University "Alexandru Ioan Cuza" of Iaşi
Faculty of Computer Science

TECHNICAL REPORT

RMT: Proving Reachability Properties in Constrained Term Rewriting Systems Modulo Theories

Ștefan Ciobâcă, Dorel Lucanu

TR 16-01, October 2016

ISSN 1224-9327

Universitatea "Alexandru Ioan Cuza" Iaşi
Facultatea de Informatică
Str. Berthelot 16, 6600-Iaşi, Romania
Tel. +40-32-201090, email: tr@info.uaic.ro

RMT: Proving Reachability Properties in Constrained Term Rewriting Systems Modulo Theories

Ștefan Ciobâcă and Dorel Lucanu

Alexandru Ioan Cuza University, Iași
{stefan.ciobaca,dlucanu}@info.uaic.ro

Abstract. We introduce the RMT tool, which takes as input a constrained term rewriting system $\mathcal{R}$ and proves reachability properties of the form $\forall \tilde{x}.(c_1(\tilde{x}) \rightarrow \exists \tilde{y}.(c_2(\tilde{x},\tilde{y}) \wedge t_1(\tilde{x}) \Rrightarrow^*_{\mathcal{R}} t_2(\tilde{x},\tilde{y})))$, where c_1, c_2 are constraints over the variables $\tilde{x}$ and respectively $\tilde{x} \cup \tilde{y}$ and t_1, t_2 are terms over the variables $\tilde{x}$ and respectively $\tilde{x} \cup \tilde{y}$. By the relation $\Rrightarrow^*_{\mathcal{R}}$, we mean that $t_2(\tilde{x},\tilde{y})$ is reachable across all paths in $\mathcal{R}$ starting in $t_1(\tilde{x})$. The reachability guarantee is sound for terminating paths starting in instances of $t_1(\tilde{x})$ and it was initially motivated by proving partial correctness of programs. The constrained term rewriting system is allowed to contain both interpreted and uninterpreted function symbols. The interpreted symbols are part of a theory that is a parameter to our tool and all rewriting steps are defined *modulo* this theory. In practice, the interpreted symbols are handled by relying on an SMT solver.

Keywords: constrained term rewriting systems, satisfiability modulo theories, reachability, partial correctness

Introduction

Program analysis and program verification play an important role in the field of software development, especially for critical systems. There have been several lines of work in which program analyses are reduced to queries on term rewriting systems with logical constraints ([12,13,24,14]). By the same reduction, correctness of a program corresponds to reachability in such constrained term rewriting systems (from hereon, CTRSs). Moreover, CTRSs can faithfully capture the operational semantics of any programming language ([32]), which means that the question of program correctness corresponds *exactly* to the problem of reachability in these CTRSs, allowing for language-parametric proofs of programs ([28,29,7,8]).

Therefore, tools that operate on CTRSs can be used to reason about real-world programs. Existing tools for CTRSs (e.g., [17,19]) concentrate on proving termination, confluence and term equivalence but there is a lack of tools that can prove reachability properties, which correspond to correctness claims; our system is meant to fill this gap.

We introduce RMT, a tool for proving reachability properties in constrained term rewriting systems modulo theories. It is available at

`http://profs.info.uaic.ro/~stefan.ciobaca/rmt`.

RMT is a command line tool that reads a file containing (one or more) CTRSs and a number of reachability queries. RMT can establish reachability in any bounded number of steps by exhaustive symbolic exploration of the possible rewrites, but it can also *prove* that a reachability property holds (even when the number of rewrite steps is not bounded a priori). In order to prove reachability properties in CTRSs, the user must provide, besides the reachability property itself, a number of helper reachability properties that also hold and which play the same role as invariants in regular program proofs. We call these helper properties *circularities*, as introduced by Roșu and Lucanu in [30]. For example, given the CTRS

$$\begin{aligned}\mathcal{R} = \{\ & \texttt{init}(u,v) \Rightarrow \texttt{loop}(u,v),\\ & \texttt{loop}(u,v) \Rightarrow \texttt{loop}(v,u-v)\ \ \texttt{if}\ \ v \neq 0,\\ & \texttt{loop}(u,v) \Rightarrow \texttt{done}(u)\ \ \texttt{if}\ \ v = 0\},\end{aligned}$$

which implements Euclid's algorithm, RMT can establish the following reachability property, which encodes the correctness of the algorithm:

$$\forall u, v. \exists x.\ (x = gcd(u,v)\ \wedge\ \texttt{init}(u,v) \Rrightarrow^+_{\mathcal{R}} \texttt{done}(x)),$$

with the help of the following circularity:

$$\forall u, v. \exists x. (x = gcd(u, v) \ \wedge \ \texttt{loop}(u, v) \Rrightarrow^*_{\mathcal{R}} \texttt{done}(x)).$$

The function symbol *gcd* is part of the built-in theory. The circularity encodes the behaviour of the rewriting loop and it essentially plays the same role as an invariant. There is no need to trust the circularities given by the user, as RMT will prove them as well. When RMT establishes a reachability property, then the property is sound for terminating instances of the terms on the left-hand side. Termination needs to be established in other ways, as the tool does not currently check it.

Structure. Section 2 contains the theoretical background relevant to our tool, including our modelling of constrained term rewriting systems modulo theories. Section 3 gives an overview of RMT and describes the main uses cases. Section 4 describes the architecture and the implementation of RMT. It includes the algorithm used for *unification modulo* and the algorithm implementing our procedure for reachability proofs that is based on a sound and relatively complete proof system for partial correctness of programs ([22,7]. Section 5) concludes the paper with a discussion of the tool and of related and future work.

Constrained Term Rewriting Systems

We consider term rewriting systems with rules of the form $l \Rightarrow r \ \texttt{if} \ b$, where a rewrite rule intuitively means that any instance of l such that the corresponding constraint b holds advances in the corresponding instance of r. Such rewrite systems are sufficiently expressive to handle the encoding of the operational semantics of programming languages ([32]).

Signature. We consider a many-sorted *built-in signature* $\Sigma_0 = (\mathcal{S}_0, \mathcal{F}_0)$ consisting of a set of sorts $\mathcal{S}_0$ and a set of function symbols Σ_0. Each function symbol $f \in \mathcal{F}_0$ has an arity denoted by $f : s_1 \times \ldots \times s_n \to s$, where $s_1, \ldots, s_n, s \in \mathcal{S}_0$ are built-in sorts. We also consider an order-sorted signature $\Sigma = (\mathcal{S}, \leq, \mathcal{F})$, where $\mathcal{S}$ is a set of sorts, $\leq$ is the subsorting relation and $\mathcal{F}$ is the set of function symbols (each function symbol $f \in \mathcal{F}$ also having an arity denoted by $f : s_1 \times \ldots \times s_n \to s$, with $s_1, \ldots, s_n, s \in \mathcal{S}$).

We assume that Σ extends Σ_0 as follows: (1) $\mathcal{S}_0 \subseteq \mathcal{S}$; (2) $\leq \subseteq \mathcal{S} \times (\mathcal{S} \setminus \mathcal{S}_0)$ is the subsorting relation (as the built-in theory is fixed, the built-in sorts are not allowed to be supersorts of any other sort) and (3) all built-in function symbols appear in $\mathcal{F}$: $\mathcal{F}_0 \subseteq \mathcal{F}$.

Additionally, for simplicity, we make the following assumptions: (1) any sort has at most one supersort; (2) overloading of function symbols is not allowed and (3) any non-built-in function symbol $f \in \mathcal{F} \setminus \mathcal{F}_0$ must return a value of a sort that is not built-in: i.e., if $f : s_1 \times \ldots \times s_n \to s \in \mathcal{F} \setminus \mathcal{F}_0$, then the sort $s \in \mathcal{S} \setminus \mathcal{S}_0$ must not be a built-in sort.

Terms. We let $\mathcal{X}$ be an $\mathcal{S}$-indexed family $\mathcal{X} = \{\mathcal{X}_s\}_{s \in \mathcal{S}}$ of sets of variables, where $\mathcal{X}_s$ denotes the variables of sort s. We assume as usual that each set $\mathcal{X}_s$ is countable and infinite and that no variable has two different sorts. As is usual, we may write $x \in \mathcal{X}$ or $x : s \in \mathcal{X}$ instead of $x \in \mathcal{X}_s$. The set of terms of sort $s \in \mathcal{S}$ is denoted by $\mathcal{T}_\Sigma(\mathcal{X})_s$ and is built inductively as usual. A context C with a placeholder of sort s is a term in $\mathcal{T}_\Sigma(\mathcal{X} \cup \{\_\})_s$, where the distinguished variable $\_$ of sort s is allowed to appear exactly once. If $t \in \mathcal{T}_\Sigma(\mathcal{X} \cup \{\_\})_s$ then we denote by $C[t]$ the term obtained from C by replacing the variable $\_$ with t.

Assumptions on Σ_0. We make a number of assumption on Σ_0, our built-in signature. These assumptions ensure that Σ_0 contains at least the symbols required for boolean terms, which are used to encode logical constraints: (1) the set of built-in sorts $\mathcal{S}_0$ must necessarily include a sort $Bool \in \mathcal{S}_0$ of *booleans*, but may of course contain other sorts as well and (2) the set of built-in function symbols $\mathcal{F}_0$, must include the following symbols:

$$\begin{array}{ll} true : \ \to Bool; & \neg : Bool \to Bool; \\ \wedge : Bool \times Bool \to Bool; & =_s : s \times s \to Bool, \text{ for every built-in sort } s \in \mathcal{S}_0. \end{array}$$

Our assumptions above are necessary to ensure that terms of sort *Bool* can encode any first-order logic formula with equalities on built-in sorts and without quantifiers: *true* represents truth, $\neg$ - logical negation, $\wedge$ - conjunction, $=_s$ - equality over sort s. Other boolean operations like implication ($\rightarrow$), disjunction ($\vee$), etc., are treated as syntactic sugar over the existing operations, as usual.

Substitutions. A substitution $\sigma : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$ is an S-indexed function such that $\sigma(x) \neq x$ for finitely many variables $x \in \mathcal{X}$. This set of variables is the domain of σ: $dom(\sigma) = \{x \mid \sigma(x) \neq x\}$. We assume as usual that σ extends homomorphically to the entire set of terms $\sigma : \mathcal{T}_\Sigma(\mathcal{X}) \to \mathcal{T}_\Sigma(\mathcal{X})$. We use the

standard notion for unifiers and we denote by $mgu(t_1, t_2)$ the unique (up to renaming ([1])) most general unifier of t_1 and t_2, if it exists.

Built-In Model. We assume that the terms of the built-in theory Σ_0 are interpreted according to a built-in model $\mathcal{M}_0$. The model $\mathcal{M}_0$ assigns to every built-in sort $s \in \mathcal{S}_0$ a set $[\![s]\!]_{\mathcal{M}_0}$ and to every function symbol $f : s_1 \times \ldots \times s_n \to s$, with $s_1, \ldots, s_n, s \in \mathcal{S}_0$, a function $[\![f]\!]_{\mathcal{M}_0} : [\![s_1]\!]_{\mathcal{M}_0} \times \ldots \times [\![s_n]\!]_{\mathcal{M}_n} \to [\![s]\!]_{\mathcal{M}_0}$. For the special case where $n = 0$, we assume that $[\![f]\!]_{\mathcal{M}_0} \in [\![s]\!]_{\mathcal{M}_0}$ is a single element.

Assumptions on $\mathcal{M}_0$. We assume that $\mathcal{M}_0$ assigns the expected interpretation to the sort *Bool* and to the function symbols operating on *Bool*:

1. The sort *Bool* is interpreted as the set of boolean values: $[\![Bool]\!]_{\mathcal{M}_0} = \{\top, \bot\}$;
2. The function symbols operating on *Bool* are interpreted as expected:
   (a) $[\![true]\!]_{\mathcal{M}_0} = \top$ is the logical true value;
   (b) $[\![\neg]\!]_{\mathcal{M}_0}$ is logical negation: $[\![\neg]\!]_{\mathcal{M}_0}(x) = \top$ iff $x = \bot$, for all $x \in [\![s]\!]_{\mathcal{M}_0}$;
   (c) $[\![\wedge]\!]_{\mathcal{M}_0} : \{\top, \bot\} \times \{\top, \bot\} \to \{\top, \bot\}$ is the *logical and* function:
   $$[\![\wedge]\!]_{\mathcal{M}_0}(x, y) = \top \text{iff } x = \top \text{ and } y = \top, \text{ for all } x, y \in [\![Bool]\!]_{\mathcal{M}_0};$$
   (d) $[\![=_s]\!]_{\mathcal{M}_0}$ is the equality for sort s: $[\![=_s]\!]_{\mathcal{M}_0}(x, y) = \top$ iff $x = y$, for all $x, y \in [\![s]\!]_{\mathcal{M}_0}$.

Note that $\mathcal{M}_0$ might contain other sorts such as *Integer* for integers, *Int32* for 32-bit integers, *Array* for arrays, etc. No assumption is made on the interpretation of these additional sorts.

Example 1. We use the following running example. Let $\mathcal{S}_0 = \{Bool, Int\}$ be the set of built-in sorts. Let $\mathcal{F}_0$ be the following S_0 sorted signature:

$$\mathcal{F}_0 = \{\wedge, \neg, true, =_{Bool}, =_{Int}, \leq, +, \times, -, /, 0, 1, 2, \ldots\}$$

of built-in function symbols. Let the model $\mathcal{M}_0$ interpret the sort *Bool* as required: $[\![Bool]\!]_{\mathcal{M}_0} = \{\top, \bot\}$ and the sort *Int* as the sort of integers: $[\![Int]\!]_{\mathcal{M}_0} = \mathbb{Z}$. The built-in function symbols $\wedge$, $\neg$, *true*, $=_{Bool}$, $=_{Int}$, $\leq$, $+$, $\times$, $-$, $/$, 0, 1, 2, etc. are interpreted in $\mathcal{M}_0$ as expected: $[\![\wedge]\!]_{\mathcal{M}_0}$ is *logical and*, $[\![\neg]\!]_{\mathcal{M}_0}$ is *logical not*, $[\![true]\!]_{\mathcal{M}_0} = \top$ is the *true* value, $[\![=_{Bool}]\!]_{\mathcal{M}_0}$ is *equality* over truth values, $[\![=_{Int}]\!]_{\mathcal{M}_0}$ is *equality* over integers, $[\![\leq_{Int}]\!]_{\mathcal{M}_0}$ is the *less-than* relation on integers, $[\![+]\!]_{\mathcal{M}_0}$, $[\![\times]\!]_{\mathcal{M}_0}$, $[\![-]\!]_{\mathcal{M}_0}$ and $[\![/]\!]_{\mathcal{M}_0}$ are the *addition*, *multiplication*, *subtraction* and *division* [1] functions on integers and $[\![0]\!]_{\mathcal{M}_0}, [\![1]\!]_{\mathcal{M}_0}, [\![2]\!]_{\mathcal{M}_0}, \ldots$ are the integers $0, 1, 2, \ldots$.

For function symbols that are usually written using infix notation, we adopt the same principle and write e.g. $t_1 \leq t_2$ instead of $\leq (t_1, t_2)$, $t_1 + t_2$ instead of $+(t_1, t_2)$, $t_1 =_{Int} t_2$ instead of $=_{Int}(t_1, t_2)$, etc.

Model. Given the built-in model $\mathcal{M}_0$ for the built-in signature Σ_0, we define a model $\mathcal{M}$ of Σ by interpreting the built-in symbols according to $\mathcal{M}_0$ and the additional symbols freely (as constructors):

1. Any built-in sort is interpreted as in $\mathcal{M}_0$: for any $s \in \mathcal{S}_0$, $[\![s]\!]_{\mathcal{M}} = [\![s]\!]_{\mathcal{M}_0}$;
2. Any non-built-in sort $s \in \mathcal{S} \setminus \mathcal{S}_0$ is interpreted as follows:
   $$[\![s]\!]_{\mathcal{M}} = \bigcup_{f \in \Sigma, f : s_1 \times \ldots \times s_n \to s', s' \leq s} \{f(e_1, \ldots, e_n) \mid e_1 \in [\![s_1]\!]_{\mathcal{M}}, \ldots, e_n \in [\![s_n]\!]_{\mathcal{M}}\};$$
3. Any built-in function is interpreted as in $\mathcal{M}_0$: for all $f \in \mathcal{F}_0$, $[\![f]\!]_{\mathcal{M}} = [\![f]\!]_{\mathcal{M}_0}$;
4. Any non-built-in function is interpreted as a constructor:
   $$\text{for all } f \in \mathcal{F} \setminus \mathcal{F}_0, [\![f]\!]_{\mathcal{M}}(x_1, \ldots, x_n) = f(x_1, \ldots, x_n);$$

Valuations. A valuation $\rho : \mathcal{X} \to [\![\mathcal{S}]\!]_{\mathcal{M}}$ is an $\mathcal{S}$-indexed function that assigns to every variable $x : s \in \mathcal{X}$ an element of $[\![s]\!]_{\mathcal{M}}$, for every sort $s \in \mathcal{S}$. Valuations extend homomorphically to terms: $\rho(f(t_1, \ldots t_n)) = [\![f]\!]_{\mathcal{M}}(\rho(t_1), \ldots, \rho(t_n))$, for any terms $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(\mathcal{X})$ of appropriate sorts.

Rewrite Systems. A *constrained rewrite rule* of sort $s \in \mathcal{S}$ is a tuple (l, r, b), where $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_s$ are terms of sort s and $b \in \mathcal{T}_\Sigma(\mathcal{X})_{Bool}$ is a term of sort *Bool*. We make no assumptions on how variables occur in l, b or r. For the rest of the article, we use the shorthand *rewrite rule* instead of *constrained rewrite rule*. We also write $l \Rightarrow r$ `if` b instead of (l, r, b) for such a rewrite rule. We write $l \Rightarrow r$ instead of $l \Rightarrow r$ `if` *true*. A *constrained rewrite system* $\mathcal{R}$ is a set of constrained rewrite rules. We sometimes use *rewrite system* or the CTRS instead of *constrained rewrite system* in the rest of the article.

[1] The division function is defined to truncate the result when the division is not exact and to return an arbitrary integer when the divisor is zero.

Definition 1 (The Transition Relation Generated by a CTRS). *The interpretation of a rewrite system $\mathcal{R}$ is an S-indexed transition relation $\Rightarrow_{\mathcal{R}}: \mathcal{M} \times \mathcal{M}$, where $u \Rightarrow_{\mathcal{R}} v$ if there exists a context $C[\cdot]$, a rewrite rule $l \Rightarrow r$ $\texttt{if}$ $b \in \mathcal{R}$ and a valuation $\rho : \mathcal{X} \to \mathcal{M}$ such that $u = \rho(C[l])$, $v = \rho(C[r])$ and $\rho(b) = \top$.*

By $\Rightarrow_{\mathcal{R}}^{*}$ and respectively $\Rightarrow_{\mathcal{R}}^{+}$ we denote the transitive and reflexive closure and respectively the transitive closure of $\Rightarrow_{\mathcal{R}}$.

Example 2. Continuing Example 1, we define a signature $\Sigma = (S, \mathcal{F})$ as follows: (1) we consider a single non-built-in sort *State*: $\mathcal{S} = \mathcal{S}_0 \cup \{State\}$; (2) we let $\mathcal{F} = \mathcal{F}_0 \cup \{\texttt{init} : Int \to State, \texttt{loop} : Int \times Int \to State, \texttt{done} : Int \to State\}$. We define the rewrite system $\mathcal{R}$ to consists of the following rules:

$$\mathcal{R} = \left\{ \begin{array}{l} \texttt{init}(n) \Rightarrow \texttt{loop}(0, n), \\ \texttt{loop}(s, n) \Rightarrow \texttt{loop}(s + n, n - 1) \texttt{ if } 1 \leq n, \\ \texttt{loop}(s, n) \Rightarrow \texttt{done}(s) \texttt{ if } \neg(1 \leq n) \end{array} \right\},$$

where $s, n \in \mathcal{X}_{Int}$ are variables. We then have: $\texttt{init}(3) \Rightarrow_{\mathcal{R}} \texttt{loop}(0, 3) \Rightarrow_{\mathcal{R}} \texttt{loop}(3, 2) \Rightarrow_{\mathcal{R}} \texttt{loop}(5, 1) \Rightarrow_{\mathcal{R}} \texttt{loop}(6, 0) \Rightarrow_{\mathcal{R}} \texttt{done}(6)$. It is easy to see that the rewrite system computes the sum of the first n positive naturals.

Constrained Terms. A *constrained term* of sort $s \in \mathcal{S}$ is a pair (t, b), where $t \in \mathcal{T}_{\Sigma}(\mathcal{X})_s$ is a term of sort s and $b \in \mathcal{T}_{\Sigma}(\mathcal{X})_{Bool}$ is a term of sort *Bool*. We use the notation l $\texttt{if}$ b for the constrained term (l, b). Constrained terms can be seen as particular cases of matching logic formulas ([27]). A term t is identified with the constrained term t $\texttt{if}$ *true*.

Definition 2 (Reachability in CTRSs).
We write $\mathcal{R} \models l \Rrightarrow^{+} r$ $\texttt{if}$ b (respectively $\mathcal{R} \models l \Rrightarrow^{*} r$ $\texttt{if}$ b) if, for all valuations ρ such that $\rho(b) = \top$ and for all executions $\rho(l) \Rightarrow_{\mathcal{R}} \gamma_1 \Rightarrow_{\mathcal{R}} \gamma_2 \Rightarrow_{\mathcal{R}} \ldots$, there exists $i \in \{1, 2, \ldots\}$ (respectively $i \in \{0, 1, 2, \ldots\}$, where $\gamma_0 = \rho(l)$) such that $\gamma_i = \rho(r)$.

The notation captures the fact that, any element matching l $\texttt{if}$ b advances, along any execution path, into an element matching r.

Example 3. Continuing Example 2, it is possible to show that

$$\mathcal{R} \models \texttt{init}(n) \Rrightarrow^{+} \texttt{done}(n \times (n + 1)/2) \texttt{ if } 0 \leq n.$$

Tool Overview

In this section, we provide an overview of our tool and its capabilities. `RMT` is a command line tool that processes a single file that contains the definition of any number of CTRSs and several reachability queries and answers each query in turn. In the following subsection, we describe the two main uses of the tool: (1) computing the set of reachable terms after a bounded number of rewrite steps and (2) proving reachability properties where the number of rewrite steps is not necessarily bounded.

Computing the set of Reachable Terms in a Bounded Number of Rewrite Steps

We first explain how to use `RMT` to automatically compute the set of terms reachable in a bounded number of rewrite steps. Figure 1 shows the input given to the tool in order to compute the direct symbolic successors (terms reachable in one step) of the term `loop(S, N)` in the CTRS that we have defined in our running example. Some parts of the input are elided and replaced with `[...]` for brevity. Although the input might seem overwhelming, most of it is boilerplate needed for the interaction with the SMT solver.

We now explain all the code in Figure 1. Any file given as input must begin with the set of sorts. In this example we define, on the first line, three sorts: `Int`, `Bool` and `State`. The first two sorts are defined as built-in and are assigned interpretations, using the `/` modifier. The string following `/` is the interpretation of the sort, specified as the name of the set that is used in the `Z3` SMT solver ([9]). After declaring the sorts, we must declare all sub-sorting relations. In our example, there are no sub-sorts and therefore the example sub-sorting is commented out.

Next follows the set of function symbols, introduced by the `signature` declaration on lines 3 to 10. Each function symbol is defined using the syntax `name : Sort1 ... SortN -> Sort`, which specifies the

```
1  sorts Int / "Int", Bool / "Bool", State; // subsort Int < State;

   signature mzero : -> Int / "0", mone : -> Int / "1",
     mplus : Int Int -> Int / "+", mminus : Int Int -> Int / "-",
     mle : Int Int -> Bool / "<=", mequals : Int Int -> Bool / "=", [...]
6
     bequals : Bool Bool -> Bool / "=", band : Bool Bool -> Bool / "and",
     bimplies : Bool Bool -> Bool / "=>", true :  -> Bool / "true", [...]

     init : Int -> State, loop : Int Int -> State, done : Int -> State;
11
   variables B : Bool, S : Int, N : Int, I : Int;

   rewrite-system simplifications
     bnot(false) => true, bnot(true) => false, bnot(bnot(B)) => B, [...];
16
   constrained-rewrite-system sum
     init(N) => loop(mzero, N),
     loop(S, N) /\ mle(mone, N) => loop(mplus(S, N), mminus(N, mone)),
     loop(S, N) /\ bnot(mle(mone, N)) => done(S);
21
   search in sum : loop(S, N);
```

Fig. 1. Computing the successors of a (constrained) term.

arity of the symbol. If a slash symbol ("`/`") follows the declaration, it means that the function symbol is built-in and the string following "`/`" defines the interpretation of the symbol. The interpretation is given as the name of the function as defined in SMT-LIB ([3](; we currently also allow extensions specific to `Z3` ([9]).

For our example, we have defined a bunch of function symbols operating on integers and prefixed with `m`: e.g. `mplus` is a function symbol interpreted as addition (`+`). There are also several function symbols starting with `b` which operate on booleans: e.g. `bnot` is defined to be a function symbol interpreted by logical negation. We also need to declare the function symbols for the constants $0, 1, \ldots \in \mathbb{Z}$. In our case, we use the nullary function symbols `mzero` and `mone`, interpreted as 0 and respectively 1. We currently require the user to declare these symbols in order to obtain maximum flexibility, but in the future we might revisit this decision and reserve special names for the symbols that are used extensively, such as integers.

The last chunk of symbols of the `signature` declaration, on line 10, consists of the free symbols `init`, `loop` and `done`, which are used in our running example. After declaring the function symbols, we require to define all variables (line 12) that are used afterwards in the file by providing their sort. In our example, we have the following variables: B of sort *Bool* and S, N, I of sort *Int*.

The next declaration (lines 14 and 15) is a rewrite system having the name `simplifications`. If there is a rewrite system with this name in the file, it will used to simplify terms (including constraints) before they are presented to the user, but also during the internal computations. This helps with legibility. It is the responsibility of the user to provide sound rewrite rules that terminate. Note that all terms are written in infix notation. This is in order to eliminate parsing ambiguities which can, in our experience, be a source of unexpected and difficult to debug errors.

The rest of the file consists of any number of declarations of rewrite systems and constrained rewrite systems, followed by a number of queries. In our example, there is a single constrained rewrite system called `sum` which encodes the rewrite system from Example 2. Note that constrained rules $l \Rightarrow r$ `if` b are written in the format `l /\ b => r`. For example, the rewrite rule `loop`$(s, n) \Rightarrow$ `loop`$(s+n, n-1)$ `if` $1 \leq n$ is written in the input as `loop(S, N) /\ mle(mone, N) => loop(mplus(S, N), mminus(N, mone))`. Although this verbosity seems unnecessary, it eliminates ambiguities and saves time in the long run.

There is a single query in the file, which asks for the direct symbolic successors of `loop(S, N)` in the rewrite system `sum`. The result of the query is:

```
2 solutions.
Solution #1: loop(mplus(S,N),mminus(N,mone)) /\ mle(mone,N)
3 Solution #2: done(S) /\ bnot(mle(mone,N))
```

For a query of the form `search t /\ b` (where the constraint `/\ b` is optional), the result is a list of solutions written in the form `s /\ c`, where `s` is a term of the same sort as `t` and `c` is a constraint (a term of sort *Bool*). The list of solutions should intuitively be interpreted as follows: if `c` holds, then any instance of t `if` b rewrites in one step into `s`. If we change our query to `search in sum : loop(S, mtwo);`, then the `RMT` returns a single solution:

```
1 solutions.
Solution #1: loop(mplus(S,mtwo),mminus(mtwo,mone)) /\ mle(mone,mtwo)
```

The solution corresponds to the application of the second rewrite rule (line 19, Figure 1) in the rewrite system. The third rule (line 20, Figure 1) cannot be applied because the constraint `bnot(mle(mone, mtwo))` is not satisfiable. The `RMT` tool detects unsatisfiable constraints by passing them to the `Z3` SMT solver. If a constraint cannot be proven unsatisfiable by `Z3` (i.e. `Z3` returns `sat` or `unknown`), then `RMT` will take the conservative approach and assume it might be satisfiable. Note that constraints are simplified only by applying rules in the `simplifications` rewrite system, if such as system is declared. This means that constraints can sometimes be presented in a form that is more complex than necessary.

By default, `RMT` computes all *direct* successors of the queried term. However, it is possible to determine all terms reachable in any number of steps by providing two optional parameters: the minimum and maximum search depth. For example, to find all elements reachable in one up to two steps, we use the following query: `search [1,2] in sum : loop(S, N);`. Then `RMT` returns all successors which are reachable in any number of steps between 1 and 2:

```
4 solutions.
Solution #1: loop(mplus(S,N),mminus(N,mone)) /\ mle(mone,N)
Solution #2: loop(mplus(mplus(S,N),mminus(N,mone)),mminus(mminus(N,mone),mone))/\
   band(mle(mone,N),mle(mone,mminus(N,mone)))
Solution #3: done(mplus(S,N)) /\ band(mle(mone,N),bnot(mle(mone,mminus(N,mone))))
Solution #4: done(S) /\ bnot(mle(mone,N))
```

Proving Reachability Properties

Searching, as described in Section 3.1, can be used to explore the behaviour of any constrained term in a bounded number of steps. This is useful for many reasons; however, the most important use case for `RMT` is to *prove* reachability properties in constrained term rewriting systems, in the sense of Definition 2, even if the number of transitions is unbounded.

We will continue Example 3 and we will show how to use `RMT` to prove that

$$\mathcal{R} \models \texttt{init}(n) \Rrightarrow^{+} \texttt{done}(n \times (n+1)/2) \texttt{ if } 0 \leq n. \tag{1}$$

We would like to emphasise that our tool is able to establish such properties, but these properties hold for terminating paths that start in ground instances of the left-hand side that satisfy the constraint, just like in *partial* correctness of programs (as opposed to *total* correctness, were termination is also established). `RMT` cannot prove termination and therefore termination needs to be established in other ways: for example, by hand or by using other tools such as [20,17].

In order to prove a reachability property such as the one in Equation (1), we will need to additionally prove a helper reachability property that is more general. These helper properties play roughly the same role of invariants in program proofs and we call them *circularities*, as introduced in [30]. A circularity is very much like a regular (constrained) rewrite rule, but, before it is applied, it must be *proven* to be sound with respect to the trusted rewrite rules. However, a circularity can be applied to prove itself, after progress has been made in the rewrite system. This use of circularities is sound ([22,7]) for terminating paths.

In order to prove our desired reachability property in Equation (1), we need to prove alongside it a single additional circularity, which will be used to account for the repetitive behaviour of the rewrite system:

$$\mathcal{R} \models \texttt{loop}(s, i) \Rrightarrow^{+} \texttt{done}(s + i \times (i+1)/2) \texttt{ if } 0 \leq i. \tag{2}$$

All reachability properties that `RMT` needs to prove, including circularities, should be placed in a single CTRS. In the tool, we do not make any distinction between the reachability properties that are the final objective of our proof and the helper reachability properties and we call all of them circularities.

Continuing our running example, we place both the desired reachability property in Equation (1) and the circularity in Equation (2) as rewrite rules that are part of a rewrite system called `circularities`, featured in Figure 2.

The `prove` command then tries to prove that all of the reachability properties given as rewrite rules in the rewrite system `circularities` are sound with respect to the rewrite system `sum`. In our case, the proof succeeds and the tool outputs a proof tree shown in Figure 3. `RMT` outputs the terms appearing in the proof using mixed syntax to save space: all interpreted function symbols are printed using SMT syntax. In the output, we replaced some subparts of the proof tree by `[ ... ]` in order to save space.

```
constrained-rewrite-system circularities
  init(N) /\ mle(mzero,N) => done(mdiv(mtimes(N,mplus(N,mone)),mtwo)),
  loop(S,I) /\ mle(mzero,I) => done(mplus(S,mdiv(mtimes(I,mplus(I,mone)),mtwo)));
prove in sum : circularities;
```

Fig. 2. Input needed for the set of reachability properties given in the `circularities` rewrite system.

```
Proving circularity #1:
--------
    - init(N) /\ false -----> done((div (* N (+ N 1)) 2))
    - init(N) /\ false =(C)=> done((div (* N (+ N 1)) 2))
    [ ... ]
    * Proved that loop(0,N) /\ (<= 0 N) => done((div (* N (+ N 1)) 2))
    - init(N) /\ (<= 0 N) =(R)=> done((div (* N (+ N 1)) 2))
* Proved that init(N) /\ (<= 0 N) => done((div (* N (+ N 1)) 2))
--------
Circularity #1 proved.

Proving circularity #2:
--------
    - loop(S,I) /\ false -----> done((+ S (div (* I (+ I 1)) 2)))
    - loop(S,I) /\ false =(C)=> done((+ S (div (* I (+ I 1)) 2)))
    [ ... ]
    * Proved that loop((+ S I),(- I 1)) /\ (and (<= 0 I) (<= 1 I)) =>
                  done((+ S (div (* I (+ I 1)) 2)))
    [ ... ]
    * Proved that done(S) /\ (and (<= 0 I) (not (<= 1 I))) =>
                  done((+ S (div (* I (+ I 1)) 2)))
    - loop(S,I) /\ (<= 0 I) =(R)=> done((+ S (div (* I (+ I 1)) 2)))
* Proved that loop(S,I) /\ (<= 0 I) => done((+ S (div (* I (+ I 1)) 2)))
--------
Circularity #2 proved.
```

Fig. 3. The result of running the `prove` command when a proof succeeds.

The system starts by turning all circularities into proof obligations. When trying to discharge a proof obligation of the form $l \Rrightarrow^* r$ `if` b, the system first tries to see if l `if` b *implies* r, in the sense that for any valuation ρ, if $\rho(b) = \top$, then $\rho(l) = \rho(r)$. If this is the case, then the proof is done. However, it might be that l `if` b implies r just in some cases. In general, the system will find a constraint c such that l `if` $(b \wedge c)$ implies r. It is always possible to find such a contraint: in the worst case we can pick $c = \mathit{false}$. This is exactly what happens when we try to prove the first circularity, namely that `init`$(N) \Rrightarrow$ `done`$(N \times (N+1)/2)$. As there can be no implication, the system finds that the weakest constraint under which the implication holds is `false` and therefore outputs the line `- init(N) /\ false -----> done((div (* N (+ N 1)) 2))` (the long arrow means implication).

The next step is to try to apply the circularities themselves. This is only allowed once progress is made and, as we have not made progress yet, the circularities cannot be applied. The system outputs `- init(N) /\ false =(C)=> done((div (* N (+ N 1)) 2))`, meaning that circularities can be applied only if `false`. The next line, namely `* Proved that loop(0,N) /\ (<= 0 N) => done((div (* N (+ N 1)) 2))`, means that the system managed to prove this reachability property recursively. The part of the proof that is missing and which is denoted by `[ ... ]` is a recursive proof of this reachability property. Since the only successor of `init(N) /\ (<= 0 N)` is `loop(0,N)`, it follows that what we wanted to prove also holds: `* Proved that init(N) /\ (<= 0 N) => done((div (* N (+ N 1)) 2))`.

The second circularity is more interesting. It cannot be discharged directly by implication or circularities and therefore we need to look at each successor of `loop(S,I) /\ (<= 0 I)`. There are two cases to consider: when `(<= 1 I)` the successor is `loop((+ S I),(- I 1))` and when `(not (<= 1 I))` the sucessor is `done(S)`. The system recursively proves that it is possible to reach the final state from both successors (in the lines starting with `*`) and therefore concludes that the circularity holds.

The case above is the happy case in which the proof goes through. However, if the circularities provided are not strong enough or if the depth limit of the proof tree is reached, then the `prove` command fails. In this case, the proof obligations that could not be discharged are printed out and they may be used in order to discover why the proof did no go through and to improve the circularities. For example, one of typical mistakes is to forget the constraint `mle(N, mzero)`:

```
constrained-rewrite-system circularities
  init(N) => done(mdiv(mtimes(N,mplus(N,mone)),mtwo)),
  loop(S,I) => done(mplus(S,mdiv(mtimes(I,mplus(I,mone)),mtwo)));
prove in sum : circularities;
```

The reachability properties given in the `circularities` rewrite system do not hold since when `N` is negative, the result is not the sum of the first `N` positive natural numbers (it is zero instead). In this case, the system tries to make the proof, but the second circularity fails. The output explains which proof obligations failed:

```
Circularity #2 not proved. The following proof obligations failed:
done(S) /\ (and (not (<= 1 I)) (not (=> (not (<= 1 I))
                                        (= (+ S (div (* I (+ I 1)) 2)) S))))
=> done((+ S (div (* I (+ I 1)) 2)))
```

By investigating the condition, we see that the system could not prove that `(div (* I (+ I 1)) 2)` is zero when `I < 1`. This suggests that we need to disallow `I < 0`, just as we have shown in Figure 2.

The `prove` command also features two optional parameters: the maximum depth of the proof (by default 100) and the maximum branching depth of the proof (by default 2). By branching depth we mean the number of rewrite steps where at least two branches are possible. So the previous query `prove in sum : circularities;` is equivalent to the command `prove [100,2] in sum : circularities;`. If the proof fails because of the depth limitations, the user can try to increase these parameters.

Architecture and Implementation

The `RMT` tool is written entirely in C++. It contains roughly 5000 lines of code, including comments and blank lines. `RMT` depends only on the standard C++ libraries and it can be compiled by any relatively modern C++ compiler out of the box. The only dependency is the `Z3` SMT solver, which should be installed and its binary should be in the system path, as `RMT` interacts with `Z3` through system calls.

At the heart of `RMT` is a hierarchy of classes for representing variables, function symbols and terms. Terms are stored in DAG format, with maximum structure sharing. The `Term` class is abstract and there are two derived classes: `VarTerm`, which stores terms that are variables and `FunTerm`, which stores terms that start with a function symbol. All terms are stored as pointers to an instance of `Term`. The `FunTerm` class contains, in addition to the function symbol, an array of `Term` pointers which contain the arguments to the function symbol.

All terms are created by a factory which maintains a pool of terms and which ensures that any single term is only allocated once: if the term already exists, it returns the existing pointer to this term. This makes structure sharing easy and, moreover, it means that syntactic term equality is equivalent to a pointer comparison, which is very fast. Constraints are stored internally as terms of sort `Bool`, which is what all modern automated reasoning systems aim to do ([21]).

The syntactic unification operation is implemented in essentially linear time and there is aggressive caching of the results of rewrites and unifications. The current bottleneck of the system is represented by the calls to `Z3`, which take place by writing a file in SMT-LIB ([3]) format and making a system call to `Z3`. This is not efficient, but it is very flexible since we can use other SMT solvers very easily. In the future however, we may want to revisit this design choice as calling `Z3` using its API would improve performance significantly.

In the following subsections, we present the algorithms implemented in `RMT` for *unification modulo theories*, which is used to compute the symbolic successors of a term and for *proving reachability properties*.

Implementation of Search

In order to compute all successors of a term, in the sense explained in Section 3.1, we employ the following algorithm for *unification modulo the built-in theory*. The algorithm for unification modulo theories takes as input two terms and produces the weakest constraint that makes the two terms equal, modulo the built-in theory.

The algorithm is based on the notion of *abstraction* of a term. The abstraction of a term t is a pair (s, σ), where (1) $dom(\sigma)$ is a fresh set of variables, (2) s is a term containing only uninterpreted function symbols and variables in the domain of σ and (3) $\sigma(x)$ contains only interpreted function symbols, for all $x \in dom(x)$. The existence of the abstraction of any term is guaranteed by the assumptions made in Section 2.

The assumptions in Section 2 ensure that the algorithm above is sound and complete. In order to compute the successors of a constrained term t `if` b with respect to a constrained rewrite system, we unify all subterms s of t (i.e. $t = C[s]$ for some context C) with all the terms l appearing as the left-hand side of a rewrite rule $l \Rightarrow r$ `if` b'. As usual, a fresh instance of the rule is used. Whenever s unifies with l (with result (π, c)), we obtain a possible successor of $C[s]$ `if` b, namely $C[r\pi]$ `if` $b\pi \wedge b'\pi$. More formally,

Algorithm 1 Algorithm for Unification Modulo

function UNIFICATION(t_1, t_2) ▷ precondition: t_1, t_2 do not share variables
  ▷ returns: a substitution π and a constraint c such that $\models c \to t_1\pi = t_2\pi$
  compute (s, σ), the abstraction of t_1
  if $\tau \leftarrow mgu(s, t_2)$ exists **then**
    $\pi \leftarrow \tau \circ \sigma$
    $c \leftarrow \bigwedge_{x \in dom(\sigma) \cap dom(\tau)} \sigma(x) = \sigma(\tau(x))$
    return (π, c) ▷ the terms are unifiable when c holds
  else
    return false ▷ the terms are not unifiable

we establish in this way that

$$b\pi \wedge b'\pi \to t\pi \Rightarrow_{\mathcal{R}} C[r\pi].$$

When the SMT solver can prove that the resulting constraint is unsatisfiable, the successor is dropped. If the SMT solver shows that the constraint is satisfiable or if it cannot determine the satisfiability status (i.e. if it returns `unknown` or if it time-outs), then we conservatively assume that this successor is possible. In the implementation, we also perform a set of simplifications on the resulting constraint in order to keep it readable: firstly, if there is a constraint of the form $x = u$, where $x \in \mathcal{X}$ is a variable and u is any term, than the constraint is removed and x is replaced by u throughtout the term and the rest of the constraint; secondly, we apply the rewrite rules in the rewrite system called `simplification`, if this rewrite system exists.

Implementation of Reachability Proving

The implementation of the proof procedure is a recursive function that takes as parameters a proof obligation ($l \Rrightarrow r$ `if` b), a set of circularities $\mathcal{C}$, a constrained term rewriting system $\mathcal{R}$ and a boolean flag remembering if progress has been made. The set of circularities $\mathcal{C}$ is syntactically given as a constrained term rewriting system. Circularities (l, r, b) are interpreted both as all-path proof obligations $l \Rrightarrow r$ `if` b and as regular constrained rewrite rules $l \Rightarrow r$ `if` b, depending on context. The function always returns successfully, but it records all proof obligations that cannot be discharged. The pseudo-code for the proof procedure is given in Algorithm 2. In practice, there is also a depth limit implemented which does not allow the procedure to go in an infinite loop.

Algorithm 2 Procedure for discharging proof obligations

1: **procedure** PROVE((l, b, r), $\mathcal{C}$, $\mathcal{R}$, *progress*) ▷ PROVE tries to show $\mathcal{R} \models l$ `if` $b \Rrightarrow^* r$
2: ▷ firstly, try to make a proof by implication:
3: $c \leftarrow$ the weakest constraint such that $\models b \wedge c \to l = r$
4: ▷ secondly, try to make a proof by circularities:
5: **if** *progress* **then** ▷ only use circularities if progress has been made
6: find all successors $\{s_i$ `if` $b_i\}_{i \in \{1,\ldots,n\}}$ of the constrained term l `if` $(b \wedge \neg c)$
7: in the rewrite system $\mathcal{C}$
8: **for all** $i \in \{1, \ldots, n\}$ **do**
9: PROVE($(s_i, (b \wedge \neg c \wedge b_i), r)$, $\mathcal{C}$, $\mathcal{R}$, *progress*)
10: ▷ lastly, try to make a proof by rewriting:
11: find all successors $\{t_i$ `if` $d_i\}_{i \in \{1,\ldots,m\}}$ of the constrained term
12: l `if` $(b \wedge \neg c \wedge \neg b_1 \wedge \ldots \wedge \neg b_n)$ in the rewrite system $\mathcal{R}$
13: **for all** $i \in \{1, \ldots, m\}$ **do** ▷ recurse for all successors; mark progress
14: PROVE($(s_i, (b \wedge \neg c \wedge \neg b_1 \wedge \ldots \wedge \neg b_n), r)$, $\mathcal{C}$, $\mathcal{R}$, *true*)
15: **if** $b \wedge \neg c \wedge \neg b_1 \wedge \ldots \wedge \neg b_n \wedge \neg d_1 \wedge \ldots \wedge \neg d_m$ is not unsatisfiable **then**
16: Recall (l, b, r) as a proof obligation that was not discharged

The **if** statement on line 17 ensures that all possible cases have been treated, either by implication, or by circularities, or by implication. In practice, before being sent to the SMT solver, the existential closure of the conditions in line 17 with respect to the variables on the left-hand side of the rewrite/circularity rules is computed.

A reachability property is successfully proved if there are no remaining proof obligation after the procedure ends. If the proof procedure successfully proves all circularities in $\mathcal{C}$, then the set of reachability properties in $\mathcal{C}$ hold, assuming that the terms on the left-hand side terminate. The successors of a constrained term are computed using the algorithm described in Section 3.1. The soundness of our algorithm is justified by the proof systems presented in [22,7].

More Challenging Examples

Proofs of programs. In this section, we would like to discuss the capabilities of our tool in terms of the type and size of problems it can handle by studying its result on a more complex case study. It has been shown that the operational semantics of any programming language can be faithfully encoded by rewrite rules of the form $l \Rightarrow r$ `if` b ([32]). If we encode the operational semantics as a CTRS, then a constrained term represents a symbolic program. Proving partial correctness of a program amounts to proving a reachability property for the respective constrained term.

To demonstrate this capability, we have encoded the operational semantics of a simple imperative language `IMP` as a CTRS. Because of space limitations, we cannot include its definition, which is available on the web page of the tool. The definition of `IMP` consists of 14 sorts, 7 subsorting relations, 48 function symbols including built-ins and 45 rewrite rules. The following term is a symbolic `IMP` program that computes the sum of the first N natural numbers:

```
1 I(push(seq(assign(x, N), seq(assign(y, mzero), seq(assign(z, mzero),
            while(le(y, x), seq(assign(z, plus(z, y)),
                                assign(y, plus(y, mone)))))), done), emp)
```

The function symbol `I` constructs `IMP` configurations which consist of a stack of code and an environment. There is a single element on the stack consisting of the program to execute (`push(·, done)`) and the environment is empty (`emp`). We can prove that the program indeed computes the sum of the first N natural numbers by using the `prove` command with a single helper circularity, which encodes the invariant of the `while` loop.

The proof succeeds with a wall time of 33 seconds on a typical laptop. Almost the entire execution time is spent calling the SMT solver. The proof tree of the second circularity reaches a depth of 103 proof steps.

Reachability properties with existential quantifiers. At the start of the article, we have promised that our tool can establish properties of the form:

$$\forall \tilde{x}.(c_1(\tilde{x}) \rightarrow \exists \tilde{y}.(c_2(\tilde{x}, \tilde{y}) \wedge t_1(\tilde{x}) \Rrightarrow^*_{\mathcal{R}} t_2(\tilde{x}, \tilde{y}))). \tag{3}$$

However, `RMT` only implements rewrite rules of the form $u \Rightarrow v$ `if` b, which intuitively correspond to the first-order formula $\forall \tilde{x}.(b(\tilde{x}) \rightarrow u(\tilde{x}) \Rightarrow_{\mathcal{R}} v(\tilde{x}))$.

In order to establish properties such as the one in Equation (3), we make the following auxiliary construction: we consider a fresh distinguished symbol ok and we add to $\mathcal{R}$ the following rewrite rule: $\mathcal{R}' = \mathcal{R} \cup \{t_2(\tilde{x}, \tilde{y}) \Rightarrow ok(\tilde{x}) \ \ \texttt{if} \ \ c_2(\tilde{x}, \tilde{y})\}$. We then have that the formula in Equation (3) holds iff $\mathcal{R}' \models t_1(\tilde{x}) \Rrightarrow^+ ok(\tilde{x}) \ \ \texttt{if} \ \ c_1(\tilde{x})$. We use this encoding in order to prove the correctness of the extended algorithm of Euclid in Section 1.

Discussion and Further Work

In this article, we have presented `RMT`, a tool for proving reachability properties in constrained term rewriting systems. `RMT` is available at `http://profs.info.uaic.ro/~stefan.ciobaca/rmt`, along with all examples used in this article. The procedure implemented in `RMT` for establishing reachability properties is based on proof systems developed in our previous work ([22,7]) and it is sound for terminating paths. We developed `RMT` to fill a gap in the set of tools for CTRSs: there are tools that establish termination, confluence and equivalence of terms, but no tools for reachability, except for specialized rewrite systems occurring in programming languages research ([8]). We want `RMT` to be easy to use and simple to incorporate by the research community into other applications. Some of the code in `RMT`, in particular the data structures used for terms, is an expanded version of previous code written by the first author ([5]) for security protocol analysis.

Related work. The best tool for everything related to rewriting is Maude ([10]). Recently, Rocha and others ([26]) extended Maude to provide rewriting modulo SMT. This extension makes Maude suitable for the same kind of rewriting techniques described in this article. Unfortunately, rewriting modulo SMT in Maude is only implemented for *topmost rewrite theories*. Almost any theory can be written as a topmost

theory ([23]), but in our experience the encoding can significantly increase the number of transitions, which produces performance concerns. For example, in our CTRLs for the IMP language, the depth of the proof tree for proving the sum program is significantly lower in IMP than in topmost IMP.

The K framework ([31]) by Roşu and others is the best tool to define the semantics of real programming languages as a rewrite theory. It has been used, for example, to define the complete rewriting-based operational semantics of the languages C ([15]), Java ([4]) and JavaScript ([25]). Recently, the same proof systems ([22,7]) that we have implemented for reachability proofs in CTRSs has been implemented in K ([8]) and used to prove a number of programs written in several languages. The main difference is that the implementation in [8] is tailored for the operational semantics of languages, with special cases for constructs that occur frequently in programming languages such as heaps. However, our tool is aimed at a more general public as it is meant to be used for any CTRS, even if it would not perform as well on rewrite systems defining operational semantics; it is easier to install because it has fewer dependencies (only `Z3`) and it feels more robust. Additionally, the algorithm in [8] contains a small source of incompleteness, as when proving a reachability property it is either discharged completely through implication or through circularities/rewrite rules. We allow a reachability rule to be discharged partially by implication and partially by reachability/rewrites, as explained in Algorithm 2. However, `RMT` is also incomplete, since it does not fully implement the relatively complete proof system in [7].

Reachability in rewriting is explored in depth in [11]. The work by Kirchner and others ([16]) is the first to propose the use of rewriting with symbolic constraints for deduction. Subsequent work ([26,18]) extends and unifies previous approaches to rewriting with constraints. The section on related work in [26] contains a comprehensive account of literature related to rewriting modulo constraints.

Future work. For future work, we would like to overcome several engineering and research challenges. On the engineering side, we would like to speed up the tool by interacting better with the SMT solver and by using term indexing techniques, but also to ease reachability proofs by providing an interactive proof environment. We will experiment with other SMT solvers such as `CVC4` [2], for which we already have a compile-time option, but which does not perform as well as `Z3`, especially because of its weak support for non-linear arithmetic. We would also like to integrate with the Maude rewrite tool to access its rewriting capabilities and with the K tool in order to access realistic language definitions. On the research side, we would like to add user defined functions, overcome the technical limitations that we have imposed on the built-in signature, allow quantifiers in the constraints, use rewriting modulo AC and implement out proof system for equivalence ([6]). We would also like to extract from `RMT` a well-documented and fast API for unification and rewriting modulo constraints, which could be used to quickly prototype research tools based on symbolic rewriting.

References

1. Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. 2001.
2. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV 2011*, pages 171–177, 2011.
3. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
4. Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *POPL 2015*, pages 445–456, 2015.
5. Ştefan Ciobâcă, Stéphanie Delaune, and Steve Kremer. Computing Knowledge in Security Protocols under Convergent Equational Theories. In *CADE 2009*, pages 355–370, 2009.
6. Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Roşu. A Language-Independent Proof System for Full Program Equivalence. *Formal Aspects of Computing*, 28(3):469–497, 2016.
7. Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *RTA-TLCA 2014*, volume 8560 of *LNCS*, pages 425–440, 2014.
8. Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *OOPSLA 2016*, to appear.
9. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS 2008*, pages 337–340, 2008.
10. Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Built-in Variant Generation and Unification, and Their Applications in Maude 2.7. In *IJCAR 2016*, pages 183–192, 2016.
11. Santiago Escobar, José Meseguer, and Prasanna Thati. Narrowing and Rewriting Logic: from Foundations to Applications. *Electronic Notes in Theoretical Computer Science*, 177:5 – 33, 2007.

12. Stephan Falke and Deepak Kapur. A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs. In *CADE 2009*, pages 277–293, 2009.
13. Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving Termination of Programs Automatically with AProVE. In *IJCAR 2014*, pages 184–191, 2014.
14. Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):1–39, 2011.
15. Chris Hathhorn, Chucky Ellison, and Grigore Rosu. Defining the Undefinedness of C. In *PLDI 2015*, pages 336–345, 2015.
16. Claude Kirchner, Helene Kirchner, and Michael Rusinowitch. Deduction with Symbolic Constraints. Technical Report RR-1358, INRIA, 1990.
17. Cynthia Kop. Termination of LCTRSs. *CoRR*, abs/1601.03206, 2016.
18. Cynthia Kop and Naoki Nishida. Term Rewriting with Logical Constraints. In *FroCoS 2013*, pages 343–358, 2013.
19. Cynthia Kop and Naoki Nishida. Automatic Constrained Rewriting Induction towards Verifying Procedural Programs. In *APLAS 2014*, pages 334–353, 2014.
20. Cynthia Kop and Naoki Nishida. Constrained Term Rewriting tooL. In *LPAR 2015*, pages 549–557, 2015.
21. Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The Vampire and the FOOL. In *CPP 2016*, pages 37–48, 2016.
22. Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. A Generic Framework for Symbolic Execution: A Coinductive Approach. *Journal of Symbolic Computation*, pages –, 2016.
23. José Meseguer and Prasanna Thati. Symbolic Reachability Analysis Using Narrowing and Its Application to Verification of Cryptographic Protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
24. Naoki Nakabayashi, Naoki Nishida, Keiichirou Kusakari, Toshiki Sakabe, and Masahiko Sakai. Lemma Generation Method in Rewriting Induction for Constrained Term Rewriting Systems. *Computer Software*, 28(1):173–189, 2010. (original article in Japanese, translated version available online: http://www.trs.cm.is.nagoya-u.ac.jp/crisys/nakabayashi10.pdf).
25. Daejun Park, Andrei Ştefanescu, and Grigore Roşu. KJS: a Complete Formal Semantics of JavaScript. In *PLDI 2015*, pages 346–356, 2015.
26. Camilo Rocha, José Meseguer, and César Muñoz. Rewriting Modulo SMT and Open System Analysis. *Journal of Logical and Algebraic Methods in Programming*, to appear, 2016.
27. Grigore Roşu. Matching logic — extended abstract. In *RTA 2015*, volume 36 of *LIPIcs*, pages 5–21, 2015.
28. Grigore Roşu and Andrei Ştefănescu. From Hoare Logic to Matching Logic Reachability. In *FM 2012*, volume 7436 of *LNCS*, pages 387–402, 2012.
29. Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS 2013*, pages 358–367, 2013.
30. Grigore Roşu and Dorel Lucanu. Circular Coinduction – A Proof Theoretical Foundation. In *CALCO 2009*, pages 127–144, 2009.
31. Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
32. Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A Rewriting Logic Approach to Operational Semantics. *Information and Computation*, 207(2):305–340, 2009.