Rewriting Modulo SMT and Open System Analysis

Camilo Rocha[1], José Meseguer[2], and César Muñoz[3]

DECC, Pontificia Universidad Javeriana, Cali, Colombia
Computer Science Department, University of Illinois, Urbana IL, USA
NASA Langley Research Center, NASA, Hampton VA, USA

Abstract. This paper proposes *rewriting modulo SMT*, a new technique that combines the power of SMT solving, rewriting modulo theories, and model checking. Rewriting modulo SMT is ideally suited to model and analyze reachability properties of infinite-state *open systems*, i.e., systems that interact with a nondeterministic environment. Such systems exhibit both internal nondeterminism, which is proper to the system, and external nondeterminism, which is due to the environment. In a reflective formalism, such as rewriting logic, rewriting modulo SMT can be reduced to standard rewriting. Hence, rewriting modulo SMT naturally extends rewriting-based reachability analysis techniques, which are available for closed systems, to open systems. The proposed technique is illustrated with the formal analysis of: (i) a real-time system that is beyond the scope of timed-automata methods and (ii) automatic detection of reachability violations in a synchronous language developed to support autonomous spacecraft operations.

1 Introduction

Symbolic techniques can be used to represent possibly infinite sets of states by means of symbolic constraints. These techniques have been developed and adapted to many other verification methods such as SAT solving, Satisfiability Modulo Theories (SMT), rewriting, and model checking. A key open research issue of current symbolic techniques is extensibility. Techniques that combine different methods have been proposed, e.g., decision procedures [50, 51], unifications algorithms [7, 11], theorem provers with decision procedures [1, 10, 53], and SMT solvers in model checkers [3, 30, 49, 62, 66]. However, there is still a lack of general extensibility techniques for symbolic analysis that simultaneously combine the power of SMT solving, rewriting- and narrowing-based analysis, and model checking.

This paper proposes a new symbolic technique that seamlessly combines rewriting modulo theories, SMT solving, and model checking. For brevity, this technique is called *rewriting modulo SMT*, although it could more precisely be called *rewriting modulo SMT+B*, where B is an equational theory having a matching algorithm. It complements another symbolic technique combining narrowing modulo theories and model checking, namely narrowing-based reachability analysis [8, 48]. Neither of these two techniques subsumes the other. Indeed, each technique has specific advantages: narrowing-based reachability analysis is more general and can perform more powerful forms of symbolic execution based on narrowing, as opposed to matching and rewriting. But rewriting modulo SMT can verify both satisfiability and validity of constraints

in the decidable built-in subtheory, and can be considerably more efficient because of the higher efficiency of matching versus unification modulo a set of axioms.

In rewriting logic [46], deterministic systems can be naturally specified by equational theories, but specification of concurrent, nondeterministic systems in rewriting logic requires rewrite theories, i.e., triples $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an equational theory describing system states as elements of the initial algebra $\mathcal{T}_{\Sigma/E}$, and R rewrite rules describing the system's local concurrent transitions. Rewriting modulo SMT techniques can then be applied to increase the power of rewrite-based equational reasoning for (Σ, E) such as, for instance, inductive theorem proving [29, 39, 40], termination checking [28, 61], and procedural verification [41]. However, the full power of rewriting modulo SMT, including its model checking capabilities, can be better exploited when applied to concurrent open systems.

An *open system* is a concurrent system that interacts with an external, nondeterministic environment. When such a system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, it has two sources of nondeterminism, one internal and the other external. Internal nondeterminism comes from the fact that in a given system state several instances of each rule in R may be enabled. The local transitions thus enabled may lead to completely different states. What is peculiar about an open system is that it also has external, and often infinitely-branching, nondeterminism due to the environment. That is, the state of an open system must include the state changes due to the environment. Technically, this means that, while system transitions in a closed system can be described by rewrite rules of the form $t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x})$, transitions in an open system can instead be modeled by rules of the form $t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y})$, where the extra variables $\overrightarrow{y}$ on the right-hand side of the rule are fresh new variables that can represent external nondeterminism such as, for instance, user input, sensor probing, and random computations. Therefore, a substitution for the variables $\overrightarrow{x} \uplus \overrightarrow{y}$ decomposes into two substitutions, one, say θ, for the variables $\overrightarrow{x}$ under the control of the system and another, say ρ, for the variables $\overrightarrow{y}$ under the control of the environment. In rewriting modulo SMT, such open systems are described by conditional rewrite rules of the form $t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y})$ **if** $\phi(\overrightarrow{x}, \overrightarrow{y})$, where ϕ is a constraint solvable by an SMT solver. This constraint ϕ may still allow the environment to choose an infinite number of substitutions ρ for $\overrightarrow{y}$, but can exclude choices that the environment will never make.

Consider the example of a real-time system comprising a thermostat sensing the temperature from the environment and an air conditioning device. The goal of the thermostat and the air conditioning is to maintain the system's temperature near a desired *setpoint*. The thermostat does this by switching the air conditioning device on or off, depending on the relation between the temperature it senses from the environment and the setpoint. For this example, the state of the system can be modeled by tuples of the form

$$\langle\, time : \_\,, temp : \_\,, setpoint : \_\,, ac : \_\, \rangle \quad \text{or} \quad [\, time : \_\,, temp : \_\,, setpoint : \_\,, ac : \_\,]$$

where the values associated to attributes *time*, *temp*, and *setpoint* are natural numbers specifying, respectively, the system's global clock, the temperature sensed by the thermostat from the environment, and the system's desired setpoint, and the value associated to attribute *ac* is a Boolean constant indicating whether the air conditioning device is turned on or not. The idea is that the first two attributes in a state (i.e., *time* and *temp*)

are the ones under control of the environment, while the last two attributes (i.e., *setpoint* and *ac*) are the ones under the internal control of the system. The state transitions can be modeled by the following three rules, with R, S, T, T_e natural number variables and B a Boolean variable:

$$\langle\, time : R, temp : T, setpoint : S, ac : B \,\rangle \rightarrow [\, time : R, temp : T, setpoint : S, ac : true \,] \textbf{ if } T > S$$
$$\langle\, time : R, temp : T, setpoint : S, ac : B \,\rangle \rightarrow [\, time : R, temp : T, setpoint : S, ac : false \,] \textbf{ if } T \leq S$$
$$[\, time : R, temp : T, setpoint : S, ac : B \,] \rightarrow \langle\, time : R + 1, temp : T_e, setpoint : S, ac : B \,\rangle$$

The first rule models the situation in which the temperature sensed from the environment exceeds the setpoint of the system and, thus, the air conditioning device must be turned on (if it was not). The second rule models the opposite situation in which the temperature sensed from the environment does not exceed the setpoint of the system and the air conditioning device must be turned off (if it was not). The third rule is a tick rule modeling the passage of time and the corresponding changes in the environment, namely, the global clock is increased by one time unit and the next value of the temperature is read from a sensor. The interplay between states of the form $\langle \_ \rangle$ and $[\_]$ can be explained as follows: rules under the internal control of the system are exclusively applicable to a state of the form $\langle \_ \rangle$, producing a state of the form $[\_]$ in a zero-time transition, while the rule under the control of the external environment is only applicable to a state of the form $[\_]$, producing a state of the form $\langle \_ \rangle$ in one time unit. In the above-stated sense, this is an example of an open and concurrent system that interacts with an external nondeterministic environment: the extra variable T_e in the right-hand side of the last rule represents the external nondeterminism due to changes in the environment. Note that this system cannot be directly executed via term rewriting because there are infinitely many substitutions for T_e.

The non-trivial challenges of modeling and analyzing open systems can now be better explained. They include: (1) the enormous and possibly infinitary nondeterminism due to the environment, which typically renders finite-state model checking impossible or unfeasible; (2) the impossibility of executing the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ in the standard sense, due to the nondeterministic choice of ρ; and (3) the, in general, undecidable challenge of checking the rule's condition ϕ, since without knowing ρ, the condition $\phi\theta$ is non-ground, so that its E-satisfiability may be undecidable. As further explained in the paper, challenges (1)–(3) are all met successfully by rewriting modulo SMT because: (1) states are represented not as concrete states, i.e., ground terms, but as symbolic constrained terms $(t\,;\varphi)$ with t a term with variables ranging over the domains handled by the SMT solver and φ an SMT-solvable formula, so that the choice of ρ is avoided; (2) rewriting modulo SMT can symbolically rewrite such pairs $(t\,;\varphi)$ (describing possibly infinite sets of concrete states) to other pairs $(t'\,;\varphi')$; and (3) decidability of $\phi\theta$ (more precisely of $\varphi \wedge \phi\theta$) can be settled by invoking an SMT solver. In this sense, rewriting modulo SMT is a symbolic reachability analysis technique for *topmost* rewrite theories, i.e., rewrite theories for which nondeterministic computation specified by the rules happens at the top of the state terms. Many systems whose state can be represented by a set or multiset of objects and messages can be naturally specified as topmost rewrite theory. In particular, most distributed systems and network protocols, including those with real-time features, can be easily modeled this way. This rewriting-based system

specification style is illustrated with the above thermostat example —which is fully specified later in the paper— and with two longer case studies.

Rewriting modulo SMT can be integrated with model-checking by exploiting the fact that rewriting logic is reflective [20], i.e., a logic in which important aspects of its metatheory can be represented at the object level in a consistent way. Hence, rewriting modulo SMT can be reduced to standard rewriting. In particular, all the techniques, algorithms, and tools available for model checking of closed systems specified as rewrite theories, such as Maude's search-based reachability analysis [19], become directly available to perform symbolic reachability analysis on systems that are now infinite-state.

The approach and formal analysis techniques proposed in this paper are illustrated with the formal analysis of the CASH scheduling protocol [15] and formal executable semantics of the Plan Execution Interchange Language (PLEXIL) [26]. The CASH protocol specifies a real-time system whose formal analysis is beyond the scope of timed-automata [2]. The language PLEXIL is a safety-critical synchronous language developed by NASA to support autonomous spacecraft operations.

This paper is an extended and revised version of [58], including:

- A running example, namely, the real-time and open system comprising the thermostat and the air conditioning device, that is used for the purpose of explaining concepts, definitions, and results in sections 3, 4, and 5.
- Complete proofs of all results in sections 3, 4, and 5.
- A more comprehensive description of the CASH protocol case study in Section 7, in which all transitions rules are included and explained.
- A new case study in Section 8 on automatically detecting symbolic reachability violations for PLEXIL.

The rest of this paper is organized as follows. Section 2 presents some preliminary material on rewriting logic. Section 3 introduces built-in subtheories and elaborates on some of its equational properties. Section 4 presents the concept of rewriting modulo a built-in equational sub-theory. Section 5 then explains how to perform sound and complete symbolic rewriting with these theories. Section 6 presents an overview of a reflective implementation in Maude that offers support for symbolic rewriting modulo SMT. Sections 7 and 8 present case studies, respectively, on the symbolic reachability analysis for the CASH real-time protocol and the plan execution language PLEXIL. Finally, Section 9 presents some concluding remarks.

2 Preliminaries

Notation on terms, term algebras, and equational theories is used as in [6,36]. An *order-sorted signature* Σ is a tuple $\Sigma=(S,\leq,F)$ with a finite poset of sorts $(S,\leq)$ and set of function symbols F typed with sorts in S. The binary relation $\equiv_{\leq}$ denotes the equivalence relation $(\leq \cup \geq)^+$ generated by $\leq$ on S and its point-wise extension to strings in S^*. The function symbols in F can be subsort-overloaded. To avoid ambiguous parses they are required to satisfy the condition that, for $w, w' \in S^*$ and $s, s' \in S$, if $f : w \longrightarrow s$

and $f : w' \longrightarrow s'$ are in F, then $w \equiv_{\leq} w'$ implies $s \equiv_{\leq} s'$. For any sort $s \in S$, the expression $[s]$ denotes the connected component of s, that is, $[s] = [s]_{\equiv_{\leq}}$. A *top sort* in Σ is a sort $s \in S$ such that for all $s' \in [s]$, $s' \leq s$.

Let $X = \{X_s\}_{s \in S}$ denote an S-indexed family of disjoint variable sets with each X_s countably infinite. The *set of terms of sort s* and the *set of ground terms of sort s* are denoted, respectively, by $T_\Sigma(X)_s$ and $T_{\Sigma,s}$; similarly, $T_\Sigma(X)$ and T_Σ denote, respectively, the set of terms and the set of ground terms. $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the corresponding order-sorted Σ-term algebras. All order-sorted signatures are assumed *preregular* [36], i.e., each Σ-term t has a unique *least sort* $ls(t) \in S$ s.t. $t \in T_\Sigma(X)_{ls(t)}$. It is also assumed that Σ has *nonempty sorts*, i.e., $T_{\Sigma,s} \neq \emptyset$ for each $s \in S$. The *set of variables* of t is written $vars(t)$ and for a list of terms $t_0, t_1, \ldots, t_n$, $vars(t_1, \ldots, t_n) = vars(t_0) \cup \cdots \cup vars(t_n)$. For a term $t \in T_\Sigma(X)$ and $\overrightarrow{x}$ a list of variables in X, the expression $t(\overrightarrow{x})$ denotes the term t and the fact that each variable in $vars(t)$ occurs in the list $\overrightarrow{x}$. A term t is called *linear* if and only if each $x \in vars(t)$ occurs only once in t. For $S' \subseteq S$, a term is called *S'-linear* if and only if each $x \in vars(t)$ with sort in S' occurs only once in t.

A *substitution* is an S-indexed mapping $\theta : X \longrightarrow T_\Sigma(X)$ that is different from the identity only for a finite subset of X and such that $\theta(x) \in T_\Sigma(X)_s$ if $x \in X_s$, for any $x \in X$ and $s \in S$. The identity substitution is denoted by id and $\theta|_Y$ denotes the restriction of θ to a family of variables $Y \subseteq X$. The domain of θ, denoted $dom(\theta)$, is the subfamily of X for which $\theta(x) \neq x$, and $ran(\theta)$ denotes the family of variables introduced by the terms $\theta(x)$, such that $x \in dom(\theta)$. Substitutions extend homomorphically to terms in the natural way. A substitution θ is called *ground* if and only if $ran(\theta) = \emptyset$. The application of a substitution θ to a term t is denoted by $t\theta$ and the composition (in diagrammatic order) of two substitutions θ_1 and θ_2 is denoted by $\theta_1\theta_2$, so that $t\theta_1\theta_2$ denotes $(t\theta_1)\theta_2$. A *context* C is a λ-term of the form $C = \lambda x_1, \ldots, x_n.c$ with $c \in T_\Sigma(X)$ and $\{x_1, \ldots, x_n\} \subseteq vars(c)$; it can be viewed as an n-ary function $(t_1, \ldots, t_n) \mapsto C(t_1, \ldots, t_n) = c\theta$, where $\theta(x_i) = t_i$ for $1 \leq i \leq n$ and $\theta(x) = x$ otherwise.

A *Σ-equation* is an unoriented pair $t = u$ with $t \in T_\Sigma(X)_s$, $u \in T_\Sigma(X)_{s'}$, and $s \equiv_{\leq} s'$. A *conditional Σ-equation* is a triple (t, u, γ), denoted $t = u$ **if** γ, with $t = u$ a Σ-equation and γ a finite conjunction of Σ-equations; Σ-equations and conditional Σ-equations will be just called Σ-equations for brevity. An *equational theory* is a tuple (Σ, E), with Σ an order-sorted signature and E a finite collection of (possibly conditional) Σ-equations. An equational theory $\mathcal{E} = (\Sigma, E)$ induces the congruence relation $=_\mathcal{E}$ on $T_\Sigma(X)$ defined for $t, u \in T_\Sigma(X)$ by $t =_\mathcal{E} u$ if and only if $\mathcal{E} \vdash t = u$, where $\mathcal{E} \vdash t = u$ denotes $\mathcal{E}$-provability by the deduction rules for order-sorted equational logic in [47]. For the purpose of this paper, such inference rules, which are analogous to those of many-sorted equational logic, are even simpler thanks to the assumption that Σ has nonempty sorts, which makes unnecessary the explicit treatment of universal quantifiers. Similarly, $=^1_\mathcal{E}$ denotes provable $\mathcal{E}$-equality in *one step* of deduction. The *$\mathcal{E}$-subsumption* ordering $\ll_\mathcal{E}$ is the binary relation on $T_\Sigma(X)$ defined for any $t, u \in T_\Sigma(X)$ by $t \ll_\mathcal{E} u$ if and only if there is a substitution $\theta : X \longrightarrow T_\Sigma(X)$ such that $t =_\mathcal{E} u\theta$. The expressions $\mathcal{T}_\mathcal{E}(X)$ and $\mathcal{T}_\mathcal{E}$ (also written $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$) denote the quotient algebras induced by $=_\mathcal{E}$ on the term algebras $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$, respectively. $\mathcal{T}_{\Sigma/E}$ is called the *initial algebra* of (Σ, E). A theory inclusion $(\Sigma, E) \subseteq (\Sigma', E')$, with $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$, is called *protecting*

if and only if the unique Σ-homomorphism $\mathcal{T}_{\Sigma/E} \longrightarrow \mathcal{T}_{\Sigma'/E'}|_\Sigma$ to the Σ-*reduct*[1] *of the initial algebra* $\mathcal{T}_{\Sigma'/E'}$ is a Σ-isomorphism, written $\mathcal{T}_{\Sigma/E} \simeq \mathcal{T}_{\Sigma'/E'}|_\Sigma$. Intuitively, if a theory inclusion $(\Sigma, E) \subseteq (\Sigma', E')$ is protecting, this means that: (i) $=_{E'}$ does not identify terms in $T_\Sigma(X)$ that cannot be proved equal by $=_E$, that is, *no confusion* is added when extending (Σ, E) to (Σ', E'), and (ii) for each sort s of Σ and $t' \in T_{\Sigma',s}$ there is a $t \in T_{\Sigma,s}$ such that $t' =_{E'} t$, that is, *no junk* is added to the sorts of Σ when extending (Σ, E) to (Σ', E').

A set of equations E is called *regular* (resp., *linear*) if and only if for any equation $(t = u \textbf{ if } \gamma) \in E$ $vars(t) = vars(u)$ (resp., both t and u are linear terms). Moreover, E is called *collapse-free* for a subset of sorts $S' \subseteq S$ if and only if for any $t = u \in E$ and for any substitution $\theta : X \longrightarrow T_\Sigma(X)$ neither $t\theta$ nor $u\theta$ map to a variable having some sort $s \in S'$. In this paper, intuitively, regularity assumptions are used to prevent some equations from adding/dropping variables and linearity assumptions are used to forbid the use of matching for denoting equality among terms; both of this assumptions are crucial for proving completeness of the proposed approach. Furthermore, by excluding collapsing equations for the relevant subset of sorts, it becomes impossible to identify those terms that are directly handled by SMT solving and those that are reduced by rewriting.

A Σ-*rewrite rule* is a triple $l \rightarrow r$ **if** ϕ, with $l, r \in T_\Sigma(X)_s$, for some sort $s \in S$, and $\phi = \bigwedge_{i \in I} t_i = u_i$ a finite conjunction of Σ-equations. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an order-sorted equational theory and R a finite set of Σ-rules. $\mathcal{R} = (\Sigma, E, R)$ is called a a *topmost rewrite theory* if it has a top sort *State* such that no operator in Σ has *State* as argument sort and each rule $l \rightarrow r$ **if** $\phi \in R$ satisfies $l, r \in T_\Sigma(X)_{State}$ and $l \notin X$. A rewrite theory $\mathcal{R}$ induces a rewrite relation $\rightarrow_\mathcal{R}$ on $T_\Sigma(X)$ defined for every $t, u \in T_\Sigma(X)$ by $t \rightarrow_\mathcal{R} u$ if and only if there is a rule $(l \rightarrow r \textbf{ if } \phi) \in R$ and a substitution $\theta : X \longrightarrow T_\Sigma(X)$ satisfying $t =_E l\theta$, $u =_E r\theta$, and $E \vdash \phi\theta$. The tuple $\mathcal{T}_\mathcal{R} = (\mathcal{T}_{\Sigma/E}, \rightarrow^*_\mathcal{R})$ is called the *initial reachability model of* $\mathcal{R}$ [14].

Appropriate requirements are needed to make an equational theory $\mathcal{E}$ *admissible*, i.e., *executable* in rewriting languages such as Maude [19]. In this paper, it is assumed that the equations of $\mathcal{E}$ can be decomposed into a disjoint union $E \uplus B$, with B a collection of regular and linear structural axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a *matching algorithm modulo* B producing a finite number of B-matching solutions, or failing otherwise. Furthermore, it is assumed that the equations E can be oriented into a set of (possibly conditional) sort-decreasing, operationally terminating, confluent rewrite rules $\overrightarrow{E}$ modulo B. The rewrite system $\overrightarrow{E}$ is *sort decreasing* modulo B if and only if for each $(t \rightarrow u \textbf{ if } \gamma) \in \overrightarrow{E}$ and substitution θ, $ls(t\theta) \geq ls(u\theta)$ if $(\Sigma, B, \overrightarrow{E}) \vdash \gamma\theta$. The system $\overrightarrow{E}$ is *operationally terminating* modulo B [25] if and only if there is no infinite well-formed proof tree in $(\Sigma, B, \overrightarrow{E})$ (see [45] for terminology and details). Furthermore, $\overrightarrow{E}$ is *confluent* modulo B if and only if for all $t, t_1, t_2 \in T_\Sigma(X)$, if $t \rightarrow^*_{E/B} t_1$ and $t \rightarrow^*_{E/B} t_2$, then there is $u \in T_\Sigma(X)$ such that $t_1 \rightarrow^*_{E/B} u$ and $t_2 \rightarrow^*_{E/B} u$. The term $t\downarrow_{E/B} \in T_\Sigma(X)$ denotes the *E-canonical form* of t modulo B so that $t \rightarrow^*_{E/B} t\downarrow_{E/B}$ and $t\downarrow_{E/B}$ cannot be further reduced by $\rightarrow_{E/B}$. Under

[1] For $\Sigma \subseteq \Sigma'$ a subsignature inclusion and $\mathcal{A}$ a Σ'-algebra, its Σ-*reduct* $\mathcal{A}|_\Sigma$ is the Σ-algebra obtained from $\mathcal{A}$ by ignoring all operators and sorts in $\Sigma' \setminus \Sigma$.

sort-decreasingness, operational termination, and confluence, the term $t\downarrow_{E/B}$ is unique up to B-equality.

For a rewrite theory $\mathcal{R}$, the rewrite relation $\rightarrow_{\mathcal{R}}$ is undecidable in general, even if its underlying equational theory is admissible, unless conditions such as coherence [65] are given (i.e, whenever rewriting with $\rightarrow_{R/E\cup B}$ can be decomposed into rewriting with $\rightarrow_{E/B}$ and $\rightarrow_{R/B}$). A key goal of this paper is to make such a relation both decidable and symbolically executable when $\mathcal{R}$ is topmost and $\mathcal{E}$ decomposes as $\mathcal{E}_0 \uplus B_1$, where $\mathcal{E}_0$ is a built-in theory for which formula satisfiability is decidable and B_1 has a matching algorithm.

3 Built-in Subtheories

For the purpose of rewriting modulo SMT, a built-in subtheory corresponds to the portion of the equational theory that will be handled by the SMT solver. The goal of this section is twofold. On the one hand, it introduces the concept of built-in subtheory, key for defining rewriting modulo built-in subtheories in Section 4. On the other hand, it presents some equational properties of these theories that are useful in proving the main theorems of this paper in Section 5.

The signature of a built-in subtheory defines the sorts and the function symbols that the SMT solver can handle.

Definition 1 (Signature with Built-ins). *An order-sorted signature $\Sigma = (S, \leq, F)$ is a signature with* built-in subsignature $\Sigma_0 \subseteq \Sigma$ *if and only if $\Sigma_0 = (S_0, F_0)$ is many-sorted, for each $s \in S_0$ its connected component $[s]$ in $(S, \leq)$ is the singleton set $[s] = \{s\}$, and, for $F_1 = F \backslash F_0$, if $f : w \longrightarrow s \in F_1$, then $s \notin S_0$ and f has no other (subsort-overloaded) typing in F_0.*

The notion of built-in subsignature in an order-sorted signature Σ is modeled by a many-sorted signature Σ_0 defining the built-in terms $T_{\Sigma_0}(X_0)$. The restriction imposed on the sorts and the function symbols in Σ w.r.t. Σ_0 provides a clear syntactic distinction between built-in terms (the only ones with built-in sorts) and all other terms.

Example 1. Consider the example of the open system comprising a thermostat and an air conditioning device from the Introduction. This example is used here to illustrate the definition of a built-in subsignature (S_0, F_0) of an order-sorted signature $(S, \leq, F)$. Later on it will also be used to illustrate rewriting modulo SMT and its use in system analysis. In this system, the set of sorts S is defined by:

$$S = \{Nat, Bool, Attribute, AttrSet, State\},$$

where *Nat* is the sort of natural numbers, *Bool* the sort of Boolean values, *Attribute* the sort of attributes, *AttrSet* the sort of multisets of attributes, and *State* the topsort representing the system's states. The partial order $\leq$ on S is the set:

$$\leq = \{(s, s) \mid s \in S\} \cup \{(Attribute, AttrSet)\}.$$

The set of function symbols F is given by the following definitions:

$$
\begin{array}{ll}
0 : \longrightarrow \mathit{Nat} & s : \mathit{Nat} \longrightarrow \mathit{Nat} \\
\mathit{true} : \longrightarrow \mathit{Bool} & \mathit{false} : \longrightarrow \mathit{Bool} \\
\_ < \_ : \mathit{Nat} \times \mathit{Nat} \longrightarrow \mathit{Bool} & \_ \leq \_ : \mathit{Nat} \times \mathit{Nat} \longrightarrow \mathit{Bool} \\
\mathit{time} : \_ : \mathit{Nat} \longrightarrow \mathit{Attribute} & \mathit{temp} : \_ : \mathit{Nat} \longrightarrow \mathit{Attribute} \\
\mathit{setpoint} : \_ : \mathit{Nat} \longrightarrow \mathit{Attribute} & \mathit{ac} : \_ : \mathit{Bool} \longrightarrow \mathit{Attribute} \\
\mathit{mt} : \longrightarrow \mathit{AttrSet} & \_ , \_ : \mathit{AttrSet} \times \mathit{AttrSet} \longrightarrow \mathit{AttrSet} \\
\langle \_ \rangle : \mathit{AttrSet} \longrightarrow \mathit{State} & [\_] : \mathit{AttrSet} \longrightarrow \mathit{State}.
\end{array}
$$

Natural numbers are represented in Peano-like notation with constant 0 and successor function s, Boolean values are represented by constants *true* and *false*, and operators $<$ and $\leq$ are used to compare natural numbers in the usual way. Tokens *time*, *temp*, *setpoint*, and *ac* are used as attribute names, with the former three taking a natural number as a parameter and the latter one a Boolean value. Token *mt* is used to represent the empty multiset of attributes, while the union of (multisets) of attributes is denoted by comma. Notice that by the subsort inclusion $\mathit{Attribute} \leq \mathit{AttrSet}$, any attribute is a singleton multiset of attributes. Finally, a state is formed by enclosing a multiset of attributes in angular braces or square brackets.

The built-in subsignature Σ_0 is defined to be that of the natural numbers and the Boolean values, namely $\Sigma_0 = (S_0, F_0)$ is defined to be

$$
S_0 = \{\mathit{Nat}, \mathit{Bool}\}, \qquad F_0 = \{0, s, \mathit{true}, \mathit{false}, <, \leq\}.
$$

There are a few extra things to be said about the signature presented in Example 1. First, note that some usual function symbols for *Nat* and *Bool* have not been included as part of the signature. Second, for *AttrSet* to correctly model multisets of attributes, some axioms such as associativity, commutativity, and identity for ',' need to be included. Additional function symbols and equational axioms for this specification will be gradually added later. Finally, observe that the Peano-like specification of the natural numbers presented in this signature can be quite problematic when dealing with large numbers. However, in rewriting logic specification languages such as Maude [19], and in state-of-the-art SMT solvers, there is no need to explicitly define natural numbers in Peano-like notation: instead, natural numbers can be specified in decimal notation and $s(n)$ can be denoted as $n + 1$. Here, the Peano notation is used to better illustrate the difference between built-in function symbols and the rest of the function symbols, but actually a possibly infinite signature Σ_0 of built-in symbols could be used.

Since the goal of rewriting modulo SMT is to achieve conditional symbolic rewriting with decidable built-in constraints by delegating to an SMT solver the handling of such constraints on built-in terms, it is important to have a mechanism for completely 'hiding' the syntactic details of built-in terms from the rewrite relation. In this paper, this idea of hiding the structure of built-in terms from a term is captured by the notion of abstraction of built-ins, a mechanism for replacing each one of the maximal built-in subterms of a term by distinct fresh new variables.

Definition 2. *If $\Sigma \supseteq \Sigma_0$ is a signature with built-ins, then an* abstraction of built-ins *for a Σ-term t is a pair $(\lambda x_1 \cdots x_n.t^\circ\,;\theta^\circ)$ consisting of context $\lambda x_1 \cdots x_n.t^\circ$ and a substitution $\theta^\circ : X_0 \longrightarrow T_{\Sigma_0}(X_0)$ such that: (i) $t^\circ \in T_{\Sigma_1}(X)$ is a S_0-linear term, (ii) $t = t^\circ\theta^\circ$; and (iii) $dom(\theta^\circ) = \{x_1, \ldots, x_n\}$ are pairwise distinct variables disjoint from $vars(t)$, and $\{x_1, \ldots, x_n\} = vars(t^\circ) \cap X_0$, where $\Sigma_1 = (S, \leq, F_1)$ and $X_0 = \{X_s\}_{s \in S_0}$.*

Lemma 1 shows that such an abstraction can always be chosen so as to provide a canonical decomposition of t enjoying useful properties.

Lemma 1. *Let Σ be a signature with built-in subsignature $\Sigma_0 = (S_0, F_0)$. For each $t \in T_\Sigma(X)$, there exist an abstraction of built-ins $(\lambda x_1 \cdots x_n.t^\circ; \theta^\circ)$. Furthermore, $\{x_1, \ldots, x_n\}$ can always be chosen to be disjoint from an arbitrarily chosen finite subset Y of X_0.*

Proof. By induction on the structure of t.

From now on, for any $t \in T_\Sigma(X)$ and $Y \subseteq X_0$ finite, the expression $abstract_{\Sigma_1}(t, Y)$ will denote the choice of an abstraction pair $(\lambda x_1 \cdots x_n.t^\circ\,;\theta^\circ)$ satisfying the disjointness condition w.r.t. Y stated in Lemma 1. Note that each substitution θ with $dom(\theta) = \{x_1, \ldots, x_n\}$ has an associated quantifier-free (QF) formula $[\theta] = \bigwedge_{i=1}^{n} (x_i = \theta(x_i))$. In particular, the formula $[\theta^\circ]$ associated to the substitution θ° in $(\lambda x_1 \cdots x_n.t^\circ\,;\theta^\circ)$ binds the abstraction variables $x_1 \cdots x_n$ to subterms of t.

Example 2. Let t be the term

$$\langle\, time : R, temp : s(s(T)), setpoint : S, ac : B \,\rangle$$

in the signature of Example 1, where R, S, T are variables of sort *Nat* and B is a variable of sort *Bool*. Consider the term t°

$$\langle\, time : N_0, temp : N_1, setpoint : N_2, ac : B_0 \,\rangle$$

and the substitution θ° defined by $\theta^\circ(N_0) = R$, $\theta^\circ(N_1) = s(s(T))$, $\theta^\circ(N_2) = S$, $\theta^\circ(B_0) = B$, and $\theta^\circ(x) = x$ otherwise, and with N_0, N_1, N_2 variables of sort *Nat* and B_0 a variable of sort *Bool*. The context $\lambda N_0, N_1, N_2, B_0.t^\circ$ is an abstraction of built-ins for t and θ° satisfies properties (i)–(iii) in Lemma 1. Moreover, for any set Y not containing variables N_0, N_1, N_2, B_0, t° and θ° satisfy $abstract_{\Sigma_1}(t, Y) = (\lambda N_0, N_1, N_2, B_0.t^\circ\,;\theta^\circ)$ with $[\theta^\circ]$ the QF-formula

$$N_0 = R \;\wedge\; N_1 = s(s(T)) \;\wedge\; N_2 = S \;\wedge\; B_0 = B.$$

As illustrated by Example 2, the abstraction of built-ins for a given term t introduces new variables even for maximal built-in subterms that correspond to a built-in variable in t. However, for efficiency reasons, an algorithm actually implementing the abstraction procedure can avoid introducing extra-variables in these cases since these extra-variables are not technically useful.

Under certain restrictions on axioms, matching a Σ-term t to a Σ-term u can be decomposed modularly into Σ_1-matching of the corresponding λ-abstraction and Σ_0-matching of the built-in subterms. This is described in Lemma 3, with the help of Lemma 2.

Lemma 2. *Let $\Sigma = (S, \leq, F)$ be a signature with built-in subsignature $\Sigma_0 = (S_0, F_0)$. Let B_0 be a set of Σ_0-axioms and B_1 a set of Σ_1-axioms. For B_0 and B_1 regular, linear, and collapse free for any sort in S_0, and sort-preserving, and $t \in T_\Sigma(X_0)$:*

(a) if $t \in T_{\Sigma_0}(X_0)$ and $t =_{B_1} t'$, then $t = t'$;
(b) if $t \in T_{\Sigma_1}(X_0)$ and $t =_{B_0} t'$, then $t = t'$;
(c) if $t \in T_{\Sigma_1}(X_0)$ and $t =^1_{B_1} t'$, then $vars(t) = vars(t')$ and t is linear if and only if t' is so;

Proof.

(a) Axioms B_1 do not mention any function symbol in F_0 and are sort-preserving. Therefore, the equation in B_1 can only apply to variables in X_0. But B_1 is collapse-free for any sort in S_0. Therefore, no B_1 equation can be applied to t, forcing $t = t'$.
(b) Same argument as (a).
(c) Consequence of B_1 being regular and linear.

Lemma 3. *Let $\Sigma = (S, \leq, F)$ be a signature with built-in subsignature $\Sigma_0 = (S_0, F_0)$. Let B_0 be a set of Σ_0-axioms and B_1 a set of Σ_1-axioms. For B_0 and B_1 regular, linear, collapse free for any sort in S_0, and sort-preserving, if $t \in T_{\Sigma_1}(X_0)$ is linear with $vars(t) = \{x_1, \ldots, x_n\}$, then for each $\theta : X_0 \longrightarrow T_{\Sigma_0}(X_0)$:*

(a) if $t\theta =^1_{B_0} t'$, then there exist $x \in \{x_1, \ldots, x_n\}$ and $w \in T_{\Sigma_0}(X_0)$ such that $\theta(x) =^1_{B_0} w$ and $t' = t\theta'$, with $\theta'(x) = w$ and $\theta'(y) = \theta(y)$ if $y \neq x$;
(b) if $t\theta =^1_{B_1} t'$, then there exists $v \in T_{\Sigma_1}(X_0)$ such that $t =^1_{B_1} v$ and $t' = v\theta$; and
(c) if $t\theta =_{B_0 \uplus B_1} t'$, then there exist $v \in T_{\Sigma_1}(X_0)$ and $\theta' : X_0 \longrightarrow T_{\Sigma_0}(X_0)$ such that $t' = v\theta'$, $t =_{B_1} v$, and $\theta =_{B_0} \theta'$ (i.e., $\theta(x) =_{B_0} \theta'(x)$ for each $x \in X_0$).

Proof. (a) It follows from Lemma 2 (b) that B_0 can only be applied on some built-in subterm $\theta(x)$ of $t\theta$, for some $x \in dom(\theta)$. That is, there is $w \in T_{\Sigma_0}(X_0)$ such that $\theta(x) =^1_{B_0} w$ and, since t is linear, $t' = t\theta'$, where $\theta'(x) = w$ and $\theta'(y) = \theta(y)$ if $y \neq x$.
(b) It follows from Lemma 2 (c) that equational deduction with B_1 can only permute the built-in variables in t and it does not equate built-in subterms such as the ones in $ran(\theta)$. Hence, by Lemma 2 (c), there exists a linear $v \in T_{\Sigma_1}(X_0)$ such that $t =^1_{B_1} v$ and $t' = v\theta$.
(c) Follows by induction on the proof's length in $B_0 \uplus B_1$.

4 Rewriting Modulo a Built-in Subtheory

This section presents both the notion of a rewrite theory modulo built-ins and the ground rewrite relation induced by it, along with some examples. This section concludes by presenting a technical result that is at the heart of the main contribution of this paper: for a very general class of these rewrite theories, matching provides a complete ground unifiability procedure.

A rewrite theory modulo a built-in subtheory is a topmost rewrite theory with a signature of built-ins and where structural axioms can be given for both built-in and non built-in terms, but equations are only allowed at the built-in level. In these rewrite theories, rules are given at a top sort, built-in extra variables are allowed in their right-hand side, and constraints are quantifier-free formulas over built-in terms.

Definition 3 (Rewriting Modulo a Built-in Subtheory). *A* rewrite theory modulo the built-in subtheory $\mathcal{E}_0$ *is a topmost rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ *with:*

(a) Σ=$(S, \leq, F)$ *a signature with built-in subsignature* Σ_0=(S_0, F_0) *and a top sort State in* $S \setminus S_0$*;*

(b) $E = E_0 \uplus B_0 \uplus B_1$*, where* E_0 *is a finite set of* Σ_0*-equations,* B_0 *(resp.,* B_1*) are* Σ_0*-axioms (resp.,* Σ_1*-axioms) satisfying the conditions in Lemma 3 (i.e.,* B_0 *and* B_1 *regular, linear, collapse-free for any sort in* S_0*, and sort-preserving),* $\mathcal{E}_0 = (\Sigma_0, E_0 \uplus B_0)$ *and* $\mathcal{E} = (\Sigma, E)$ *are admissible, and the theory inclusion* $\mathcal{E}_0 \subseteq \mathcal{E}$ *is protecting;*

(c) R *a finite set of* rewrite rules *of the form* $l(\overrightarrow{x_1}, \overrightarrow{y}) \rightarrow r(\overrightarrow{x_2}, \overrightarrow{y})$ ***if*** $\phi(\overrightarrow{x_3})$ *such that* $l, r \in T_\Sigma(X)_{State}$*,* l *is* $(S \setminus S_0)$*-linear,* $\overrightarrow{x_i}$ *is of the form* $\overrightarrow{x_i}{:}\overrightarrow{s_i}$ *with* $\overrightarrow{s_i} \in S_0^*$*, for* $i \in \{1, 2, 3\}$*,* $\overrightarrow{y}{:}\overrightarrow{s}$ *with* $\overrightarrow{s} \in (S \setminus S_0)^*$*, and* $\phi \in QF_{\Sigma_0}(X_0)$*, where* $QF_{\Sigma_0}(X_0)$ *denotes the set of quantifier-free* Σ_0*-formulas with variables in* X_0*.*

In Definition 3, a quantifier-free Σ_0-formula in $QF_{\Sigma_0}(X_0)$ is a Boolean combination of atoms, where an atom is a Σ_0-equation with variables in X_0. Note that no assumption is made on the relationship between the built-in variables $\overrightarrow{x_1}$ in the left-hand side, $\overrightarrow{x_2}$ in the right-hand side, and $\overrightarrow{x_3}$ in the condition ϕ of a rewrite rule. This freedom is key for specifying open systems with a rewrite theory because, for instance, $\overrightarrow{x_2}$ can have new variables not appearing in $\overrightarrow{x_1}$.

The restriction to topmost rewrite theories in Definition 3 is a fairly mild one: in practice, most distributed systems (including distributed object-based ones), actor systems, cyber-physical systems, and network protocols, can be easily modeled by topmost rewrite theories using the simple theory transformation described in [48].

Example 3. Recall the thermostat example from the Introduction, and the signature with built-ins from Examples 1 and 2. Consider the following extension of this signature with natural number addition and Boolean conjunction, and some axioms and equations for these symbols, where variables M, N have sort *Nat* and variables X, Y have sort *Bool*:

$$\_ + \_ : Nat \times Nat \longrightarrow Nat \qquad \_ \wedge \_ : Bool \times Bool \longrightarrow Bool$$
$$M + N = N + M \qquad X \wedge Y = Y \wedge X$$
$$M + 0 = M \qquad M + s(N) = s(M + N).$$

Also, consider the topmost rewrite rules of the thermostat system from Section 1, where R, S, T, T_e are variables of sort *Nat* and B is a variable of sort *Bool*:

$$\langle\, time{:}R, temp{:}T, setpoint{:}S, ac{:}B \,\rangle \rightarrow [\, time{:}R, temp{:}T, setpoint{:}S, ac{:}true \,] \textbf{ if } T > S$$
$$\langle\, time{:}R, temp{:}T, setpoint{:}S, ac{:}B \,\rangle \rightarrow [\, time{:}R, temp{:}T, setpoint{:}S, ac{:}false \,] \textbf{ if } T \leq S$$
$$[\, time{:}R, temp{:}T, setpoint{:}S, ac{:}B \,] \rightarrow \langle\, time{:}R + 1, temp{:}T_e, setpoint{:}S, ac{:}B \,\rangle$$

This is an example of a rewrite theory modulo a built-in subtheory with built-in subsignature:

$$\Sigma_0 = (\{Nat, Bool\}, \{0, s, true, false, <, \leq, +, \wedge\})\,.$$

In this rewrite theory, the sort *State* fulfills Condition (a) in Definition 3, and the sets B_0 and E_0 have two elements each, namely,

$$B_0 = \{M + N = N + M, X \wedge Y = Y \wedge X\}$$
$$E_0 = \{M + 0 = M, M + s(N) = s(M + N)\}.$$

Note that the axioms in B_0 are regular, linear, collapse-free for *Nat* and *Bool*, respectively, and sort-preserving. It is easy to check that the built-in subtheory $\mathcal{E}_0 = (\Sigma_0, E_0 \uplus B_0)$ is admissible. Furthermore, in each rewrite rule, the left-hand side is linear, the extra variables in the right-hand side are built-in variables, and the condition is a QF-free Σ_0-formula.

The next task is to define what *ground* computation means for a rewrite theory modulo a built-in subtheory.

Definition 4 (Ground Rewrite Relation). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. The* ground rewrite relation $\rightarrow_{\mathcal{R}}$ *induced by $\mathcal{R}$ on $T_{\Sigma,State}$ is defined for $t, u \in T_{\Sigma,State}$ by $t \rightarrow_{\mathcal{R}} u$ if and only if there is a rule $l \rightarrow r$ **if** ϕ in R and a ground substitution $\sigma : X \longrightarrow T_\Sigma$ such that (a) $t =_E l\sigma$, $u =_E r\sigma$, and (b) $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$.*

The ground rewrite relation $\rightarrow_{\mathcal{R}}$ is the topmost rewrite relation induced by R modulo E on $T_{\Sigma,State}$. This relation is defined even when a rule in R has extra variables in its right-hand side: the rule is then nondeterministic and such extra variables can be arbitrarily instantiated, provided that the corresponding instantiation of ϕ holds. Also, note that non-built-in variables can occur in l, but $\phi\sigma$ is a *ground* (i.e., variable-free) formula in $QF_{\Sigma_0}(\emptyset)$, so that either $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$ or $\mathcal{T}_{\mathcal{E}_0} \not\models \phi\sigma$.

Example 4. Recall the rewrite theory modulo a built-in subtheory $\mathcal{R}$ from Example 3. The following is an example of a rewrite computation with $\rightarrow_{\mathcal{R}}$ for the thermostat and air conditioning system:

$$\begin{aligned}
&\langle\, time : 0, temp : 69, setpoint : 73, ac : false \,\rangle \\
\rightarrow_{\mathcal{R}}\ &[\, time : 0, temp : 69, setpoint : 73, ac : false \,] \\
\rightarrow_{\mathcal{R}}\ &\langle\, time : 1, temp : 71, setpoint : 73, ac : false \,\rangle \\
\rightarrow_{\mathcal{R}}\ &[\, time : 1, temp : 71, setpoint : 73, ac : false \,] \\
\rightarrow_{\mathcal{R}}\ &\langle\, time : 2, temp : 74, setpoint : 73, ac : false \,\rangle \\
\rightarrow_{\mathcal{R}}\ &[\, time : 2, temp : 74, setpoint : 73, ac : true \,] \\
\rightarrow_{\mathcal{R}}\ &\langle\, time : 3, temp : 71, setpoint : 73, ac : true \,\rangle
\end{aligned}$$

The initial state corresponds to the global clock having 0 time units, the thermostat registering a temperature of 69 degrees from the environment and having setpoint at 73 degrees, while the air conditioning device is turned off. In this state, only a transition under control of the system is enabled, namely, the transition that turns off the air conditioning device, which is already off. The thermostat then senses 71 degrees from the environment and the global clock is advanced in one time unit. The new temperature does not exceed the setpoint of the thermostat and thus the air conditioning device remains off after the next internal transition. At time 2, the temperature is updated to 74

degrees and, consequently, the next zero-time computation turns the air conditioning device on. Finally, at time 3, 71 degrees are sensed from the environment.

According to Definition 4, the first rewrite step in this computation is induced by the rewrite rule

$$[\, time : R, temp : T, setpoint : S, ac : B \,] \rightarrow \langle\, time : R + 1, temp : T_e, setpoint : S, ac : B \,\rangle$$

and ground substitution σ satisfying

$$\sigma(R) = 0, \qquad \sigma(T) = 69, \qquad \sigma(S) = 73 \qquad \sigma(B) = \mathit{false}, \qquad \sigma(T_e) = 71.$$

In this case, note that the constraint ϕ corresponds to the empty conjunction and then $\phi\sigma = \mathit{true}$, which is always satisfiable. On the other hand, note that in each ground rewrite step with $\rightarrow_{\mathcal{R}}$ in the above trace, the extra built-in variable T_e in the right-hand side of the applied rule is non-deterministically chosen, which is technically captured by the ground substitution σ.

For technical reasons, it is very useful to shift the focus to a class of rewrite theories modulo built-ins in which the rewrite rules are left-linear. For any rewrite theory modulo built-ins such a simpler rewrite theory can always be obtained by means of the semantics-preserving theory transformation $\mathcal{R} \mapsto \mathcal{R}^\circ$ presented in Definition 5. As shown by Lemma 4, this transformation preserves ground rewriting. The specific reason for this transformation is towards achieving the ultimate goal of having an SMT solver exclusively handling *all* constraints over built-in terms, including those used for expressing equality. If a left-hand side of a rule were allowed to be non-linear for built-in sorts, then equality over built-in terms could be wrongfully delegated to the matching algorithm used for rewriting.

Definition 5 (Normal Form of a Rewrite Theory Modulo $\mathcal{E}_0$). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. Its* normal form $\mathcal{R}^\circ = (\Sigma, E, R^\circ)$ *has rules:*

$$R^\circ = \left\{ l^\circ \rightarrow r \textbf{ if } \phi \wedge [\theta^\circ] \mid (l \rightarrow r \textbf{ if } \phi \in R) \wedge \left(\lambda \overrightarrow{x}.l^\circ ; \theta^\circ\right) = \mathit{abstract}_\Sigma(l, \mathit{vars}(\{l, r, \phi\}))\right\}.$$

Note that the rewrite theory in Example 3 is already in normal form, since its set of rules is left-linear. Lemma 4 formalizes the previous claim about the fact that the rewrite relation induced by a rewrite theory modulo built-ins is preserved under the transformation $\mathcal{R} \mapsto \mathcal{R}^\circ$ in Definition 5, specifically meaning that both theories satisfy the same reachability properties.

Lemma 4 (Invariance of Ground Rewriting under Normalization). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. Then $\rightarrow_{\mathcal{R}} = \rightarrow_{\mathcal{R}^\circ}$.*

Proof. It is shown that $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}^\circ}$ and $\rightarrow_{\mathcal{R}^\circ} \subseteq \rightarrow_{\mathcal{R}}$.

($\subseteq$) Let $t, u \in T_{\Sigma,State}$. If $t \rightarrow_{\mathcal{R}} u$, then there is a rule $(l \rightarrow r \textbf{ if } \phi) \in R$ and a ground substitution $\sigma : X \longrightarrow T_\Sigma$ such that $t =_E l\sigma$, $u =_E r\sigma$, and $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$. It suffices to prove $t \rightarrow_{\mathcal{R}^\circ} u$ with witnesses $(l^\circ \rightarrow r \textbf{ if } \phi \wedge [\theta^\circ]) \in R^\circ$ and $\rho = \theta^\circ\sigma$. Note that $t =_E l\sigma = l^\circ\theta^\circ\sigma = l^\circ\rho$. For $\mathcal{T}_{\mathcal{E}_0} \models (\phi \wedge [\theta^\circ])\rho$ first note that $\mathcal{T}_{\mathcal{E}_0} \models \phi\rho$ since

$\phi\rho = \phi\theta^\circ\sigma = \phi\sigma$ (because $vars(\phi) \cap dom(\theta^\circ) = \emptyset$) and $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$ by assumption. For $\mathcal{T}_{\mathcal{E}_0} \models \phi^\circ\rho$ notice that $\theta^\circ\theta^\circ = \theta^\circ$ because $ran(\theta^\circ) \cap dom(\theta^\circ) = \emptyset$, and then:

$$[\theta^\circ]\rho = \left(\bigwedge_{i=1}^{n} x_i = \theta^\circ(x_i)\right)\rho = \bigwedge_{i=1}^{n} x_i\rho = \theta^\circ(x_i)\rho = \bigwedge_{i=1}^{n} \theta^\circ(x_i)\sigma = \theta^\circ(x_i)\theta^\circ\sigma$$
$$= \bigwedge_{i=1}^{n} \theta^\circ(x_i)\sigma = \theta^\circ(x_i)\sigma = \top.$$

Hence, $t \to_{\mathcal{R}^\circ} u$.

($\supseteq$) Let $t, u \in T_{\Sigma,State}$. If $t \to^\circ_{\mathcal{R}} u$, then there is a rule $(l \to r \textbf{ if } \phi) \in R$, with $(\lambda x_1 \cdots x_n.l^\circ\,;\theta^\circ)$ be the abstraction of built-ins for l, and a ground substitution $\sigma : X \longrightarrow T_\Sigma$ such that $t =_E l^\circ\sigma$, $u =_E r\sigma$, and $\mathcal{T}_{\mathcal{E}_0} \models (\phi \wedge [\theta^\circ])\sigma$. It suffices to prove that $t \to_{\mathcal{R}} u$ with rule $(l \to r \textbf{ if } \phi) \in R$. Substitution σ can be decomposed into substitutions $\theta : X_0 \longrightarrow T_{\Sigma_0}(X_0)$ and $\rho : X \longrightarrow T_\Sigma$, with $\theta(x) = \sigma(x)$ if $x \in \{x_1, \ldots, x_n\}$ and $\theta(x) = x$ otherwise, such that $\sigma = \theta\rho$. From $\mathcal{T}_{\mathcal{E}_0} \models (\phi \wedge [\theta^\circ])\sigma$ it follows that $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$, i.e., $\mathcal{T}_{\mathcal{E}_0} \models \phi\rho$ because $vars(\phi) \cap dom(\theta) = \emptyset$. Also, it follows that $\mathcal{T}_{\mathcal{E}_0} \models \bigwedge_{i=1}^{n} \theta(x_i)\rho = \theta^\circ(x_i)\rho$, which implies that:

$$t =_E l^\circ\sigma = l^\circ\theta\rho =_{E_0 \uplus B_0} l^\circ\theta^\circ\rho = l\rho.$$

Hence, $t \to_{\mathcal{R}} u$.

Finally, Lemma 5 states that for the class of normalized rewrite theories modulo built-ins, matching provides a complete ground unifiability procedure. More specifically, by the properties of the axioms in a rewrite theory modulo built-ins $\mathcal{R} = (\Sigma, E_0 \uplus B_0 \uplus B_1)$, B_1-matching a term $t \in T_\Sigma(X_0)$ to a left-hand side l° of a rule in R° provides a complete unifiability algorithm for ground B_1-unification of t and l°.

Lemma 5 (Matching Lemma). *Let $\mathcal{R} = (\Sigma, E_0 \uplus B_0 \uplus B_1, R)$ be a rewrite theory modulo $\mathcal{E}_0$. For $t \in T_\Sigma(X_0)_{State}$ and l° a left-hand side of a rule in R° with $vars(t) \cap vars(l^\circ) = \emptyset$,*

$$t \ll_{B_1} l^\circ \quad \textit{if and only if} \quad GU_{B_1}(t = l^\circ) \neq \emptyset$$

where $GU_{B_1}(t = l^\circ) = \{\sigma : X \longrightarrow T_\Sigma \mid t\sigma =_{B_1} l^\circ\sigma\}$.

Proof.

($\Longrightarrow$) If $t \ll_{B_1} l^\circ$, then $t =_{B_1} l^\circ\theta$ for some $\theta : X \longrightarrow T_\Sigma(X)$. Let $\rho : X \longrightarrow T_\Sigma$ be any ground substitution, which exists because Σ has nonempty sorts. Then $\theta\rho \in GU_{B_1}(t = l^\circ)$.

($\Longleftarrow$) Let $\sigma \in GU_{B_1}(t = l^\circ)$ with $l \to r \textbf{ if } \phi \in R$. Let $vars(l^\circ) \cap X_0 = \{x_1, \ldots, x_n\}$ and $X_1 = X \setminus X_0$. Note that there are substitutions

$$\alpha : vars(l^\circ) \cap X_1 \longrightarrow T_{\Sigma_1}(X_0)$$
$$\rho : X \setminus dom(\alpha) \longrightarrow T_\Sigma$$

satisfying $\sigma = \alpha\rho$ and such that $(l^\circ\alpha) \in T_{\Sigma_1}(X_0)$ is linear and

$$ran(l^\circ\alpha) \cap (vars(t, l^\circ)) = \emptyset.$$

Let $ran(\alpha) = \{y_1, \ldots, y_m\}$. Therefore, by Lemma 3, there exists $u \in T_{\Sigma_1}(X_0)$ such that $u =_{B_1} l^\circ\alpha$, u is linear, and $vars(u) = vars(l^\circ\alpha) = x_1, \ldots, x_n, y_1, \ldots, y_m$, and $u\rho = t$. Moreover, t can be written as $u(t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m})$ with $t_i \in T_{\Sigma_0}(X_0)$. Define $\theta : X_0 \longrightarrow T_{\Sigma_0}(X_0)$ by $\theta(x) = t_i$ if $x \in \{x_1, \ldots, x_n\}$, $\theta(x) = t_{i+n}$ if $x \in \{y_1, \ldots, y_m\}$, and $\theta(x) = x$ otherwise. Then:

$$\begin{aligned} t &= u(t_1, \ldots, t_n, t_{n+1}, \ldots, t_{m+n}) \\ &= u(x_1, \ldots, x_n, y_1, \ldots, y_m)\theta \\ &=_{B_1} l^\circ\alpha\theta. \end{aligned}$$

Therefore, $t \ll_{B_1} l^\circ$.

5 Symbolic Rewriting Modulo a Built-in Subtheory

This section explains how a rewrite theory modulo built-ins, as proposed in Section 4, induces a symbolic rewrite relation and presents a general mechanism for symbolic reachability analysis, along with some examples. One of the main results of this section is that the symbolic rewrite relation is *sound and complete* w.r.t. to the ground rewriting semantics for rewrite theories modulo built-ins from Section 4. The key idea is that, when constrains over the built-ins are decidable, the transitions of the symbolic relation can be performed by rewriting modulo axioms and satisfiability of the constraints can be handled by an SMT decision procedure. This approach provides an executable symbolic method via rewriting, called *rewriting modulo SMT*, that is a sound and complete symbolic reachability mechanism for rewrite theories.

The symbolic rewrite relation induced by a rewrite theory with built-ins $\mathcal{R}$ operates over pairs $(t\,;\varphi)$, called *constrained terms*, where t is a term and φ a constraint of built-ins. Intuitively, in a constrained term $(t\,;\varphi)$, the term t can contain built-in variables and thus can serve the purpose of a template for all its ground instances that are constrained by φ. Definition 6 spells out the precise semantics of a constrained term.

Definition 6 (Constrained Terms). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. A* constrained term *is a pair $(t\,;\varphi)$ in $T_\Sigma(X)_{State} \times QF_{\Sigma_0}(X_0)$. Its* denotation *$[\![t]\!]_\varphi$ is defined as $[\![t]\!]_\varphi = \{t' \in T_{\Sigma,State} \mid (\exists\sigma : X \longrightarrow T_\Sigma)\ t' =_E t\sigma \wedge \mathcal{T}_{\mathcal{E}_0} \models \varphi\sigma\}$.*

The domain of σ in Definition 6 ranges over all variables X and consequently $[\![t]\!]_\varphi \subseteq T_{\Sigma,State}$ for any $t \in T_\Sigma(X)_{State}$, even if $vars(t) \not\supseteq vars(\varphi)$. Note, then, that $[\![t]\!]_\varphi$ semantically represents the set of all ground states that are E-equal to instances of t and satisfy φ.

The following auxiliary notation for variable renaming is used for formally introducing the symbolic rewrite relation on constrained terms: in the rest of the paper, the expression *fresh-vars*(Y), for $Y \subseteq X$ with Y finite, represents the choice of a variable renaming $\zeta : X \longrightarrow X$ satisfying $Y \cap ran(\zeta) = \emptyset$.

Definition 7 (Symbolic Rewrite Relation). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$. The* symbolic rewrite relation *$\rightsquigarrow_\mathcal{R}$ induced by $\mathcal{R}$ on $T_\Sigma(X)_{State} \times QF_{\Sigma_0}(X_0)$ is defined for $t, u \in T_\Sigma(X)_{State}$ and $\varphi, \varphi' \in QF_{\Sigma_0}(X_0)$ by $(t\,;\varphi) \rightsquigarrow_\mathcal{R} (u\,;\varphi')$ if*

and only if there is a rule $l \to r$ **if** ϕ *in* R *and a substitution* $\theta : X \longrightarrow T_\Sigma(X)$ *such that: (a)* $t =_E l\zeta\theta$ *and* $u = r\zeta\theta$*, (b)* $\mathcal{T}_{\mathcal{E}_0} \models (\varphi' \Leftrightarrow \varphi \wedge \phi\zeta\theta)$*, and (c)* φ' *is* $\mathcal{T}_{\mathcal{E}_0}$*-satisfiable, where* $\zeta = \mathit{fresh\text{-}vars}(\mathit{vars}(t, \varphi))$*.*

The symbolic relation $\rightsquigarrow_{\mathcal{R}}$ on constrained terms is defined as a topmost rewrite relation induced by R modulo E on $T_\Sigma(X)$ with extra bookkeeping of constraints. Note that φ' in $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}} (u\,;\varphi')$, when witnessed by $l \to r$ **if** ϕ and θ, is *semantically equivalent* to $\varphi \wedge \phi\zeta\theta$, in contrast to being *syntactically* equal. This extra freedom allows for simplification of constraints if desired. Also, such a constraint φ' is satisfiable in $\mathcal{T}_{\mathcal{E}_0}$, implying that φ and $\phi\theta$ are both satisfiable in $\mathcal{T}_{\mathcal{E}_0}$, and therefore $[\![t]\!]_\varphi \neq \emptyset \neq [\![u]\!]_{\varphi'}$. Note that, up to the choice of the semantically equivalent φ' for which a fixed strategy is assumed, the symbolic relation $\rightsquigarrow_{\mathcal{R}}$ is deterministic, in the sense of being determined by the rule and the substitution $\zeta\theta$, because the renaming of variables in the rules is fixed by *fresh-vars*. This is key when executing $\rightsquigarrow_{\mathcal{R}}$, as explained in Section 6.

Example 5. Recall the rewrite theory modulo built-ins $\mathcal{R}$ for the thermostat and air conditioning device from Example 4. In what follows, variables T, S, N_0, N_1, N_2 range over the sort *Nat*. Below, there is an example of a *symbolic* rewrite computation with $\rightsquigarrow_{\mathcal{R}}$:

$$
\begin{aligned}
& (\langle\, \mathit{time} : 0, \mathit{temp} : T, \mathit{setpoint} : S, \mathit{ac} : \mathit{false} \,\rangle\,; \mathit{true}) \\
\rightsquigarrow_{\mathcal{R}}\ & ([\, \mathit{time} : 0, \mathit{temp} : N_0, \mathit{setpoint} : S, \mathit{ac} : \mathit{false} \,]\,; \mathit{true}) \\
\rightsquigarrow_{\mathcal{R}}\ & (\langle\, \mathit{time} : 1, \mathit{temp} : N_0, \mathit{setpoint} : S, \mathit{ac} : \mathit{false} \,\rangle\,; \mathit{true}) \\
\rightsquigarrow_{\mathcal{R}}\ & ([\, \mathit{time} : 1, \mathit{temp} : N_0, \mathit{setpoint} : S, \mathit{ac} : \mathit{false} \,]\,; N_0 \leq S) \\
\rightsquigarrow_{\mathcal{R}}\ & (\langle\, \mathit{time} : 2, \mathit{temp} : N_1, \mathit{setpoint} : S, \mathit{ac} : \mathit{false} \,\rangle\,; N_0 \leq S) \\
\rightsquigarrow_{\mathcal{R}}\ & ([\, \mathit{time} : 2, \mathit{temp} : N_1, \mathit{setpoint} : S, \mathit{ac} : \mathit{true} \,]\,; S < N_1 \wedge N_0 \leq S) \\
\rightsquigarrow_{\mathcal{R}}\ & (\langle\, \mathit{time} : 3, \mathit{temp} : N_2, \mathit{setpoint} : S, \mathit{ac} : \mathit{true} \,\rangle\,; S < N_1 \wedge N_0 \leq S)\,.
\end{aligned}
$$

At time 0, the initial state represents all those system instances where the setpoint and the temperature reading from the environment are unspecified, and the air conditioning system is turned off; in this case, the state constraint is the empty one represented by *true*. At time 1, the system transitions to a symbolic state in which the temperature reading from the environment is captured by the fresh built-in variable N_0; the next internal transition is possible because the constraint $N_0 \leq S$ is satisfiable. At time 2, the system reaches a state in which the external temperature is represented by the fresh built-in variable N_1. The next internal computation turns the air conditioning device on because the constraint $S < N_1 \wedge N_0 \leq S$ is satisfiable. In the last transition, at time 3, the system reaches a state in which the external temperature is represented by the fresh built-in variable N_2. As a remark, note that the ground computation with $\to_{\mathcal{R}}$ given as part of Example 4 is a semantic instance of the above-given *symbolic* computation with $\rightsquigarrow_{\mathcal{R}}$. More precisely, the ground trace in Example 4 is an instance of the symbolic trace above, witnessed by a ground substitution σ satisfying $\sigma(S) = 73$, $\sigma(T) = 69$, $\sigma(N_0) = \sigma(N_2) = 71$, and $\sigma(N_1) = 74$. As a final remark, note that the constraints are accumulated in the symbolic computation, despite the fact that some of their conjuncts are 'meaningless' w.r.t. the corresponding constrained term. For example, $N_0 \leq S$ does

not play any role when constraining the term $[\, time : 2, temp : N_1, setpoint : S, ac : true \,]$ because N_0 does not occur in this state. In practice, such an important optimization can be considered as part of an efficient implementation of the symbolic rewrite relation.

The next important question to ask is whether this symbolic rewrite relation soundly and completely simulates its ground rewriting counterpart. The rest of this section affirmatively answers this question in the case of *normalized* rewrite theories modulo built-ins. Thanks to Lemma 4, the conclusion is therefore that $\rightsquigarrow_{\mathcal{R}^\circ}$ soundly and completely simulates $\rightarrow_{\mathcal{R}}$ for *any* rewrite theory $\mathcal{R}$ modulo built-ins $\mathcal{E}_0$.

The soundness of $\rightsquigarrow_{\mathcal{R}^\circ}$ w.r.t. $\rightarrow_{\mathcal{R}^\circ}$ is stated in Theorem 1.

Theorem 1 (Soundness). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$, $t, u \in T_\Sigma(X)_{State}$, and $\varphi, \varphi' \in QF_{\Sigma_0}(X_0)$. If $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}^\circ} (u\,;\varphi')$, then $t\rho \rightarrow_{\mathcal{R}^\circ} u\rho$ for all $\rho : X \longrightarrow T_\Sigma$ satisfying $\mathcal{T}_{\mathcal{E}_0} \models \varphi'\rho$.*

Proof. Let $\rho : X \longrightarrow T_\Sigma$ satisfy $\mathcal{T}_{\mathcal{E}_0} \models \varphi'\rho$. The goal is to show that $t\rho \rightarrow_{\mathcal{R}^\circ} u\rho$. Let $l^\circ \rightarrow r$ **if** $\phi \in R^\circ$ and $\theta : X \longrightarrow T_\Sigma(X)$ witness $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}^\circ} (u\,;\varphi')$. Then $t =_E l^\circ\zeta\theta$, $u =_E r\zeta\theta$, $\mathcal{E}_0 \vdash (\varphi' \Leftrightarrow \varphi \wedge \phi\zeta\theta)$, and φ' is $\mathcal{T}_{\mathcal{E}_0}$-satisfiable. Without loss of generality assume $dom(\theta) = vars(l^\circ\zeta)$ and $\theta|_{vars(t,\varphi)} = id$, and let $\sigma = \zeta\theta\rho$. Then note that $t\rho =_E (l^\circ\zeta\theta)\rho = l^\circ\zeta\theta\rho = l^\circ\sigma$ and $u\rho =_E (r\zeta\theta)\rho = r\zeta\theta\rho = r\sigma$. Moreover, $\mathcal{T}_{\mathcal{E}_0} \models (\varphi' \Leftrightarrow \varphi \wedge \phi\zeta\theta)$ and $\mathcal{T}_{\mathcal{E}_0} \models \varphi'\rho$ imply $\mathcal{T}_{\mathcal{E}_0} \models \phi\zeta\theta\rho$, i.e., $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$. Therefore, $t\rho \rightarrow_{\mathcal{R}^\circ} u\rho$, as desired.

The completeness of $\rightsquigarrow_{\mathcal{R}^\circ}$ w.r.t. $\rightarrow_{\mathcal{R}^\circ}$ is stated in Theorem 2. Intuitively, completeness states that a symbolic relation yields an over-approximation of its ground rewriting counterpart.

Theorem 2 (Completeness). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$, $t \in T_\Sigma(X)_{State}$, $u' \in T_{\Sigma,State}$, and $\varphi \in QF_{\Sigma_0}(X_0)$. For any $\rho : X \longrightarrow T_\Sigma$ such that $t\rho \in [\![t]\!]_\varphi$ and $t\rho \rightarrow_{\mathcal{R}^\circ} u'$, there exist $u \in T_\Sigma(X)_{State}$ and $\varphi' \in QF_{\Sigma_0}(X_0)$ such that $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}^\circ} (u\,;\varphi')$ and $u' \in [\![u]\!]_{\varphi'}$.*

Proof. By the assumptions there is a rule $(l^\circ \rightarrow r$ **if** $\phi) \in R^\circ$ and a ground substitution $\sigma : X \longrightarrow T_\Sigma$ satisfying $t\rho =_E l^\circ\sigma$, $u' =_E r\sigma$, and $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$. Without loss of generality assume $vars(t, \varphi) \cap vars(l^\circ, r, \phi)) = \emptyset$, because l, r, ϕ can be renamed by means of *fresh-vars*. Furthermore, since $vars(t, \varphi) \cap vars(l^\circ, \phi)) = \emptyset$, $\sigma = \rho$ can be assumed. The goal is to show the existence of $u \in T_\Sigma(X)_{State}$ and $\varphi' \in QF_{\Sigma_0}(X_0)$ such that (i) $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}^\circ} (u\,;\varphi')$ and (ii) $u' \in [\![u]\!]_{\varphi'}$. Since l° is linear and built-in subterms are variables, by Lemma 3 there exists $\alpha : X \longrightarrow T_\Sigma$ satisfying $t\alpha =_{B_1} l^\circ\alpha$. Hence $GU_{B_1}(t = l^\circ) \neq \emptyset$ and, by Lemma 5, there exists $\theta' : X \longrightarrow T_\Sigma(X)$ satisfying $t =_{B_1} l^\circ\theta'$ and a fortiori $t =_{E_0 \uplus B_0 \uplus B_1} l^\circ\theta'$. Let $\theta : X \longrightarrow T_\Sigma(X)$ be defined by $\theta(x) = \theta'(x)$ if $x \in vars(l)$ and $\theta(x) = \rho(x)$ otherwise. Note that $\theta|_{vars(l)}\rho =_{E_0 \uplus B_0} \rho|_{vars(l)}$. Define $u = r\theta$ and $\varphi' = \varphi \wedge \phi\theta$, and then for (i) and (ii) above:

(i) It suffices to prove that $\mathcal{T}_{\mathcal{E}_0} \models \varphi'\rho$, i.e., $\mathcal{T}_{\mathcal{E}_0} \models (\varphi \wedge \phi\theta)\rho$. By assumption $\mathcal{T}_{\mathcal{E}_0} \models \varphi\rho$ and $\mathcal{T}_{\mathcal{E}_0} \models \phi\rho$. Notice that:

$$\phi\theta\rho = (\phi\theta|_{vars(l)})\rho =_{E_0 \uplus B_0} (\phi\rho)\rho = \phi\rho.$$

Hence $\mathcal{T}_{\mathcal{E}_0} \models \phi\theta\rho$.

(ii) By assumption $u' =_{E_0 \uplus B_0 \uplus B_1} r\rho$; also:

$$r\rho =_{E_0 \uplus B_0 \uplus B_1} r\theta|_{vars(l)}\rho = r\theta\rho = u\rho.$$

Hence $u' =_{E_0 \uplus B_0 \uplus B_1} u\rho \in [\![u]\!]_{\varphi'}$ by part (i).

5.1 Computing with $\rightsquigarrow_{\mathcal{R}^\circ}$

Although the above soundness and completeness theorems, plus Lemma 4, show that $\rightarrow_{\mathcal{R}}$ is characterized symbolically by $\rightsquigarrow_{\mathcal{R}^\circ}$, for any rewrite theory $\mathcal{R}$ modulo a built-in subtheory $\mathcal{E}_0$, a key question to ask is how to effectively compute this symbolic relation. More specifically, given a constrained term $(t\,;\varphi)$ in $T_\Sigma(X)_{State} \times QF_{\Sigma_0}(X_0)$, how can one compute all constrained terms $(u\,;\varphi')$ in $T_\Sigma(X)_{State} \times QF_{\Sigma_0}(X_0)$ such that $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}^\circ} (u\,;\varphi')$?

Given a rule $l^\circ \rightarrow r$ **if** ϕ in R° and according to the proof of Theorem 2, the existence of a substitution $\theta : X \longrightarrow T_\Sigma(X)$ satisfying $t =_E l^\circ\zeta\theta$ (i.e., Condition (a) in Definition 7) can be achieved by employing the strategy of first reducing t to its E_0/B_0-canonical form $t\downarrow_{E_0/B_0}$ (which exists and is unique by the admissibility of $\mathcal{E}_0$) and then trying to check if $t\downarrow_{E_0/B_0} \ll_{B_1} l^\circ\zeta$ via a matching algorithm modulo B_1 (which exists and is finitary by the admissibility of (Σ, E)). If the set of B_1-matching solutions produced by the matching algorithm is empty, then such a substitution θ does not exist for the given constrained term $(t\,;\varphi)$ and rule $l^\circ \rightarrow r$ **if** ϕ. Otherwise, each one of the B_1-matching solutions θ produced by the matching algorithm is such that $t =_{E_0 \uplus B_0} t\downarrow_{E_0/B_0} =_{B_1} l^\circ\zeta\theta$, i.e., $t =_E l^\circ\zeta\theta$; in this case, $u = r\zeta\theta$. Since the set of rules R° is finite and the matching algorithm is finitary, there are finitely many of such substitutions θ for a given constrained pair $(t\,;\varphi)$.

For checking Condition (b) in Definition 7 and given θ satisfying $t =_E l^\circ\zeta\theta$ as computed above, formula φ' can be chosen to be any quantifier-free formula in $QF_{\Sigma_0}(X_0)$ that is provably equivalent to $\varphi \wedge \phi\zeta\theta$ in $\mathcal{E}_0$. In particular, φ' can be chosen to be the formula $\varphi \wedge ((\phi\zeta\theta)\downarrow_{E_0/B_0})$.

Finally, checking Condition (c) in Definition 7 is in general undecidable. However, checking this condition becomes decidable for built-in theories $\mathcal{E}_0$ that can be extended to a *decidable theory* $\mathcal{E}_0^+$ (typically by adding some inductive consequences and, perhaps, some extra symbols) such that

$$(\forall \psi \in QF_{\Sigma_0}(X_0))\ \psi \text{ is } \mathcal{E}_0^+\text{-satisfiable} \iff (\exists \sigma : X_0 \longrightarrow T_{\Sigma_0})\ \mathcal{T}_{\mathcal{E}_0} \models \psi\sigma. \qquad (1)$$

Many decidable theories $\mathcal{E}_0^+$ of interest are supported by SMT solvers satisfying this requirement. For example, $\mathcal{E}_0$ can be the equational theory of natural number addition, i.e., $\mathcal{T}_{\mathcal{E}_0} = (\mathbb{N}, +, s, 0, <, \leq)$, and $\mathcal{E}_0^+$ Presburger arithmetic. That is, $\mathcal{T}_{\mathcal{E}_0}$ is the *standard model* of both $\mathcal{E}_0$ and $\mathcal{E}_0^+$, and $\mathcal{E}_0^+$-satisfiability coincides with satisfiability in such a standard model. Under such conditions, satisfiability of $\varphi \wedge \phi\zeta\theta$ (and therefore of φ') in a step $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}^\circ} (u\,;\varphi')$ becomes decidable by invoking an SMT-solver for $\mathcal{E}_0^+$.

Theorem 3 (Rewriting Modulo Axioms and Modulo SMT). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$ and $\mathcal{E}_0^+$ a theory extending $\mathcal{E}_0$ such that satisfiability of QF $\mathcal{E}_0^+$-formulas is decidable and $\mathcal{E}_0^+$-satisfiability coincides with satisfiability in $\mathcal{T}_{\mathcal{E}_0}$. Then $\rightsquigarrow_{\mathcal{R}^\circ}$ can be effectively computed.*

5.2 Symbolic Reachability Analysis

The goal of this section is to explain how rewriting modulo SMT can be used as a mechanism for solving *existential* reachability goals in the initial model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ modulo built-ins $\mathcal{E}_0$. This technique can be especially useful for symbolically proving or disproving safety properties of $\mathcal{R}$ such as, for instance, inductive invariants of $\mathcal{T}_{\mathcal{R}}$.

Consider the constrained terms $(t\,;\varphi)$ and $(u\,;\varphi')$, with $t, u \in T_{\Sigma}(X)_{State}$ and $\varphi, \varphi' \in QF_{\Sigma_0}(X_0)$. For many safety properties, the *existential reachability* question of whether there are concrete states $t' \in [\![t]\!]_{\varphi}$ and $u' \in [\![u]\!]_{\varphi'}$ such that $t' \rightarrow_{\mathcal{R}}^{*} u'$ is of particular interest, that is, whether from some state in $[\![t]\!]_{\varphi}$ is it possible to reach some state in $[\![u]\!]_{\varphi'}$. When $[\![u]\!]_{\varphi'}$ is a set of *bad* states, the idea is to know whether reaching a bad state is possible.

This intuitive formulation is almost right, but overlooks the fact that if $\overrightarrow{x} = vars(t;\varphi)$ and $\overrightarrow{y} = vars(u;\varphi')$, the set of *shared* variables $\overrightarrow{z} = \overrightarrow{x} \cap \overrightarrow{y}$ may be non empty, where the variables $\overrightarrow{z}$ can be called the *parameter* variables. The interpretation of those parameter variables $\overrightarrow{z}$ should then *agree* when instantiated to the concrete states $t' \in [\![t]\!]_{\varphi}$ and $u' \in [\![u]\!]_{\varphi'}$ such that $t' \rightarrow_{\mathcal{R}}^{*} u'$.

This can be formulated more precisely by saying that, given symbolic descriptions $(t\,;\varphi)$ of a set $[\![t]\!]_{\varphi}$ of *source* states, and $(u\,;\varphi')$ of a set $[\![u]\!]_{\varphi'}$ of *target* states, the interest is in settling the question of whether in the initial model $\mathcal{T}_{\mathcal{R}}$ the following existential reachability formula is satisfied:

$$\mathcal{T}_{\mathcal{R}} \models \left(\exists \overrightarrow{x} \cup \overrightarrow{y}\right)\ t \rightarrow_{\mathcal{R}}^{*} u \wedge \varphi \wedge \varphi'. \tag{2}$$

This, of course, exactly means settling whether there is a ground substitution ρ such that $t\rho \rightarrow_{\mathcal{R}}^{*} u\rho$ and $\mathcal{T}_{\mathcal{E}_0} \models \varphi\rho \wedge \varphi'\rho$, so that ρ interprets the parameter variables $\overrightarrow{z}$ in the exact same way in the source and the target states.

Recall the system comprising the thermostat and the air conditioning device presented in Example 3. In this example, $(t\,;\varphi)$ could be the source constrained term

$$(\langle\, time : 0, temp : T, setpoint : S, ac : false \,\rangle\,; true)$$

and $(u\,;\varphi')$ the target constrained term

$$([\, time : N_0, temp : N_1, setpoint : S, ac : B_0 \,]\,; B_1 \wedge N_1 \leq S \wedge B_0)\,,$$

so that the only parameter variable shared by both terms is S. An affirmative answer to the above reachability query would mean that from some ground initial state at time zero and in which the air conditioning device is off, a problematic state can be reached, namely, a state in which all zero-time transitions have taken place and in which the air conditioning device is turned on despite the fact that the temperature sensed from the environment does not exceed the system's setpoint.

The question, of course, is how to use symbolic rewriting to find answers to existential reachability queries of this kind. Since in rewriting modulo SMT there is a useful division of labor between *matching* pattern terms modulo B_1 and *SMT solving* of built-in constraints, the above existential formula needs to be slightly modified into an equivalent one, more suitable for technical reasons. Note that the set $\overrightarrow{z}$ of parameter variables

decomposes as a disjoint union $\overrightarrow{z} = \overrightarrow{z_0} \uplus \overrightarrow{z_1}$, where $\overrightarrow{z_0} \subset X_0$, and $\overrightarrow{z_1} \subset (X \setminus X_0)$. To be able to use matching modulo B_1, the idea is to have: (i) the built-in parameter variables $\overrightarrow{z_0}$ *not* to appear in u, but only in its condition φ', and (ii) u to be a Σ_1-term. This can easily be accomplished by an abstraction of built-ins for the original u in $(u\,;\varphi')$. That is, the Σ_0-subterms of u can be abstracted with fresh abstraction variables $\overrightarrow{y'} \subseteq X_0$, where u° is S_0-linear, and if $\overrightarrow{y'} = y_1, \ldots, y_n$ then $[\gamma]$ is the conjunction $y_1 = v_1 \wedge \cdots \wedge y_n = v_n$ associated to the substitution $\gamma = \{y'_1 \mapsto v_1, \ldots, y'_n \mapsto v_n\}$ such that $u = u^\circ\gamma$. This yields a reformulation of the above existential formula as the semantically equivalent one:

$$\mathcal{T}_\mathcal{R} \models \left(\exists \overrightarrow{x} \cup \overrightarrow{y} \cup \overrightarrow{y'}\right)\ t \rightarrow^*_\mathcal{R} u^\circ \ \wedge\ \varphi \ \wedge\ \varphi' \wedge [\gamma], \tag{3}$$

where two essential points are: (i) $[\![u]\!]_{\varphi'} = [\![u^\circ]\!]_{\varphi' \wedge [\gamma]}$, and (ii) the built-in parameter variables $\overrightarrow{z_0}$ no longer appear in u° but appear instead in $\varphi' \wedge [\gamma]$.

In the above example, since $\overrightarrow{z_0}$ consisted only of the variable S, this can be easily accomplished by reformulating the target constrained term $(u\,;\varphi')$ as the following constrained term $(u^\circ\,;\varphi' \wedge [\gamma])$:

$$([\,\mathit{time} : N_0, \mathit{temp} : N_1, \mathit{setpoint} : N_2, \mathit{ac} : B_0\,]\,; B_1 \wedge N_1 \leq N_2 \wedge B_0 \wedge N_2 = S\,).$$

Therefore, the existential reachability goal in (3) can be written for this example as

$$\begin{aligned}\mathcal{T}_\mathcal{R} \models (\exists T, S, N_0, N_1, N_2, B_0)\ \langle\, \mathit{time} : 0, \mathit{temp} : T, \mathit{setpoint} : S, \mathit{ac} : \mathit{false}\,\rangle \\ \rightarrow^*_\mathcal{R} [\,\mathit{time} : N_0, \mathit{temp} : N_1, \mathit{setpoint} : N_2, \mathit{ac} : B_0\,] \\ \wedge\ N_1 \leq N_2 \wedge B_0 \wedge N_2 = S.\end{aligned} \tag{4}$$

In general, thanks to the soundness and completeness results, Theorems 1 and 2, the solvability of existential reachability queries of the form (3) can be achieved by the symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}^\circ}$. This results in a sound and complete symbolic reachability analysis technique based on rewriting modulo SMT.

Theorem 4 (Symbolic Reachability Analysis). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$. The model-theoretic satisfaction relation (3) holds if and only if there exist a term $v \in T_\Sigma(X)_{State}$, a constraint $\psi \in QF_{\Sigma_0}(X_0)$, and a B_1-unifier[2] θ of the equation $v^\circ = u^\circ$ such that: (a) $(t\,;\varphi) \rightsquigarrow^*_{\mathcal{R}^\circ} (v\,;\psi)$, and (b) $(\psi \wedge [\eta] \wedge \varphi' \wedge [\gamma])\theta$ is $\mathcal{T}_{\mathcal{E}_0}$-satisfiable, where $(v^\circ;\eta)$ is an abstraction of built-ins for v.*

Proof. By Theorems 1 and 2, and induction on the length of the rewrite derivation. The key points to bear in mind are that: (i) $[\![v]\!]_\psi = [\![v^\circ]\!]_{\psi \wedge [\eta]}$ and therefore (ii) $[\![v]\!]_\psi \cap [\![u^\circ]\!]_{\varphi' \wedge [\gamma]} = [\![v^\circ]\!]_{\psi \wedge [\eta]} \cap [\![u^\circ]\!]_{\varphi' \wedge [\gamma]}$; but then, since v° and u° are Σ_1-terms, reachability is achieved, i.e., $[\![v]\!]_\psi \cap [\![u^\circ]\!]_{\varphi' \wedge [\gamma]} \neq \emptyset$, iff there is a B_1-unifier θ of the equation $v^\circ = u^\circ$ such that (b) holds.

[2] It is assumed that, in addition to a B_1-matching algorithm, there is also a finitary B_1-unification algorithm, so that a finite number of most general such unifiers can be effectively computed; but see Remark 1 below for an alternative formulation not requiring B_1-unification.

Note that φ is not included as a conjunct in the constraint $(\psi \wedge [\eta] \wedge \varphi' \wedge [\gamma])\theta$ of Condition (b) of Theorem 4 because $(t\,;\varphi) \rightsquigarrow_{\mathcal{R}^\circ} (v\,;\psi)$ implies that φ is a semantic consequence of ψ.

To be able to exploit Maude's efficient built-in *search* command, which is based on B_1-matching, the reflexive implementation described in Section 6 uses the following alternative reformulation of Theorem 4.

Remark 1. Theorem 4 can be reformulated in a way in which only B_1-matching, as opposed to B_1-unification, is required to solve existential reachability queries. The key point is that, since $\mathcal{R} = (\Sigma, E, R)$ is a topmost rewrite theory, there is one operator (or at most a finite number of them: the generalization to several such operators is straightforward), say $[\_, \ldots, \_] : s_1 \ldots s_n \to \mathit{State}$, typically not obeying any axioms[3] B_1, such that any Σ-term u of sort *State* is of the form $u = [u_1, \ldots, u_n]$, so that there is a substitution $\alpha = \{y''_1 \mapsto u_1, \ldots, y''_n \mapsto u_n\}$ with y''_i of sort s_i such that $u = [y''_1, \ldots, y''_n]\alpha$. This means that the semantic relation (2) can be reformulated as:

$$\mathcal{T}_{\mathcal{R}} \models \left(\exists \overrightarrow{x} \cup \overrightarrow{y} \cup \overrightarrow{y''}\right)\ t \to^*_{\mathcal{R}} [y''_1, \ldots, y''_n] \wedge \varphi \wedge \varphi' \wedge [\alpha]. \tag{5}$$

But this means that the model-theoretic satisfaction relation (5) holds if and only if there exist a term $v \in T_\Sigma(X)_{\mathit{State}}$, a constraint $\psi \in QF_{\Sigma_0}(X_0)$, and a substitution θ such that (a) $(t\,;\varphi) \rightsquigarrow^*_{\mathcal{R}^\circ} (v\,;\psi)$, (b) $v =_{B_1} [y''_1, \ldots, y''_n]\theta$, and (c) $\psi \wedge (\varphi' \wedge [\alpha])\theta$ is $\mathcal{T}_{\mathcal{E}_0}$-satisfiable.

Example 6, at the end of Section 6, illustrates how Remark 1 is useful in practice for querying the reachability goal (4) in the thermostat example by invoking Maude's *search* command. In this example, as pointed out in Remark 1, B_1-matching can be used for the purpose of solving existential queries via the symbolic relation $\rightsquigarrow_{\mathcal{R}}$ because the target term in the query is general enough and thus avoids the need for performing B_1-unification in the search process.

6 Reflective Implementation of $\rightsquigarrow_{\mathcal{R}^\circ}$

This section discusses the design and implementation of a prototype that offers support for symbolic rewriting modulo SMT in the Maude system. The prototype relies on Maude's meta-level features, which implements rewriting logic's reflective capabilities, and on SMT solving for $\mathcal{E}_0^+$ integrated in Maude as CVC3's decision procedures. The extension of Maude with CVC3 is available from the Matching Logic Project [59]. In the rest of this section, $\mathcal{R} = (\Sigma, E_0 \uplus B_0 \uplus B_1, R)$ is a rewrite theory modulo built-ins $\mathcal{E}_0$, where $\mathcal{E}_0$ satisfies Condition (1) in Section 5. The theory mapping $\mathcal{R} \mapsto \mathbf{u}(\mathcal{R})$, basically, makes the rules unconditional by removing the constraints ϕ in the conditions of the rules in R.

In Maude, reflection is efficiently supported by its *META-LEVEL* module [19], which provides key functionality for rewriting logic's *universal theory* $\mathcal{U}$ [20]. In particular, rewrite theories $\mathcal{R}$ are meta-represented in $\mathcal{U}$ as terms $\overline{\mathcal{R}}$ of sort *Module*, and

[3] If $[\_, \ldots, \_]$ were to obey any axioms in B_1, since we assume that there is a B_1-matching algorithm, we would just need a finite number $\alpha_1, \ldots, \alpha_k$ of matching substitutions instead of a single α, so that $[\alpha]$ would be replaced by $[\alpha_1] \vee \ldots \vee [\alpha_k]$.

a term t in $\mathcal{R}$ is meta-represented in $\mathcal{U}$ as a term $\bar{t}$ of sort *Term*. The key idea of the reflective implementation is to reduce symbolic rewriting with $\rightsquigarrow_{\mathcal{R}^\circ}$ to *standard rewriting* in an associated reflective rewrite theory that extends the universal theory $\mathcal{U}$. This reduction is specially important for formal analysis purposes, because it makes available to $\rightsquigarrow_{\mathcal{R}^\circ}$ some formal analysis features provided by Maude for rewrite theories such as ground reachability analysis by search. This capability is illustrated by the running example in Section 5, and by the case studies in sections 7 and 8.

The prototype defines a parametrized functional module $SAT(\Sigma_0, E_0 \uplus B_0)$ of quantifier-free formulas with Σ_0-equations as atoms. In particular, this module extends $(\Sigma_0, E_0 \uplus B_0)$ with new sorts *Atom* and *QFFormula*, and new *constants* $var(X_0)$ representing the variables X_0. It has, among other functions, a function $sat : QFFormula \longrightarrow Bool$ such that for ϕ, $sat(\phi) = \top$ if ϕ is $\mathcal{E}_0^+$-satisfiable, and $sat(\phi) = \bot$ otherwise.

The process of computing the one-step rewrites of a given constrained term $(t\,;\varphi)$ under $\rightsquigarrow_{\mathcal{R}^\circ}$ is decomposed into two conceptual steps using Maude's metalevel. First, all possible triples $(u\,;\theta\,;\phi)$ such that $t \rightarrow_{\mathbf{u}(\mathcal{R}^\circ)} u$ is witnessed by a matching substitution θ and a rule with constraint ϕ are computed[4]. Second, these triples are filtered out by keeping only those for which the quantifier-free formula $\varphi \wedge \phi\theta$ is $\mathcal{E}_0^+$-satisfiable.

The first step in the process is mechanized by function *next*, available from the parametrized module $NEXT(\overline{\mathcal{R}}, \overline{State}, \overline{QFFormula})$ where $\overline{\mathcal{R}}$, $\overline{State}$, and $\overline{QFFormula}$ are the metalevel representations, respectively, of the rewrite theory module $\mathcal{R}$, the state sort *State*, and the sort *QFFormula* for quantifier-free formulas. The function *next* uses Maude's *meta-match* function and the auxiliary function *new-vars* for computing fresh variables (see Section 5). In particular, the call

$$next(\overline{((S, \leq, F \uplus var(X_0)), E_0 \uplus B_0 \uplus B_1, R^\circ)}, \bar{t}, \overline{\varphi})$$

computes all possible triples $\left(\overline{u}\,;\overline{\theta'}\,;\overline{\phi'}\right)$ such that $t \rightsquigarrow_{\mathcal{R}^\circ} u$ is witnessed by a substitution θ' and a rule with constraint ϕ'. More precisely, such a call first computes a renaming $\zeta = fresh\text{-}vars(vars(t, \varphi))$ and then, for each rule$(l^\circ \rightarrow r$ **if** $\phi)$, it uses the function *meta-match* to obtain a substitution $\overline{\theta} \in meta\text{-}match(\overline{((S, \leq, F \uplus var(X_0)), B_0 \uplus B_1)}, \overline{t\downarrow_{E_0/B_0 \uplus B_1}}, \overline{l^\circ\zeta})$, and returns $\left(\overline{u}\,;\overline{\theta'}\,;\overline{\phi'}\right)$ with $\overline{u} = \overline{r\zeta\theta}$, $\overline{\theta'} = \overline{\zeta\theta}$, and $\overline{\phi'} = \overline{\phi\zeta\theta}$. Note that by having a *deterministic* choice of fresh variables (including those in the constraint), function *next* is actually a *deterministic* function.

Using the above-mentioned infrastructure, the parametrized module *NEXT* implements the symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}^\circ}$ as a *standard rewrite relation*, extending *META-LEVEL*, by means of the following conditional rewrite rule:

$$(X\,;C) \rightarrow (Y\,;C')$$
$$\textbf{if } \left(\overline{Y}\,;\overline{\theta}\,;\overline{\phi}\right)\ S := next(\overline{\mathcal{R}^\bullet}, \overline{X}, \overline{C})\ \wedge\ C' := (C \wedge \phi)\ \wedge\ sat(C')$$

where X, Y range over sort *State* and C, C' over sort *Bool*, and $\mathcal{R}^\bullet = ((S, \leq, F \uplus var(X_0)), B, R^\circ)$. Therefore, a call to an external SMT solver is just an invocation of the function *sat* in $SAT(\Sigma_0, E_0 \uplus B_0)$ in order to achieve the above functionality more efficiently and in a

[4] Note that in $\mathbf{u}(\mathcal{R}^\circ)$ variables in X_0 are interpreted as *constants*. Therefore, the number of matching substitutions θ thus obtained is finite.

built-in way. The *matching condition* [19]

$$\left(\overline{Y}\,;\overline{\theta}\,;\overline{\phi}\right)\ S\ :=\ next(\overline{\mathcal{R}^{\bullet}},\overline{X},\overline{C})$$

is a syntactic variant of an equational condition mathematically interpreted as an ordinary equation. Operationally, when executing this matching condition, the variables introduced in the left-hand side of the equation are instantiated by *matching* the term $\left(\overline{Y}\,;\overline{\theta}\,;\overline{\phi}\right)\ S$ against the canonical form of the term $next(\overline{\mathcal{R}^{\bullet}},\overline{X},\overline{C})$. Since matching substitutions need not be unique, this matching condition provides a convenient way to perform a search through the canonical form of the structure $next(\overline{\mathcal{R}^{\bullet}},\overline{X},\overline{C})$ without the need to explicitly define a function for this purpose.

Recall the existential reachability problem (5) in Section 5:

$$\mathcal{T}_{\mathcal{R}} \models \left(\exists\overrightarrow{x} \cup \overrightarrow{y} \cup \overrightarrow{y''}\right)\ t \rightarrow^{*}_{\mathcal{R}} [y''_1,\ldots,y''_n]\ \wedge\ \varphi\ \wedge\ \varphi'\ \wedge\ [\alpha].$$

Given that the symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}^{\circ}}$ is encoded as a standard rewrite relation, this reachability problem can be solved by symbolic search, *directly available* in Maude from its *search* command. More precisely, for solving this reachability goal, the following invocation of Maude's *search* command will find a solution, if one exists:

$$search\ \ (t\,;\varphi) \rightarrow^{*} ([y''_1,\ldots,y''_n]\,;C)\ \ \ such\ that\ \ sat(C\wedge\varphi'\wedge[\alpha]).$$

In this command, C is a built-in variable of sort *Bool* used for matching *any* constraint found in the search process. In this way, whenever a symbolic state is reached, the interest is in checking whether the constraint $C\wedge\varphi'\wedge[\alpha]$ is satisfiable, meaning that a witness in the set $\left[\!\left[[y''_1,\ldots,y''_n]\right]\!\right]_{\varphi'\wedge[\alpha]}$ of target states can be reached from at least one term in the set $[\![t]\!]_{\varphi}$ of source states.

Example 6. Recall the existential reachability problem (4) in Section 5 for the thermostat and the air conditioning system:

$$\begin{aligned}\mathcal{T}_{\mathcal{R}} \models (\exists T,S,N_0,N_1,N_2,B_0)\ \langle\, time : 0, temp : T, setpoint : S, ac : false\,\rangle\\ \rightarrow^{*}_{\mathcal{R}}\ [\, time : N_0, temp : N_1, setpoint : N_2, ac : B_0\,]\\ \wedge\ \ N_1 \leq N_2 \wedge B_0 \wedge N_2 = S.\end{aligned}$$

Since the target term $[\, time : N_0, temp : N_1, setpoint : N_2, ac : B_0\,]$ is a pattern satisfying the requirements found in Remark 1, there is not need for involving unification algorithms in the symbolic search process, as stated in Theorem 4. Instead, the following invocation of Maude's *search* command will find a solution to this query, if one exists, up to a depth search bound of 10:

```
search [,10] ( < time: 0, temp: T, setpoint: S, ac: false > ; true )
         =>* ( [ time: N0, temp: N1, setpoint: N2, ac: B0 ] ; B1 )
             such that sat(B1 and N1 <= N2 and B0 and N2 = S) .
```

As expected, this command terminates without finding any solution to the reachability query. It is key to note that the search command appearing in the above code snippet is

the one used in Maude for *ground* search, but thanks to the reflective implementation of rewriting modulo SMT in the Maude system, this same command can actually be used also for *symbolic* reachability analysis, as explained above. The pair `[,10]` in the above search command is an optional argument providing a bound on: (i) the number of desired solutions (first pair component) and (ii) the maximum depth of the search (second pair component): in this case, no bound is given for the number of desired solutions, but a bound of 10 is given for the maximum depth of the search task.

In general, and as witnessed by the case studies presented in sections 7 and 8, the prototype implementation discussed in this section has been successfully used for checking reachability properties of interesting open and real-time systems. However, as pointed out as part of the concluding remarks of this work in Section 9, some future work in incorporating various state space reduction techniques can be beneficial for handling a broader class of problems with rewriting modulo SMT.

7 Analysis of the CASH Algorithm

This section presents a case study, developed jointly with K. Bae, of a real-time system that can be symbolically analyzed in the prototype tool described in Section 6. The analysis applies model checking based on rewriting modulo SMT and it is about the symbolic analysis of the CASH algorithm [15], a real-time scheduling algorithm that attempts to maximize system performance while guaranteeing that critical tasks are executed in a timely manner. The CASH algorithm achieves this goal by maintaining a priority queue of unused execution budgets that can be reused by other jobs to maximize processor utilization. This algorithm poses non-trivial modeling and analysis challenges because it contains, for instance, an unbounded priority queue, that cannot be modeled in timed-automata formalisms, such as those of UPPAAL [43] or Kronos [67], which assume a finite discrete state. More details about the case study, including its full implementation, and the prototype tool can be found in [9].

The CASH algorithm was specified and analyzed in Real-Time Maude by explicit-state model checking in an earlier paper by P. C. Ölveczky and M. Caccamo [52], which showed that, under certain variations on both the assumptions and the design of the protocol, it could miss deadlines. Explicit-state model checking has intrinsic limitations which the new analysis by rewriting modulo SMT presented below overcomes. The CASH algorithm is parametrized by: (i) the number N of servers in the system, and (ii) the values of a maximum budget b_i and period p_i, for each server $1 \leq i \leq N$. Even if N is fixed, *there are infinitely many initial states* for N servers, since the maximum budgets b_i and periods p_i range over the natural numbers. Therefore, explicit state model checking cannot perform a full analysis. If a counterexample for N servers exists, it may be found by explicit-state model checking for some chosen initial states, as done in [52], but it could be missed if the wrong initial states are chosen.

Rewriting modulo SMT is useful for symbolically analyzing infinite-state systems like CASH. Infinite sets of states are symbolically described by constrained terms which may involve user-definable data structures such as priority queues, but whose only variables range over decidable types for which an SMT solving procedure is available. For

the CASH algorithm, the built-in sorts are *Int* and *Bool* ranging, respectively, over the integer numbers and the Boolean values.

7.1 Symbolic States

In the symbolic CASH algorithm specification $\mathcal{R}$, a symbolic state is a pair $(t\,;\varphi)$ of sort *Sys* in which the term $t \in T_{\Sigma}(X_0)_{Configuration}$ represents the symbolic state of execution of the algorithm and the formula $\varphi \in QF_{\Sigma_0}(X_0)$ is a constraint on t. In this symbolic specification, the built-in terms range over the sorts *Int* for integer numbers and *Bool* for Boolean values, and have the usual connectives. A symbolic state *t* has sort *Configuration*, representing multisets of objects, in which multiset union is denoted by juxtaposition (i.e., by the empty syntax). An object has sort *Object* and has the form $\langle \_ : \_ \mid \_ \rangle$, where the first argument is an object identifier having sort *Oid*, the second argument a class identifier having sort *Cid*, and the third argument is a multiset of attributes having sort *AttrSet*, where attribute union is denoted by comma. In each object configuration there is a global object (of class *global*) that models the time of the system (with attribute name *time*), the priority queue (with attribute name *cq*), the availability (with attribute name *available*), and a deadline missed flag (with attribute name *deadlineMiss*). A configuration can also contain any number of server objects (of class *server*). Each server object models the maximum budget (the maximum time within which a given job will be finished, with attribute name *maxBudget*), period (with attribute name *period*), internal state (with attribute name *state*), time executed (with attribute name *timeExecuted*), budget time used (with attribute name *usedOfBudget*), and time to deadline (with attribute name *timeToDeadline*).

7.2 Symbolic Transitions

The symbolic transitions of CASH are specified by 13 conditional rewrite rules whose conditions specify constraints solvable by the SMT decision procedure and some *extra* conditions, with the help of auxiliary functions, which are solvable by rewriting. The goal of the extra conditions is to minimize the number of (constrained) rewrite rules in the formal specification of the protocol.

In what follows, the following conventions are adopted:

- Variables *iNZT*, *iI*, *iT*, *inI1*, *inI2*, etc., and their primed versions have built-in sort *Int* and denote symbolic integer expressions.
- Variables ϕ, *iB*, *inB1*, *inB2*, etc., and their primed versions have built-in sort *Bool* and denote symbolic constraints.
- Variable *B* has sort *Boolean* and denotes non built-in Boolean values, i.e., Boolean values in the Maude language.
- Variable *REST* has sort *Configuration* and it is used in the rules to match parts of a state that are not relevant to a particular rule.
- Variables *G*, *O*, and *O'* have sort *Oid* and denote object identifiers.
- Variables *AtSG*, *AtS*, and *AtS'* have sort *AttrSet* (see Section 4).
- Variables *CQ*, *CQ1*, *CQ2*, and their primed versions have sort *Queue* and denote priority queues.

In the following rules, matching equations of the form _:=_ are extensively used. As explained in Section 6, they are mathematically interpreted as ordinary equations, but operationally the variables introduced in the left-hand side of the equation are instantiated by matching the canonical form of the instance of the term on the right-hand side. Intuitively, since matching condition is performed modulo axioms, a matching condition provides a convenient way to perform a search through the canonical form of the structure of the right-hand side term without the need to explicitly define a function for this purpose.

Rule [*idleToExecuting*] This rule models the situation in which an inactive server can start executing if the processor is available. In this case, the server transitions from state *idle* to *executing*, with zero use of its budget and execution time, and the system's processor is made unavailable.

$$
\begin{array}{ll}
 & (REST\langle G : global \mid available : true, AtS\, G\rangle \\
 & \langle O : server \mid period : iNZT, state : idle, timeToDeadline : iT, \\
 & \ timeExecuted : inI1, usedOfBudget : inI2, AtS\rangle; \phi) \\
\rightarrow & (REST\langle G : global \mid available : false, AtS\, G\rangle \\
 & \langle O : server \mid period : iNZT, state : executing, timeToDeadline : iI, \\
 & \ timeExecuted : 0, usedOfBudget : 0, AtS\rangle; \phi') \\
\textbf{if} & (int : iI\ const : iB); ECS := \\
 & \qquad (int : iT + iNZT\ const : iT > 0); (int : iNZT\ const : iT \leq 0) \\
\wedge & \phi' := \phi \wedge iNZT > 0 \wedge iB \\
\wedge & sat(\phi')
\end{array}
$$

Auxiliary function symbol *int:_const:_* represents a pair with first element an integer expression of sort *Int* and second argument a constraint of sort *Bool*. In the case of this rule, there are two of these constructs representing the situations in which *timeToDeadline* can be positive and negative. Depending on the situation, quantity *timeToDeadline* can be updated in two different ways, each specified by the corresponding symbolic integer expression. Note that because constraint *iNZT > 0*, this rule is enabled only in a state having a server with positive period.

Rules [*idleToActiveP*] *and* [*idleToActiveN*] These rules model the situation where a server becomes active and another server is executing, which in turn will either preempt (rule [*idleToActiveP*]) or not (rule [*idleToActiveN*]) according to its internal state. Rules

[*idleToActiveP*] and [*idleToActiveN*], in that order, are presented below.

$$
\begin{array}{ll}
 & (REST\langle O : server \mid period : iNZT, state : idle, timeToDeadline : iT, \\
 & \ timeExecuted : inI1, usedOfBudget : inI2, AtS\rangle \\
 & \ \langle O' : server \mid timeToDeadline : iT', state : executing, AtS'\rangle; \phi) \\
\rightarrow & (REST\langle O : server \mid period : iNZT, state : executing, \\
 & \ timeToDeadline : iI, timeExecuted : 0, usedOfBudget : 0, AtS\rangle \\
 & \ \langle O' : server \mid timeToDeadline : iT', state : waiting, AtS'\rangle; \phi') \\
\textbf{if} & (int : iI\ const : iB); ECS := \\
 & \qquad (int : iT + iNZT\ const : (iT > 0 \wedge iT' > 0 \wedge iT + iNZT < iT')); \\
 & \qquad (int : iNZT\ const : (iT \leq 0 \wedge iT' > 0 \wedge iNZT < iT')) \\
\wedge & \phi' := \phi \wedge iNZT > 0 \wedge iB \\
\wedge & sat(\phi')
\end{array}
$$

$$
\begin{array}{ll}
 & (REST\langle O : server \mid period : iNZT, state : idle, timeToDeadline : iT, \\
 & \ timeExecuted : inI1, usedOfBudget : inI2, AtS\rangle \\
 & \ \langle O' : server \mid state : executing, timeToDeadline : iT', AtS'\rangle; \phi) \\
\rightarrow & (REST\langle O : server \mid period : iNZT, state : waiting, \\
 & \ timeToDeadline : iI, timeExecuted : 0, usedOfBudget : 0, AtS\rangle \\
 & \ \langle O' : server \mid state : executing, timeToDeadline : iT', AtS'\rangle; \phi') \\
\textbf{if} & (int : iI\ const : iB); ECS := \\
 & \qquad (int : iT + iNZT\ const : (iT > 0 \wedge iT + iNZT \geq iT')); \\
 & \qquad (int : iNZT\ const : (iT \leq 0 \wedge iNZT \geq iT')) \\
\wedge & \phi' := \phi \wedge iNZT > 0 \wedge iB \\
\wedge & sat(\phi')
\end{array}
$$

Rules [*stopExecuting1A*] *and* [*stopExecuting1B*] These rules model the situation where a server finishes execution and there is at least one server waiting. In this case, the first waiting server in the queue starts executing. Also, if there is any budget left, it is added to the global CASH. Below, rules [*stopExecuting1A*] and [*stopExecuting1B*] are

presented.

$$
\begin{array}{ll}
 & (REST\langle G : global \mid cq : CQ\ CQ', AtS\, G\rangle \\
 & \langle O : server \mid state : executing, usedOfBudget : iT, maxBudget : iNZT, \\
 & \quad timeToDeadline : iT', timeExecuted : iNZT', period : iNZT'', AtS\,\rangle \\
 & \langle O' : server \mid state : waiting, timeToDeadline : iT'', AtS'\rangle; \phi) \\
\rightarrow & (REST \\
 & \langle G : global \mid cq : (CQ\ (deadline : iT' budget : (iNZT - iT))\ CQ'), AtS\, G\rangle \\
 & \langle O : server \mid state : idle, usedOfBudget : iNZT, maxBudget : iNZT, \\
 & \quad timeToDeadline : iT', timeExecuted : iNZT', period : iNZT'', AtS\,\rangle \\
 & \langle O' : server \mid state : executing, timeToDeadline : iT'', AtS'\rangle; \phi') \\
\textbf{if} & ALL := \cdots \\
\wedge & inB1 := nextDeadlineWaiting(ALL, O, iT'') \\
\wedge & inB2 := belowDeadline(iT', CQ) \\
\wedge & inB3 := aboveOrEqualDeadline(iT', CQ') \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iNZT > 0 \wedge iNZT' > 0 \wedge iNZT'' > 0 \wedge iNZT > iT \wedge \\
 & \quad iT' > 0 \wedge iNZT \leq iT + iT' \wedge inB1 \wedge inB2 \wedge inB3 \\
\wedge & sat(\phi')
\end{array}
$$

Variable *ALL*, whose specification has been omitted, represents the entire object configuration in the left hand side of the rule. Function call *nextDeadlineWaiting(ALL,O,iT")* computes a constraint over all waiting servers in *ALL* different from *O* that is satisfiable by any of such servers whose *timeToDeadline* attribute is at least *iT"*. If the system has missed a deadline, this constraint is unsatisfiable. Function call *belowDeadline(iT',CQ)* computes a constraint that is satisfiable if and only if all deadlines in *CQ* are less than *IT'*. Analogously, function call *aboveOrEqualDeadline(iT',CQ')* computes a constraint that is satisfiable if and only if all deadlines in *CQ'* are at least *IT'*. Note that these two functions are used together in order to keep the representation invariant of the system's priority queue when inserting a new element into it.

The following is the specification of rule [*stopExecuting1B*]:

$$
\begin{array}{ll}
 & (REST \\
 & \langle O : server \mid state : executing, usedOfBudget : iT, maxBudget : iNZT, \\
 & \quad timeToDeadline : iT', timeExecuted : iNZT', period : iNZT'', AtS \rangle \\
 & \langle O' : server \mid state : waiting, timeToDeadline : iT'', AtS' \rangle ; \phi) \\
\rightarrow & (REST \\
 & \langle O : server \mid state : idle, usedOfBudget : iNZT, maxBudget : iNZT, \\
 & \quad timeToDeadline : iT', timeExecuted : iNZT', period : iNZT'', AtS \rangle \\
 & \langle O' : server \mid state : executing, timeToDeadline : iT'', AtS' \rangle ; \phi') \\
\textbf{if} & ALL := \cdots \\
 & inB1 := nextDeadlineWaiting(ALL, O, iT'') \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iNZT > 0 \wedge iNZT' > 0 \wedge iNZT'' > 0 \wedge \\
 & \quad iNZT \leq iT \wedge inB1 \\
\wedge & sat(\phi')
\end{array}
$$

Rules [*stopExecuting2A*] *and* [*stopExecuting2B*] These two rules complement the previous two rules for situations where a server finishes execution and there is no server waiting. The effect on the system is that the processor that was being used by the finishing server is released.

The following is the specification of rule [*stopExecuting2A*]:

$$
\begin{array}{ll}
 & (REST \langle G : global \mid cq : CQ\ CQ', available : B, AtS\, G \rangle \\
 & \langle O : server \mid state : executing, usedOfBudget : iT, maxBudget : iNZT, \\
 & \quad timeToDeadline : iT', timeExecuted : iNZT', period : iNZT'', AtS \rangle ; \phi) \\
\rightarrow & (REST \\
 & \langle G : global \mid cq : (CQ\, (deadline : iT' budget : (iNZT - iT))\, CQ'), \\
 & \quad available : true, AtS\, G \rangle \\
 & \langle O : server \mid state : idle, usedOfBudget : iNZT, maxBudget : iNZT, \\
 & \quad timeToDeadline : iT', timeExecuted : iNZT', period : iNZT'', AtS \rangle ; \phi') \\
\textbf{if} & ALL := \cdots \\
\wedge & inB1 := belowDeadline(iT', CQ) \\
\wedge & inB2 := aboveOrEqualDeadline(iT', CQ') \\
\wedge & inB3 := noServerWaiting(ALL, O)) \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iNZT > 0 \wedge iNZT' > 0 \wedge iNZT'' > 0 \wedge iNZT > iT \wedge \\
 & \quad iT' > 0 \wedge iNZT \leq iT + iT' \wedge inB1 \wedge inB2 \wedge inB3 \\
\wedge & sat(\phi')
\end{array}
$$

The following is the specification of rule [*stopExecuting2B*]:

$$
\begin{array}{ll}
 & (REST\langle G : global \mid available : B, AtS\, G\rangle \\
 & \langle O : server \mid state : executing, usedOfBudget : iT, timeToDeadline : iT', \\
 & \quad maxBudget : iNZT, timeExecuted : iNZT', period : iNZT'', AtS\rangle; \phi) \\
\rightarrow & (REST\langle G : global \mid available : true, AtS\, G\rangle \\
 & \langle O : server \mid state : idle, usedOfBudget : iNZT, timeToDeadline : iT', \\
 & \quad maxBudget : iNZT, timeExecuted : iNZT', period : iNZT'', AtS\rangle; \phi') \\
\textbf{if} & ALL := \cdots \\
\wedge & inB1 := noServerWaiting(ALL, O)) \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iNZT > 0 \wedge iNZT' > 0 \wedge iNZT'' > 0 \wedge \\
 & \quad iNZT \leq iT \wedge\ inB1 \\
\wedge & sat(\phi')
\end{array}
$$

Rule [*deadlineMiss*] This rule models the detection of a deadline miss for a server with non-zero maximum budget, i.e., a situation where the system has reached an overflow and the allocated execution time cannot be exhausted before the server's deadline.

$$
\begin{array}{ll}
 & (REST\langle G : global \mid deadlineMiss : B, AtS\, G\rangle \\
 & \langle O : server \mid state : S\,t, usedOfBudget : iT, timeToDeadline : iT', \\
 & \quad maxBudget : iNZT, AtS\rangle; \phi) \\
\rightarrow & (REST \\
 & \langle G : global \mid deadlineMiss : true, AtS\, G\rangle \\
 & \langle O : server \mid state : S\,t, usedOfBudget : iT, timeToDeadline : iT', \\
 & \quad maxBudget : iNZT, AtS\rangle; \phi') \\
\textbf{if} & St \neq idle \\
\wedge & (int : iI\ const : iB); ECS := \\
 & \quad (int : iT\ const : (iT' > 0 \wedge iNZT > iT + iT')); \\
 & \quad (int : iT\ const : (iT' \leq 0 \wedge iNZT > iT)) \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iNZT > 0 \wedge iB \\
\wedge & sat(\phi')
\end{array}
$$

The following rules are included in the specification for modeling a job which is longer than the execution time in one round of the server. This setting is considered in the rest of the rules, where an idle server may be immediately activated again.

Rules [*continueExInNextRound*], [*continueActInNextRound1*] *and* [*continueActInNextRound2*] These rules model the situation in which a server has executed all it can in the current round but wishes to continue executing in the next round. Since the server's deadline is increased, it cannot just continue executing but must check if some waiting server suddenly gets a shorter deadline.

Rule [*continueExInNextRound*] considers the case in which no other server is waiting when a server wishes to continue executing in the next round.

$$
\begin{array}{ll}
 & (REST \\
 & \langle O : server \mid state : executing, maxBudget : iNZT, usedOfBudget : iNZT', \\
 & \quad period : iNZT'', timeToDeadline : iT, timeExecuted : inI1, AtS \rangle ; \phi) \\
\rightarrow & (REST \\
 & \langle O : server \mid state : executing, maxBudget : iNZT, usedOfBudget : 0, \\
 & \quad period : iNZT'', timeToDeadline : iI, timeExecuted : 0, AtS \rangle ; \phi') \\
\textbf{if} & ALL := \cdots \\
\wedge & inB1 := eval(ALL, noServerWaiting(O)) \\
\wedge & \phi' := \phi \wedge iNZT > 0 \wedge iNZT'' > 0 \wedge iB \wedge inB1 \\
\wedge & sat(\phi')
\end{array}
$$

Rule [*continueActInNextRound1*] considers the case in which some other server is waiting and the server willing to continue executing becomes preempted.

$$
\begin{array}{ll}
 & (REST \\
 & \langle O : server \mid state : executing, maxBudget : iNZT, usedOfBudget : iNZT', \\
 & \quad period : iNZT'', timeExecuted : inI1, timeToDeadline : iT, AtS \rangle \\
 & \langle O' : server \mid state : waiting, timeToDeadline : iT', AtS' \rangle ; \phi) \\
\rightarrow & (REST \\
 & \langle O : server \mid state : waiting, maxBudget : iNZT, usedOfBudget : 0, \\
 & \quad period : iNZT'', timeExecuted : 0, timeToDeadline : iI, AtS \rangle \\
 & \langle O' : server \mid state : executing, timeToDeadline : iT', AtS' \rangle ; \phi') \\
\textbf{if} & ALL := \cdots \\
\wedge & inB1 := nextDeadlineWaiting(ALL, O, iT')) \\
\wedge & \phi' := \phi \wedge iNZT > 0 \wedge iNZT'' > 0 \wedge inB1 \wedge iB \\
\wedge & sat(\phi')
\end{array}
$$

Rule [*continueActInNextRound2*] considers the case in which some other server is waiting but the server willing to continue executing can do so.

$$
\begin{array}{ll}
 & (REST \\
 & \langle O : server \mid state : executing, maxBudget : iNZT, usedOfBudget : iNZT', \\
 & \quad period : iNZT'', timeExecuted : inI1, timeToDeadline : iT, AtS \rangle \\
 & \langle O' : server \mid state : waiting, timeToDeadline : iT', AtS' \rangle ; \phi) \\
\rightarrow & (REST \\
 & \langle O : server \mid state : executing, maxBudget : iNZT, usedOfBudget : 0, \\
 & \quad period : iNZT'', timeExecuted : 0, timeToDeadline : iT + iNZT'', AtS \rangle \\
 & \langle O' : server \mid state : waiting, timeToDeadline : iT', AtS' \rangle ; \phi') \\
\textbf{if} & (int : iI\ const : iB); ECS := \\
 & \quad (int : iT + iNZT''\ const : (iT > 0 \wedge iT' \geq iT + iNZT'')); \\
 & \quad (int : iNZT''\ const : (iT \leq 0 \wedge iT' \geq iNZT'')) \\
\wedge & ALL := \cdots \\
\wedge & \phi' := \phi \wedge iNZT > 0 \wedge iNZT'' > 0 \wedge inB1 \wedge iT' \geq iT + iNZT'' \\
\wedge & sat(\phi')
\end{array}
$$

Rules [*tickExecutingSpareCapacity*] *and* [*tickExecutingOwnBudget*] These two rules are directly involved with modeling the timed behavior of the protocol. In both rules the time is increased by 1 unit.

Rule [*tickExecutingSpareCapacity*] models the situation in which time elapses when a server is executing a spare capacity.

$$
\begin{array}{ll}
 & (REST \\
 & \langle G : global \mid time : iT, cq : (deadline : iI1\ budget : iI2)\ CQ, AtS\, G\rangle \\
 & \langle O : server \mid state : executing, timeExecuted : iT', timeToDeadline : iT'', \\
 & \quad AtS\,\rangle; \phi) \\
\rightarrow & (deltaServers(REST, 1) \\
 & \langle G : global \mid time : iT + 1, cq : delta(CQ2, 1), AtS\, G\rangle \\
 & \langle O : server \mid state : executing, timeExecuted : iT' + 1, \\
 & \quad timeToDeadline : iT'' - 1, AtS\,\rangle; \phi') \\
\textbf{if} & (queue : CQ1\ CQ2\ const : iB); ECS' := \\
 & \qquad\qquad usc1((deadline : iI1\ budget : iI2)\ CQ) \\
\wedge & ALL := \cdots \\
\wedge & inB1 := mteServer(ALL, O, 1) \\
\wedge & inB2 := mteQueue((deadline : iI1\ budget : iI2)\ CQ, 1) \\
\wedge & inB3 := noDeadlineMiss(ALL) \\
\wedge & inB4 := belowDeadline(2, CQ1) \wedge aboveOrEqualDeadline(2, CQ2) \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iT' \geq 0 \wedge iB \wedge inB1 \wedge inB2 \wedge iT'' \geq 1 \wedge \\
 & \quad inB3 \wedge iI1 \leq iT'' \wedge inB4 \\
\wedge & sat(\phi')
\end{array}
$$

Function call *deltaServers(REST,1)* updates attribute *timeToDeadline* in non-executing servers in *REST* by decreasing its value by 1 unit. Analogously, the function call *delta(CQ2,1)* decreases the value associated to *deadline* in 1 unit for each element in the priority queue *CQ2*. Given a non-empty queue, the function call

usc1((deadline: iI1 budget: iI2) CQ)

performs case splitting on the first element of the queue: it considers the case in which the budget *iI2* is at most 1 and the case when this quantity is more than 1. In either case, the server continues executing, but with different contents in the priority queue. Auxiliary functions *mteServer* and *mteQueue* generate constraints that simulate the time increment by 1 unit for servers and values in the priority queue, respectively.

Rule [*tickExecutingOwnBudget*] models the situation in which time elapses when a server is executing its own budget.

$$
\begin{array}{ll}
 & (REST \\
 & \langle G : global \mid time : iT, cq : CQ\ CQ', AtS\,G\rangle \\
 & \langle O : server \mid state : executing, timeExecuted : iT', \\
 & \quad usedOfBudget : iT'', timeToDeadline : iT''', AtS\,\rangle; \phi) \\
\rightarrow & (delta\text{-}servers(REST, 1) \\
 & \langle G : global \mid time : iT + 1, cq : delta(CQ', 1), AtS\,G\rangle \\
 & \langle O : server \mid state : executing, timeExecuted : iT' + 1, \\
 & \quad usedOfBudget : iT'' + 1, timeToDeadline : iT''' - 1, AtS\,\rangle; \phi') \\
\textbf{if} & ALL := \cdots \\
\wedge & inB1 := mteServer(ALL, G, 1) \\
\wedge & inB2 := mteQueue(CQ\ CQ', 1) \\
\wedge & inB3 := noDeadlineMiss(ALL) \\
\wedge & inB4 := lessThanFirstDeadline(iT''', CQCQ') \\
\wedge & inB5 := belowDeadline(2, CQ) \wedge aboveOrEqualDeadline(2, CQ') \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iT' \geq 0 \wedge iT'' \geq 0 \wedge inB1 \wedge inB2 \wedge inB3 \wedge inB4 \wedge inB5 \\
\wedge & sat(\phi')
\end{array}
$$

Rule [*tickIdle*] Finally, rule [*tickIdle*] models the increase of time by 1 unit in the entire system.

$$
\begin{array}{ll}
 & (REST\langle G : global \mid time : iT, cq : CQ, available : true, AtS\,G\rangle; \phi) \\
\rightarrow & (deltaServers(REST, 1) \\
 & \langle G : global \mid time : iT + 1, cq : delta(CQ2, 1), available : true, AtS\,G\rangle; \phi') \\
\textbf{if} & (queue : CQ1\ CQ2\ const : iB); ECS' := usc1(CQ) \\
\wedge & ALL := \cdots \\
\wedge & inB1 := mteServer(ALL, G, 1) \\
\wedge & inB2 := noDeadlineMiss(ALL) \\
\wedge & inB3 := belowDeadline(2, CQ1) \wedge aboveOrEqualDeadline(2, CQ2) \\
\wedge & \phi' := \phi \wedge iT \geq 0 \wedge iB \wedge inB1 \wedge inB2 \wedge inB3 \\
\wedge & sat(\phi')
\end{array}
$$

7.3 Symbolic Detection of Missed Deadlines

The goal is to verify *symbolically* the existence of missed deadlines of the CASH algorithm for the *infinite set of initial configurations* containing two server objects s_0 and s_1 with maximum budgets b_0 and b_1 and periods p_0 and p_1 as unspecified natural numbers, and such that each server's maximum budget is strictly smaller than its period, i.e.,

$0 \leq b_0 < p_0 \wedge 0 \leq b_1 < p_1$. This infinite set of initial states is specified symbolically by the equational definition (not shown) of term *init*. Maude's *search* command can then be used, as explained in Section 6, to symbolically check if there is a reachable state for any ground instance of *init* that misses the deadline:

```
search  init =>* ( Cnf < g : global | AtS, deadlineMiss : true > ; iB ) .
Solution 1 (state 233)
states: 234  rewrites: 60517 in 2865ms cpu (2865ms real) (21118 rewrites/second)
Cnf -->
  < s1 : server | maxBudget : X0, period : X1, state : waiting,
    usedOfBudget : 0, timeToDeadline : ((X1 - 1) - 1), timeExecuted : 0 >
  < s2 : server | maxBudget : X2, period : X3, state : executing,
    usedOfBudget : 2, timeToDeadline : ((X3 - 1) - 1), timeExecuted : 2 >
AtS --> time : 2, cq : emptyQueue, available : false
iB --> ((X0 <= 0 ^ X1 <= 0) v X0 <= 0 + X1 ^ ...
```

A counterexample is found at (modeling) time two, after exploring 233 symbolic states in less than 3 seconds. By using a satisfiability witness of the constraint *iB* computed by the search command, a concrete counterexample is found by exploring only 54 ground states. This result compares favorably, in both time and computational resources, with the ground counterexample found by explicit-state model checking in [52], where more that 52,000 concrete states were explored before finding a counterexample.

8 Symbolic Reachability Analysis for PLEXIL Modulo Integer Constraints

This section gives an overview of, and presents a case study about, the analysis of reachability properties for the *Plan Execution Interchange Language* (PLEXIL) [26] that can be expressed in rewriting modulo SMT and executed with the help of the prototype tool in Section 6 and the Maude Model Checker [19]. The symbolic reachability analysis for PLEXIL presented in this section is able to automatically detect reachability violations on input plans, where the values of external variables can be left unspecified, for a large subset of the language. More details on the symbolic specification of PLEXIL and the analysis performed on it can be found at [55]. Moreover, this section assumes some basic knowledge on LTL model checking in the Maude system [19].

PLEXIL is a synchronous language developed by NASA to support autonomous spacecraft operations. Synchronous languages were introduced in the 1980s to program *reactive systems*, i.e., open systems whose behavior is determined by their continuous reaction to the environment where they are deployed. Given the safety-critical nature of spacecraft operations, PLEXIL's operational semantics has been formally defined [23] and several properties of the language, such as determinism and compositionality, have been mechanically verified [22]. A rewriting logic semantics of PLEXIL has been previously developed in Maude and has been used, within a formal interactive verification environment [24, 56], to validate the intended semantics of the language against a wide variety of plan examples. The symbolic specification of PLEXIL used in this section extends and complements the *ground* rewriting logic semantics of the language with

symbolic reachability analysis, a task that is impossible to achieve with the rewriting logic semantics of the language [24].

PLEXIL programs define reactive systems that interact with an external environment of sensors and actuators. Such programs are *deterministic* by assuming a given concrete value for each of the sensors that the reactive system interacts with. Therefore, to execute by standard rewriting the rewriting logic semantics in [24] (and perform various kinds of reachability analysis verification in Maude), *concrete values of the data in sensors* had to be assumed for the reactive interactions. Since, in general, the possible tuples of such values can be infinite or (assuming finite arithmetic precision) extremely large, the concrete executions and formal analyses allowed by the concrete rewriting semantics had to be necessarily incomplete. This is analogous to the incompleteness of simulating and analyzing the CASH algorithm example in Real-Time Maude, versus the complete analysis by rewriting modulo SMT presented in Section 7. Using rewriting modulo SMT, symbolic analysis based on the rewriting logic semantics for PLEXIL can symbolically cover all possible values in an external environment [55].

8.1 PLEXIL Overview

This section presents an overview of PLEXIL; the reader is referred to [26] for a detailed description of the language.

A PLEXIL program, called a *plan*, is a tree of *nodes* representing a hierarchical decomposition of tasks. Interior nodes, called *list nodes*, provide control structure and naming scope for local variables. The primitive actions of a plan are specified in the leaf nodes. Leaf nodes can be *assignment nodes*, which assign values to local variables, *command nodes*, which call external commands, or *empty nodes*, which do nothing. PLEXIL plans interact with a functional layer that provides the interface with the external environment. This functional layer executes the external commands and communicates the status and result of their execution to the plan through *external variables*.

Nodes have an *execution state*, which can be *inactive*, *waiting*, *executing*, *failing*, *iterationend*, *finishing*, or *finished*, and an *execution outcome*, which can be *unknown*, *skipped*, *success*, or *failure*. They can declare local variables that are accessible to the node in which they are declared and all its descendants. In contrast to local variables, the execution state and outcome of a node are visible to all nodes in the plan. Assignment nodes also have a *priority*, which can help in solving race conditions. The *internal state* of a node consists of the current values of its execution state, execution outcome, and local variables.

Each node is equipped with a set of *gate conditions* and *check conditions* that govern the execution of a plan. Gate conditions provide control flow mechanisms that react to external events. In particular, the *start condition* specifies when a node starts its execution, the *end condition* specifies when a node ends its execution, the *repeat condition* specifies when a node can repeat its execution, and the *skip condition* specifies when the execution of a node can be skipped. Check conditions are used to signal abnormal execution states of a node and they can be either *pre-condition*, *post-condition*, or *invariant* conditions. The language includes Boolean, integer, and floating-point arithmetic, and string expressions. It also includes *lookup expressions* that read the value of external variables provided to the plan through the executive. Expressions appear in conditions,

assignments, and arguments of commands. Each of the basic types is extended by a special value *unknown* that can result, for example, when a lookup fails.

The execution of a plan in PLEXIL is driven by external events from the environment that trigger changes in the gate conditions. All nodes affected by a change in a gate condition synchronously respond to the event by modifying their internal state. These internal modifications may trigger more changes in gate conditions that in turn are synchronously processed until quiescence is reached for all nodes involved. External events are considered in the order in which they are received. An external event and all its cascading effects are processed before the next event is considered. This behavior is known as *run-to-completion semantics*.

The *atomic relation* describes the execution of an individual node in terms of state transitions triggered by changes in the environment. The *micro relation* describes the *synchronous* reduction of the atomic relation with respect to the *maximal redexes strategy*, i.e., the synchronous application of the atomic relation to the maximal set of nodes of a plan. The remaining three relations are the *quiescence relation*, the *macro relation*, and the *execution relation* that describe, respectively, the reduction of the micro relation until normalization, the interaction of a plan with the external environment upon one external event, and the n-iteration of the macro relation corresponding to n time steps. Figure 1 depicts the transition diagram defining PLEXIL's atomic transitions for lists in state *executing*. According to this diagram, when a list node is in state *executing*, the only way for it to reach state *finishing* is whenever the invariant of its ancestor node and its own invariant, together with its end condition, are all true. In any other case, this node's execution fails.

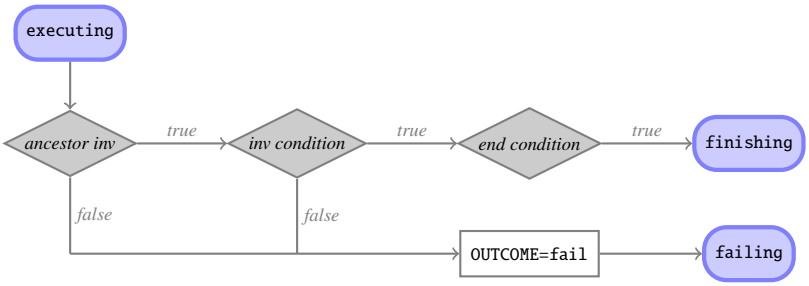

Fig. 1. Atomic transitions for list nodes in state *executing*.

Since local variables declared in a node are shared by its children nodes, it may be possible that two nodes attempt to synchronously write the same variable. The priority mechanism included in the semantics of PLEXIL can be used by programmers to deal with this problem. Unfortunately, priorities are optional and, in practice, race conditions may occur during the execution of a PLEXIL program. For instance, consider the plan *AssignWithConflig* in Figure 2. This plan has one list node and two assignment nodes, *NonNeg* and *NonPos*. It declares a local integer memory x and interacts with the external environment via the integer variable S. Note that depending on the value of S, the

assignment nodes *NonNeg* and *NonPos* may or may not start execution, and a race condition can happen on x when the value of S is 0. With the symbolic semantics presented in this section, the race condition on x can be automatically detected.

```
AssignWithConflict: {
  Integer       x = 0;
  Invariant:    x >= 0;
  NodeList:
  NonNeg: {
    Start:      Lookup(S) >= 0;
    Assignment: x := 1;
  }
  NonPos: {
    Start:      Lookup(S) <= 0;
    Assignment: x := 2;
  }
}
```

Fig. 2. A PLEXIL plan with a parallel assignment having a potential race condition.

8.2 Symbolic Detection of Race Conditions

In the symbolic rewriting logic semantics of PLEXIL $\mathcal{R}$, a symbolic state is a pair $(t\,;\varphi)$ in which the term $t \in T_{\Sigma}(X_0)_{Configuration}$ represents the symbolic state of execution of a plan and the formula $\varphi \in QF_{\Sigma_0}(X_0)$ is a constraint on t. In this rewriting logic semantics, the built-in terms range over the sorts *Int* for integer numbers and *Bool* for Boolean values, and are used to specify values of external variables under the control of the environment that can be left unspecified. Similar to the CASH specification in Section 7, a state t has sort *Configuration*, representing multisets of objects, in which multiset union is denoted by juxtaposition (i.e., by the empty syntax). An object has sort *Object* and has the form $\langle \_ : \_ \mid \_ \rangle$, where the first argument is an object identifier having sort *Oid*, the second argument a class identifier having sort *Cid*, and the third argument is a multiset of attributes having sort *AttrSet*, where attribute union is denoted by comma. A node in a PLEXIL plan is represented by an object.

As mentioned above, detection of race conditions on local memories and violation of node invariants are important in PLEXIL. As such, predicates for checking these predicates are already available from the symbolic rewriting logic semantics. In particular, states predicates *inv* and *race-free* are offered to the user; both predicates take as input the identifier of a node in a plan and check if the corresponding condition holds for that node in a given symbolic state. For this purpose, operator $\_ \models \_$ from the Maude Model Checker is used to equationally define the semantics of the relevant state predicates, where the first argument is a symbolic state $(t\,;\varphi)$ and the second the predicate

π being defined on that state, associating a Kripke structure to the initial reachability model of the PLEXIL specification (see [19] for more details). For example, the following equation defines the satisfiabilty of predicate *inv* w.r.t. a symbolic state:

$$(\langle O : C \mid inv{:}B, AtS\rangle\ Cnf ; B') \models inv(O)$$
$$= unsat(B' \wedge \neg B)$$

where the variable O has sort *Oid*, the variable C has sort *Cid*, the variables B, B' have sort *Bool*, the variable *AtS* has sort *AttrSet*, and the variable *Cnf* has sort *Configuration*. Function *unsat* is used to check for the *unsatisfiability* of a given constraint. In this case, the invariant condition of a node O, represented here by the built-in variable B, yields an invariant violation whenever the conjunction of the state's constraint *B'* and the negation of B is unsatisfiable: that is, whenever a state is reachable in which the negation of the invariant B holds.

Recall the plan *AssignWithConflict* in Figure 2, which has a potential race condition for the local memory x. Let *init* be a configuration of objects representing an initial configuration for *AssignWithConflict* in which all nodes in the plan are in state *inactive*. Consider the following safety verification requirements, where symbol $\Box$ represents the 'always' modal operator in LTL:

$$\mathcal{T}_{\mathcal{R}}, (init\,; true) \models \Box race\text{-}free(x.AssignWithConflict), \tag{6}$$
$$\mathcal{T}_{\mathcal{R}}, (init\,; S \geq 1) \models \Box race\text{-}free(x.AssignWithConflict), \tag{7}$$
$$\mathcal{T}_{\mathcal{R}}, (init\,; S \geq 1) \models \Box inv(AssignWithConflict). \tag{8}$$

In this verification requirements, variable S ranges over the built-in sort *Int* and represents the variable S under the control of the environment in the plan *AssignWithConflict*, in Figure 2. Property (6) asserts that all reachable states from *init* are free from race conditions on memory x whenever S has no initial constraints. Property (7) asserts that all reachable states from *init* are free from race conditions on memory x whenever S is assumed to be at least 1. Property (8) asserts that the invariant condition of node *AssignWithConflict* holds in all reachable states from *init* whenever S is assumed to be at least 1. Note that these properties are symbolic reachability requirements because of the nature of the external variable S. Also, the constrained terms defining the initial states in these properties represent, in each case, infinitely many initial states.

By using Maude's LTL Model Checker, Property (6) can be disproved, and properties (7) and (8) can be proved automatically.

```
reduce in ASSIGNWITHCONFLICT :
  verify-lite( ( init ; true ), [] race-free(x . AssignWithConflict)) .
rewrites: 2590 in 525ms cpu (1629ms real) (4929 rewrites/second)
result Bool: false

reduce in ASSIGNWITHCONFLICT :
  verify-lite( ( init ; S >= 1 ), [] race-free(x . AssignWithConflict)) .
rewrites: 2846 in 575ms cpu (614ms real) (4947 rewrites/second)
result Bool: true
```

```
reduce in ASSIGNWITHCONFLICT :
  verify-lite( ( init ; S >= 1 ), [] inv(AssignWithConflict) .
rewrites: 3191 in 576ms cpu (702ms real) (5534 rewrites/second)
result Bool: true
```

The function *verify-lite* is a wrapper to Maude's LTL Model Checker function *modelCheck*. This mapping outputs either *true* or *false*, depending on the output of the model checker function, ignoring the details of a counterexample, if any (see [55] for more details).

9 Related Work and Concluding Remarks

The idea of combining term rewriting/narrowing techniques and constrained data structures is an active area of research, especially since the advent of modern theorem provers and model checkers with highly efficient decision procedures in the form of SMT solvers. The overall aim of these techniques is to advance applicability of methods in symbolic verification where the constraints are expressed in some logic that has an efficient decision procedure. In particular, the work presented here has strong similarities with the narrowing-based symbolic analysis of rewrite theories initiated in [48] and extended in [8]. There are also some similarities with symbolic reachability analysis based on (tree) automata (see, e.g., [32] and references there). In comparison with the tree automata methods, a considerably richer set of infinite states and properties of such states can be expressed. This is because sets of states —and therefore properties to be verified— which are describable by regular tree languages roughly correspond in our setting to symbolic states of the form $u; \top$, where u is a *linear* term. Instead, in rewriting modulo SMT richer sets of states (and properties) can be specified by pairs $u; \varphi$, where u is an arbitrary, not necessarily linear, term, φ a decidable built-in formula, and the ground instances of u are understood *modulo* the equations E. Then, rewriting modulo axioms B_1 is combined with SMT solving to explore reachable sets of states. This greater expressiveness makes unnecessary the use of *over-approximations* of sets of states by regular languages needed in tree automata methods. The main difference in comparison with narrowing-based methods is the replacement of narrowing modulo axioms by rewriting modulo axioms and SMT solving, the decidability advantages of SMT for constraint solving, and the greater efficiency of matching modulo axioms B over unification modulo B.

Besides the just-mentioned narrowing-based model checking of infinite-state systems, the present work has also important similarities with SMT-based model checking approaches such as, for example, those by A. Podelski [54], G. Delzano and A. Podelski [21], T. Rybina and A. Voronkov [60], the work by S. Ghilardi, S. Ranise, and various other researchers around the MTMC SMT-based model checker [34, 35] and, more recently, the IC3- and SMT-based model checking techniques [18], and the constrained Horn-clause-based approach for model checking timed systems by Hojjat et al. in [37]. In comparison with that body of work, what is indeed common is the use of SMT solving to handle symbolically infinite sets of states, but there are some notable differences having to do with both the structure that is possible for states, and support for open systems. Specifically, in [21, 54, 60] the state must always be an *n-tuple* of

data and control elements, with associated state variables $\overrightarrow{x} = x_1, \ldots, x_n$, and state changes are specified by guarded (simultaneous) assignment commands of the form $\phi \Rightarrow \overrightarrow{x}' := \overrightarrow{t}$, where $\overrightarrow{t} = t_1, \ldots, t_n$ is a sequence of Σ-terms with variables in $\overrightarrow{x}$, ϕ is a Σ-formula, and (Σ, T) is a decidable theory. Such guarded assignments are just conditional rewrite rules of the form $\langle x_1, \ldots, x_n \rangle \rightarrow \langle t_1, \ldots, t_n \rangle$ **if** ϕ. Such rules can have internal non-determinism, depending on which rule is chosen, but the systems so specified are *closed*, i.e., they do not have any external non-determinism. Furthermore, the state structure must necessarily be a tuple. The work in [34, 35] allows greater flexibility in this regard: the state structure is also fixed, namely, it must be an *array*, but this makes it easy to specify parametric systems. Also, rather than assuming a specific format for state transitions, such as that of guarded assignments, transitions can be defined by Σ-formulas with (Σ, T) a decidable theory. This allows for a possibility of transitions that, when viewed as conditional rewrite rules, can have extra variables in their right-hand sides and can model an open system. In a similar way, the work in [37] assumes a fixed state structure consisting of an array of processes and a shared global state. By contrast, in the approach presented in this paper the state structure is completely general and user-definable and can obey structural axioms also specified by the user; and support for openness is a key part of the semantic framework.

In spite of the above-mentioned differences, there is great commonality in the type of advanced techniques used in SMT-based model checking to speed up and often attain convergence of the reachability analysis process, since all of them —including state subsumption, backwards reachability, k-induction, interpolants, and the combination of IC3 with SMT— can also be applied to rewriting modulo SMT. The current reflection-based prototype described in Section 6 does not yet support any of these techniques, but they should certainly be added to a future implementation.

Finally, SMT-based reachability analysis has been used in software testing in tools such as KLEE [16] for symbolic execution and constraint solving, finding possible inputs that will cause a program to crash and outputting these as test cases, and SMT-CBMC [3] and Corral [42] for bounded model checking where unbounded types are represented by built-in variables and the syntax of expressions is restricted so that it can be efficiently decided by SMT solving. See [17] for a comprehensive account of symbolic techniques for reachability analysis in software testing, including SMT-based ones.

The work by C. Kirchner, H. Kirchner, and M. Rusinowitch on deduction with symbolic constraints [39] is a pioneering work where the notions of constraints, rewrite rule with symbolic constraints, simplification with these rules, and applications to equational superposition theorem proving based on such notions were proposed. These ideas have had an important influence in several areas, such as, for example, subsequent work on superposition theorem proving with constraints, see, e.g., [31]; and in the constrained rewriting approach by H. Kirchner and C. Ringeissen to the combination of symbolic constraint solvers [38]. In a similar vein, M. Ayala-Rincón [5] investigated, in the setting of many-sorted equational logic, the expressiveness of conditional equational systems whose conditions may use built-in predicates. This class of equational theories is important because the combination of equational and built-in premises yield a type of clauses which is more expressive than purely conditional equations.

Rewriting notions like sufficient completeness, confluence, termination, and critical pairs have also been investigated for rewriting modulo built-ins. There is the work of A. Bouhoula and F. Jacquemard [13], who studied the problem of sufficient completeness for conditional and constrained term rewrite systems, and propose a solution based on tree grammars and narrowing. S. Falke and D. Kapur [28] studied the problem of termination of rewriting with constrained built-ins. In particular, they extended the dependency pair framework to handle termination of equational specifications with semantic data structures and evaluation strategies in the Maude functional sublanguage. The same authors used the idea of combining rewriting induction and linear arithmetic over constrained terms [29]. Their aim is to obtain equational decision procedures that can handle semantic data types represented by the constrained built-ins. The main difference between their work and rewriting modulo SMT presented in this paper is that the notion of symbolic rewriting modulo decidable constraints is completely different. According to Definition 14 in [29], a symbolic rewrite step $u;\phi \rightsquigarrow_{\mathcal{R}} v;\phi$ with a rule $l \rightarrow r$ **if** φ in *their* sense requires a matching substitution θ such that $\phi \Rightarrow (\varphi\theta)$ is $\mathcal{T}_{\mathcal{E}_0}$*-valid*. This is a *universal* notion of symbolic rewriting with constraints completely different from our *existential* notion in Definition 7, which is based instead on constrain *satisfiability*. This difference is understandable by observing that the goal in [29] is to prove universal formulas about equational specifications by inductive theorem proving, whereas our goal is very different, namely, to prove existential reachability formulas about a concurrent system specified by a rewrite theory. More recently, C. Kop and N. Nishida [40] have proposed a way to unify the ideas regarding equational rewriting with logical constraints and have proposed in [41] an inductive method of proving properties of programs in an imperative language by their notion of symbolic rewriting modulo decidable constraints. The main difference with our approach is that, as in [29], their notion of symbolic rewriting is *universal*, and therefore completely different from our existential notion in Definition 7; furthermore, in [41] termination of the rewrite theory is required for inductive reasoning, whereas no termination is required at all in our setting. Again, all this is understandable given their focus on inductive theorem proving of universal formulas. One similarity between the work in [41] and our work is that, to handle input-output in an imperative language, they allow, as we do, extra variables in the righthand sides of rewrite rules. In general, while approaches such as in [5,12,27–29,38–40] address symbolic reasoning for *equational* theorem proving purposes, or apply these techniques to *imperative program analysis and verification*, even allowing sometimes extra variables in the right-hand sides of equations, e.g., [41,63,64], theses approaches are quite different from ours because of their predominant focus on equational reasoning for proving, often inductively, universal formulas, and/or on applications to, typically sequential, programming languages.

Last but not least, recently, A. Arusoaie et al. [4] have proposed a language-independent symbolic execution framework, within the *K* framework [44], for languages endowed with a formal operational semantics based on term rewriting. There, the built-in subtheories are the datatypes of a programming language and symbolic analysis is performed on constrained terms (called *patterns*); unification is also implemented by matching for a restricted class of rewrite rules and uses SMT solvers to check constraints. This work is also related to our approach. A more detailed comparison of how

both approaches are applied to analyzing conventional programs based on their rewriting semantics is an interesting task for future research.

This paper has presented rewrite theories modulo built-ins and has shown how they can be used for *symbolically* modeling and analyzing concurrent open systems, where nondeterministic values from the environment can be represented by built-in terms [55, 57]. In particular, the main contributions of this paper can be summarized as follows: (1) it presents rewriting modulo SMT as a new symbolic technique combining the powers of rewriting, SMT solving, and model checking; (2) this combined power can be applied to model and analyze systems outside the scope of each individual technique; (3) in particular, it is ideally suited to model and analyze the challenging case of *open systems*; and (4) because of its reflective reduction to standard rewriting, current algorithms and tools for model checking closed systems can be *reused* in this new symbolic setting without requiring any changes to their implementation.

Under reasonable assumptions, including decidability of $\mathcal{E}_0^+$, a rewrite theory modulo is executable by term rewriting modulo SMT. This feature makes it possible to use, for symbolic analysis, state-of-the-art tools already available for Maude, such as its space search commands, with no change whatsoever required to use such tools. In this paper, it has been proved that the symbolic rewrite relation is sound and complete with respect to its ground counterpart. Furthermore, the paper has presented an overview of the prototype that offers support for rewriting modulo SMT in Maude and two case studies. These case studies regard the symbolic analysis of the CASH scheduling algorithm and the PLEXIL synchronous language illustrating the use of these techniques.

Future work on a mature implementation on extending the idea of rewriting modulo SMT with other symbolic constraint solving techniques such as narrowing modulo should be pursued. Furthermore, the generalization of rewrite theories modulo a built-in subtheory with equations for the non built-ins should also be investigated. Finally, the extension to other symbolic LTL model checking properties, together with state space reduction techniques, should be considered, taking into account the rich experience already available on model checking of temporal logic properties in SMT-based model checkers, e.g., [21, 33, 34, 54]. Further applications to Real-Time Maude, PLEXIL, and other languages should also be pursued.

Acknowledgments The authors would like to thank S. Eker for fruitful discussions on these ideas and their implementation in Maude, and the anonymous referees for their very helpful comments that helped us improve the paper. This work was partially supported by NSF Grant CNS 13-19109. The first author would like to thank the National Institute of Aerospace for a short visit supported by the Assurance of Flight Critical System's project of NASA's Aviation Safety Program at Langley Research Center under Research Cooperative Agreement No. NNL09AA00A.

References

1. E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic SUP(LA). In S. Ghilardi and R. Sebastiani, editors, *7th International Symposium on Frontiers of Combining Systems*, volume 5749 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2009.

2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal of Software Tools for Technology Transfer*, 11(1):69–83, 2009.
4. A. Arusoaie, D. Lucanu, and V. Rusu. A generic framework for symbolic execution. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *6th International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 281–301. Springer, 2013.
5. M. Ayala-Rincón. *Expressiveness of Conditional Equational Systems with Built-in Predicates*. PhD thesis, Universität Kaiserslauten, 1993.
6. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
7. F. Baader and K. Schulz. Unification in the union of disjoint equational theories: combining decision procedures. *Journal of Symbolic Computation*, 21:211–243, 1996.
8. K. Bae, S. Escobar, and J. Meseguer. Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 81–96. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.
9. K. Bae and C. Rocha. A note on symbolic reachability analysis modulo integer constraints for the CASH algorithm. Available at: `http://maude.cs.uiuc.edu/cases/scash`, 2012.
10. M. P. Bonacina, C. Lynch, and L. M. de Moura. On deciding satisfiability by theorem proving with speculative inferences. *Journal of Automated Reasoning*, 47(2):161–189, 2011.
11. A. Boudet. Combining unification algorithms. *Journal of Symbolic Computation*, 16(6):597–626, 1993.
12. A. Bouhoula and F. Jacquemard. Automated induction with constrained tree automata. In A. Armando, P. Baumgartner, and G. Dowek, editors, *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 539–554. Springer Berlin Heidelberg, 2008.
13. A. Bouhoula and F. Jacquemard. Sufficient completeness verification for conditional and constrained TRS. *Journal of Applied Logic*, 10(1):127 – 143, 2012. Special issue on Automated Specification and Verification of Web Systems.
14. R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.
15. M. Caccamo, G. C. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *IEEE 34th Real-Time Systems Symposium*, pages 295–304. IEEE Computer Society, 2000.
16. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.
17. C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, Feb. 2013.
18. A. Cimatti and A. Griggio. Software model checking via IC3. In P. Madhusudan and S. A. Seshia, editors, *24th International Conference on Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.
19. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
20. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

21. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
22. G. Dowek, C. Muñoz, and C. Păsăreanu. A formal analysis framework for PLEXIL. In F. Ingrand and K. Rajan, editors, *3rd Workshop on Planning and Plan Execution for Real-World Systems*, pages 45–51, September 2007.
23. G. Dowek, C. Muñoz, and C. Păsăreanu. A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA, 2008.
24. G. Dowek, C. A. Muñoz, and C. Rocha. Rewriting logic semantics of a plan execution language. In B. Klin and P. Sobocinski, editors, *6th Workshop on Structural Operational Semantics*, volume 18 of *Electronic Proceedings in Theoretical Computer Science*, pages 77–91, 2009.
25. F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain. Proving Operational Termination of Membership Equational Programs. *Higher Order Symbolic Computation*, 21(1-2):59–88, 2008.
26. T. Estlin, A. Jónsson, C. Păsăreanu, R. Simmons, K. Tso, and V. Verma. Plan Execution Interchange Language (PLEXIL). Technical Memorandum TM-2006-213483, NASA, 2006.
27. S. Falke and D. Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In A. Voronkov, editor, *19th International Conference on Rewriting Techniques and Applications*, volume 5117 of *Lecture Notes in Computer Science*, pages 94–109. Springer Berlin Heidelberg, 2008.
28. S. Falke and D. Kapur. Operational termination of conditional rewriting with built-in numbers and semantic data structures. *Electronic Notes in Theoretical Computer Science*, 237:75–90, 2009.
29. S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In B. Gramlich, D. Miller, and U. Sattler, editors, *6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 2012.
30. M. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 794–801, Nov 2006.
31. H. Ganzinger and R. Nieuwenhuis. Constraints and theorem proving. In *Constraints in Computational Logics: Theory and Applications, International Summer School*, volume 2002 of *Lecture Notes in Computer Science*, pages 159–201. Springer, 1999.
32. T. Genet, T. Le Gall, A. Legay, and V. Murat. A completion algorithm for lattice tree automata. In S. Konstantinidis, editor, *18th International Conference on Implementation and Application of Automata*, volume 7982 of *Lecture Notes in Computer Science*, pages 134–145. Springer, 2013.
33. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Combination methods for satisfiability and model-checking of infinite-state systems. In *21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 362–378. Springer, 2007.
34. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model checking of array-based systems. In *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.
35. S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *5th International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 22–29. Springer, 2010.
36. J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

37. H. Hojjat, P. Rümmer, P. Subotic, and W. Yi. Horn clauses for communicating timed systems. In N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors, *1st Workshop on Horn Clauses for Verification and Synthesis*, volume 169 of *Electronic Proceedings in Theoretical Computer Science*, pages 39–52, 2014.
38. H. Kirchner and C. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.
39. K. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'intelligence artificielle*, 4(3):9–52, 1990.
40. C. Kop and N. Nishida. Term rewriting with logical constraints. In *9th International Symposium on Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2013.
41. C. Kop and N. Nishida. Automatic constrained rewriting induction towards verifying procedural programs. In J. Garrigue, editor, *12th Asian Symposium on Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 334–353. Springer, 2014.
42. A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. Technical Report MSR-TR-2012-9, Microsoft Research, January 2012.
43. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
44. D. Lucanu, T.-F. Serbanuta, and G. Rosu. K framework distilled. In *9th International Workshop on Rewriting Techniques and Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012.
45. S. Lucas and J. Meseguer. Operational termination of membership equational programs: the order-sorted way. *Electronic Notes in Theoretical Computer Science*, 238(3):207–225, 2009.
46. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
47. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *12th International Workshop on Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.
48. J. Meseguer and P. 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.
49. A. Milicevic and H. Kugler. Model checking using SMT and theory of lists. In *NASA 3rd International Symposium on Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2011.
50. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
51. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(t). *Journal of the ACM*, 53(6):937–977, 2006.
52. P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *9th International Conference on Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.
53. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
54. A. Podelski. Model checking as constraint solving. In *7th International Symposium on Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2000.

55. C. Rocha. *Symbolic Reachability Analysis for Rewrite Theories*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.
56. C. Rocha, H. Cadavid, C. A. Muñoz, and R. Siminiceanu. A formal interactive verification environment for the Plan Execution Interchange Language. In J. Derrick, S. Gnesi, D. Latella, and H. Treharne, editors, *9th International Conference on Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2012.
57. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT. Technical Memorandum NASA/TM-2013-218033, NASA, Langley Research Center, Hampton VA 23681-2199, USA, August 2013.
58. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. In S. Escobar, editor, *10th International Workshop on Rewriting Logic and Its Applications*, volume 8663 of *Lecture Notes in Computer Science*, pages 247–262. Springer International Publishing, 2014.
59. G. Roşu and A. Ştefănescu. Matching logic: a new program verification approach. In *33rd International Conference on Software Engineering*, pages 868–871, New York, NY, USA, 2011. ACM.
60. T. Rybina and A. Voronkov. A logical reconstruction of reachability. In *5th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 2003.
61. T. Sakata, N. Nishida, and T. Sakabe. On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In H. Kuchen, editor, *WFLP*, volume 6816 of *Lecture Notes in Computer Science*, pages 138–155. Springer, 2011.
62. M. Veanes, N. Bjørner, and A. Raschke. An SMT approach to bounded reachability analysis of model programs. In *28th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, pages 53–68. Springer, 2008.
63. G. Vidal. Closed symbolic execution for verifying program termination. In *IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 34–43, Sept 2012.
64. G. Vidal. Symbolic execution as a basis for termination analysis. *Science of Computer Programming*, 102:142 – 157, 2015.
65. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.
66. D. Walter, S. Little, and C. Myers. Bounded model checking of analog and mixed-signal circuits using an SMT solver. In K. S. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, editors, *5th International Symposium on Automated Technology for Verification and Analysis*, pages 66–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
67. S. Yovine. KRONOS: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1):123–133, 1997.