.B. Saxe, Avoiding exponential explosion: generating compact verification conditions, in: Conf. Record of the 28th Annual ACM Symposium on Principles of Programming Languages, London, UK, January 2001, pp. 193–205.

[17] S. Graf, H. Saïdi, Construction of abstract state graphs via PVS, in: Computer Aided Verification, 9th Internat. Conf., Lecture Notes in Computer Science, vol. 1254, Springer, Berlin, 1997, pp. 72–83.

[18] E. Gunter, D. Peled, Tracing the executions of concurrent programs, in: K. Havelund, G. Rosu (Eds.), Electronic Notes in Theoretical Computer Science, vol. 70, Elsevier, Amsterdam, 2002.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, Lazy abstraction, in: Proc. 29th Symp. Principles of Programming Languages, London, UK, January 2001, pp. 28–70.

[20] J. Jaffar, J.L. Lassez, Constraint logic programming, in: Proc. ACM SIGPLAN Symp. Principles of Programming Languages, Munich, Germany, January 1987, pp. 111–119.

[21] J. Jaffar, M.J. Maher, Constraint logic programming: a survey, J. Logic Programming 19/20 (1994) 503–581.

[22] J. Jaffar, M.J. Maher, K. Marriott, P.J. Stuckey, The semantics of constraint logic programs, J. Logic Programming 37 (1–3) (1998) 1–46.

[23] L. Lamport, How to Write a Long Formula, Technical Report 119, DEC Systems Research Center, 1994.

[24] M.J. Maher, A logic programming view of CLP, in: Internat. Conf. Logic Programming, Budapest, Hungary, 1993, pp. 737–753.
[25] K.L. McMillan, Symbolic Model Checking: An Approach to the State-Explosion Problem, Kluwer Academic Publishers, Dordrecht, 1993.
[26] J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems in CESAR, in: 5th Internat. Symp. Programming, Lecture Notes in Computer Science, vol. 137, Springer, Berlin, 1982, pp. 337–351.
[27] A.U. Shankar, An introduction to assertional reasoning for concurrent systems, Comput. Surveys 25 (3) (1993) 225–302.
[28] SICStus Prolog, On the web at http://www.sics.se/sicstus/.
[29] S. Stoller, Model-checking multi-threaded distributed Java programs, in: Proc. 7th Internat. SPIN Workshop on Model Checking and Software Verification, Lecture Notes in Computer Science, vol. 1885, Springer, Berlin, 2000, pp. 224–244.
[30] E. Yahav, Verifying safety properties of concurrent Java programs using 3-valued logic, in: Proc. 28th Symp. Principles of Programming Languages, London, UK, January 2001, pp. 27–40.