Synchronous set relations in rewriting logic

Camilo Rocha a,∗, César Muñoz b

a Escuela Colombiana de Ingeniería, Bogotá, D.C., Colombia
b NASA Langley Research Center, Hampton, VA, USA

A R T I C L E  I N F O

Article history:
Received 10 April 2012
Received in revised form 3 July 2013
Accepted 17 July 2013
Available online xxxx

Keywords:
Synchronous set relations
Synchronous semantics
Rewriting logic
Formal simulation and verification
PLEXIL

A B S T R A C T

This paper presents a mathematical foundation and a rewriting logic infrastructure for the execution and property verification of synchronous set relations. The mathematical foundation is given in the language of abstract set relations. The infrastructure, which is written in the Maude system, enables the synchronous execution of a set relation provided by the user. By using the infrastructure, algorithm verification techniques such as reachability analysis and model checking, already available in Maude for traditional asynchronous rewriting, are automatically available to synchronous set rewriting. In this way, set-based synchronous languages and systems such as those built from agents, components, or objects can be naturally specified and simulated, and are also amenable to formal verification in the Maude system. The use of the infrastructure and some of its Maude-based verification capabilities are illustrated with an executable operational semantics of the Plan Execution Interchange Language (PLEXIL), a synchronous language developed by NASA to support autonomous spacecraft operations.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Synchronous languages are extensively used in embedded and critical applications and synchronous set relations provide a natural model for describing their operational semantics. In a previous work, Rocha et al. [16] have proposed a serialization procedure for simulating the execution of synchronous set-based relations by asynchronous term rewriting. The synchronous execution of a set relation is a parallel reduction, where the terms to be reduced in parallel are selected according to some strategy. Such a serialization procedure was manually implemented to provide the rewriting logic semantics of the Plan Execution Interchange Language (PLEXIL) [7], a synchronous plan execution language developed by NASA to support spacecraft automation [8]. Defining the synchronous semantic relation of a programming language, such as PLEXIL, is generally difficult, time consuming, and error-prone. Moreover, manually implementing the generic serialization procedure proposed in [16] adds complexity to an already challenging task. This paper presents new contributions in the direction of supporting more general synchronous set relations and a computer infrastructure that automatically implements a serialization procedure for a general class of synchronous set relations.

In particular, the work presented in this paper extends the development in [16] in two significant ways. First, it generalizes the theoretical foundations of synchronous set relations by extending the notion of strategy to include a larger set of synchronous transformations such as those that enable non-deterministic ways to select elements in a set to be synchronously reduced. Strategies, as defined in [16], only enable deterministic ways to select elements in a set to be synchronously reduced. Deterministic strategies are appropriate in the context of PLEXIL’s operational semantics is deterministic. However, they may be too restrictive for encoding the semantics of non-deterministic synchronous languages.

© 2013 Elsevier B.V. All rights reserved.
Moreover, non-deterministic strategies are fundamental to the development of new techniques for symbolic reachability analysis of synchronous languages [13]. Even in the case of deterministic languages, such as PLEXIL, non-determinism is introduced by the inherent nature of symbolic variables.

The second major contribution with respect to [16] is a computer infrastructure that implements, on-the-fly, a serialization procedure for a synchronous language provided by the user. This infrastructure is written in Maude [4], a high-performance rewriting logic engine. The infrastructure provides syntactic constructs for defining synchronous relations and their, possibly non-deterministic, reduction strategies. Using Maude’s reflective capabilities, these constructs are translated into asynchronous rewriting systems that soundly and completely simulate the synchronous relations provided as inputs.

These two contributions allow for simpler and more succinct language specifications, and more general synchronous set relations. Also, since programming languages tend to evolve constantly, this infrastructure can help language designers to focus exclusively on the synchronous semantic design without shifting their attention to details in the serialization procedure implementation.

The rest of this paper is organized as follows. Section 2 presents, in an abstract setting, definitions that are relevant to synchronous set relations. These definitions properly extend those given in [16]. Formally, a synchronous set relation is defined as the synchronous closure of an atomic relation with a given strategy. Two sets are synchronously related for a particular strategy if the first set can be transformed into the second set by parallel atomic transformations according to the strategy. The strategy selects sets of redexes that can be synchronously transformed.

Section 3 describes Maude’s rewriting logic, which provides the formal framework for the development presented in this paper. Maude is a reflective language and rewriting logic system that supports both equational and rewrite theories. Maude implements set rewriting, i.e., rewriting modulo axioms such as associativity, commutativity, and identity. These capabilities are well-suited for set-based and multiset-based concurrent systems.

Section 4 describes the proposed infrastructure, which comprises generic sorts and terms, algebraic properties of generic datatypes, and functions that support the on-the-fly implementation of a serialization procedure. The algebraic datatypes in this specification can be instantiated with the specific constructs of a set-based synchronous language provided by the user. These constructs are translated into labeled conditional rewriting rules that, under some conditions, becomes a formal interpreter for the synchronous semantic relation of the language.

Section 5 describes how the proposed infrastructure can be used for defining the executable semantics of a simple synchronous language with arithmetic expressions. Code snippets of the Maude specification are used to illustrate, in a concrete way, how the infrastructure is used.

As a more involved example, Section 6 presents a rewriting logic semantics of PLEXIL implemented on top of the infrastructure developed in Maude. This semantics comprises a significant subset of the language that includes Boolean and arithmetic expressions, and all language specific predicates. As a direct advantage of using the infrastructure, all commands in Maude for rewrite specifications such as its rewrite and search commands, and formal verification tools such as Maude’s LTL Model Checker, are available for analyzing properties of programs in PLEXIL.

The infrastructure in Maude and the examples presented in this paper are available from http://shemesh.larc.nasa.gov/people/cam/PLEXIL.

2. Abstract synchronous set relations

This section introduces the concepts of abstract set relations used in this paper.

Let \( U \) be a set whose elements are denoted \( A, B, \ldots \) and let \( \to \) be a binary relation on \( U \). An element \( A \in U \) is called a \( \to \)-redex if there exists \( B \in U \) such that the pair \( (A, B) \in \to \). The expressions \( A \to B \) and \( A \not\to B \) denote, respectively, \( (A; B) \in \to \) and \( (A; B) \not\in \to \). The identity relation and reflexive-transitive closure of \( \to \) are defined as usual and denoted \( \to^0 \) and \( \to^* \), respectively.

Henceforth, it is assumed that \( U \) is the family of all nonempty finite sets over an abstract and possibly infinite set \( T \), i.e., \( U \subseteq 2^T \), \( 0 \not\in U \), and if \( A \in U \) then \( \text{card}(A) \in \mathbb{N} \) (therefore, \( \to \) is a binary relation on finite sets of \( T \)). The elements of \( U \) are called terms in this section. The elements of \( T \) will be denoted by lowercase letters \( a, b, \ldots \). When it is clear from the context, curly brackets are omitted from set notation, e.g., \( \{a, b\} \to \{a\} \). Because of this abuse of notation, the symbol ‘\( \vdash \)’ is overloaded to denote set union, e.g., if \( A \) denotes the set \( \{a, b\} \), \( B \) denotes the set \( \{c, d\} \), and \( D \) denotes the set \( \{d, e\} \), then notation \( A, B \to B, D \) denotes \( \{a, b, c, d, e\} \to \{c, d, e\} \).

The parallel relation \( \parallel \to \) of \( \to \) is the relation defined as the parallel closure of \( \to \), i.e., the set of pairs \( (A; B) \) in \( U \times U \) such that \( A \to B \) if and only if there exist \( A_1, \ldots, A_n \), (nonempty) pairwise disjoint subsets of \( A \), and sets \( B_1, \ldots, B_n \) such that \( A_i \to B_i \) and \( B = (A \setminus \bigcup_{1 \leq i \leq n} A_i) \cup \bigcup_{1 \leq i \leq n} B_i \).

This paper focuses on synchronous set relations. The synchronous relation of an abstract set relation \( \to \) is defined as a subset of the parallel closure of \( \to \), where a given strategy selects elements from \( \to \). Formally, a \( \to \)-strategy is a function \( s \) that maps an element \( A \in U \) into a set \( s(A) \subseteq 2^T \) such that if \( \{(A_1; B_1), \ldots, (A_n; B_n)\} \in s(A) \), then \( A_i \subseteq A \) and \( A_i \to B_i \), for \( 1 \leq i \leq n \), and \( A_1, \ldots, A_n \) are pairwise disjoint.

Definition 1 (Synchronous relation). Let \( s \) be a \( \to \)-strategy. The relation \( \to^s \) denotes the set of pairs \( (A; B) \) in \( U \times U \) such that \( A \to^s B \) if and only if \( B = (A \setminus \bigcup_{1 \leq i \leq n} A_i) \cup \bigcup_{1 \leq i \leq n} B_i \), where \( \{(A_1; B_1), \ldots, (A_n; B_n)\} \in s(A) \).
Example 1. Let \( T \) be the set of distinct elements \( a, b, c, d, e \), and the relation \( \rightarrow = \{ r_1, r_2, r_3 \} \), where \( r_1 = \{a; b; d\} \), \( r_2 = \{c; d\} \), and \( r_3 = \{a; c; e\} \). Let \( s_1, s_2, \) and \( s_3 \) be \( \rightarrow \)-strategies defined for \( A = \{a, b, c, d\} \) as follows:

\[
\begin{align*}
\ s_1(A) &= \{r_2\}, \{r_3\}, \\
\ s_2(A) &= \{r_1, r_2\}, \\
\ s_3(A) &= \{r_1, r_2\}, \{r_3\}.
\end{align*}
\]

It holds that

\[
\begin{align*}
\ a, b, c, d &\rightarrow s_1 a, b, d, \\
\ a, b, c, d &\rightarrow s_1 b, d, e, \\
\ a, b, c, d &\rightarrow s_2 b, d, \\
\ a, b, c, d &\rightarrow s_2 b, d, e.
\end{align*}
\]

A simple mechanism for defining strategies is through priorities. A priority \( \prec \) for a relation \( \rightarrow \) is a \( \mathcal{U} \)-indexed set \( \prec = \{\prec A\}_{A \in \mathcal{U}} \), where each \( \prec A \) is a strict partial order on \( \rightarrow \cap (\wp(A) \times \mathcal{U}) \). Priorities can be used to decide between overlapping redexes.

Definition 2 (Saturation). A set \( (A_1; B_1), \ldots, (A_n; B_n) \subseteq \rightarrow \) is \( \prec\)-saturated for \( A \in \mathcal{U} \) (or \( \prec A \)-saturated), with \( \prec \) a priority for \( \rightarrow \), if and only if

1. The sets \( A_1, \ldots, A_n \) are nonempty pairwise disjoint subsets of \( A \),
2. Each \( (A_1; B_1) \) is such that for any \( A' \rightarrow B' \) with \( A' \subseteq A \) and \( A' \cap A_1 \neq \emptyset \), \( (A_1; B_1) \not\prec A' \),
3. If there is \( A' \rightarrow B' \) with \( (A'; B') \notin \{(A_1; B_1), \ldots, (A_n; B_n)\} \) and \( A' \subseteq A \), then either
   (i) there is an \( (A_j; B_j) \), for some \( 1 \leq j \leq n \), such that \( A_j \cap A' \neq \emptyset \) or
   (ii) there is an \( A'' \rightarrow B'' \) with \( A'' \subseteq A \), \( A'' \cap A' \neq \emptyset \), and \( (A'; B') \not\prec A'' \).

A \( \prec A \)-saturated set is a complete collection of non-overlapping redexes in a term \( A \in \mathcal{U} \) that are maximal with respect to \( \prec \). More precisely, in Definition 2, the condition (1) states that a \( \prec\)-saturated set does not contain overlapping redexes, condition (2) states that a \( \prec\)-saturated set can only contain pairs \( (A_1; B_1) \) that are \( \prec\)-maximal, and condition (3) states that any possible extension of a saturated set with a pair \( (A'; B') \) would violate the first or the second conditions. Note that the \( \prec\)-maximality tests in conditions (2) and (3) of Definition 2 are given with respect to all \( A' \rightarrow B' \) in \( \prec A \), and hence \( \prec A \)-saturation exclusively depends on the ordering of the finitely many subsets of \( \rightarrow \cap (\wp(A) \times \mathcal{U}) \).

Example 2. Recall the relation \( \rightarrow = \{r_1, r_2, r_3\} \) and the set \( A = \{a, b, c, d\} \) from Example 1. Let \( \prec A \) be such that \( r_1 \not\prec A r_3 \). It holds that the sets \( \{r_2\} \) and \( \{r_3\} \) are \( \prec A \)-saturated. The set \( \{r_1, r_2\} \) is not \( \prec A \)-saturated because \( r_1 \) falsifies condition (2) in Definition 2 with witness \( r_3 \). Let \( \prec A \) be such that \( r_3 \not\prec A r_1 \). In this case, the only \( \prec A \)-saturated set is \( \{r_1, r_2\} \). The set \( \{r_3\} \) is not \( \prec A \)-saturated because \( r_3 \) falsifies condition (2) in Definition 2 with witness \( r_1 \). For \( \prec A = \emptyset \), the sets \( \{r_1, r_2\} \) and \( \{r_3\} \) are the only \( \prec A \)-saturated sets.

A maximal strategy defines the most general synchronous behavior of a relation, which is given by all \( \prec\)-saturated sets.

Definition 3 (Maximal strategies). Let \( \prec \) be a priority for \( \rightarrow \). A \( \rightarrow \)-strategy \( s \) is \( \prec\)-maximal for \( A \in \mathcal{U} \) (or \( \prec A \)-maximal) if and only if \( s(A) \) is the collection of all \( \prec A \)-saturated sets. A \( \rightarrow \)-strategy is \( \prec\)-maximal if and only if it is \( \prec A \)-maximal for all \( A \in \mathcal{U} \).

Example 3. From Examples 1 and 2, \( \rightarrow \)-strategies \( s_1, s_2, \) and \( s_3 \) are, respectively, \( \prec A \)-maximal, \( \prec A \)-maximal, and \( \prec A \)-maximal.

Algorithm 1 witnesses the existence of maximal strategies, which are unique for a given relation \( \rightarrow \) and a priority \( \prec \) (for \( \rightarrow \)).

Theorem 1. Let \( \prec \) be a priority for \( \rightarrow \). Then a \( \prec\)-maximal \( \rightarrow \)-strategy exists. Therefore, from Definition 3, the \( \prec\)-maximal \( \rightarrow \)-strategy is unique.

Proof. It is proved that the existence of a \( \prec\)-maximal \( \rightarrow \)-strategy is witnessed by Algorithm 1, for any \( A \in \mathcal{U} \) and priority \( \prec \) for \( \rightarrow \). First, the following are important and easy to prove remarks about Algorithm 1:

- All three loops (lines 3, 6, and 12) repeat finitely many times and all quantified conditions (lines 7 and 14) require finitely many comparisons because \( A \in \mathcal{U} \) has finitely many elements; also the complexity of \( \gamma \) decreases with each iteration of the third loop, i.e., Algorithm 1 terminates;
- \( \alpha = \rightarrow \cap (\wp(A) \times \mathcal{U}) \) is finite and can be computed effectively;
- \( \beta = \alpha \setminus \{ (A'; B') \sqsubseteq \alpha | (\exists (A''; B'') \in \alpha) A'' \cap A' \neq \emptyset \land (A'; B')<\prec A (A''; B'') \} \), i.e., \( \beta \) is the subset of \( \alpha \) in which all conflicting pairs in \( \alpha \) that are not maximal elements in \( \prec A \) have been omitted;
\[ \sigma \subseteq \varphi(\beta) \] is the collection of largest non-conflicting subsets of \( \beta \); and

- If \( C \in \mathcal{C} \), then for any nonempty \( C' \subseteq (\beta \setminus C) \), \( C \cup C' \not\subseteq \sigma \).

Let \( D = \{ (A_1; B_1), \ldots, (A_n; B_n) \} \). It is enough to prove, for \( A \in \mathcal{U} \) and priority \( \prec \) for \( \rightarrow \), that \( D \) is \( \prec_A \)-saturated if and only if \( D \in \sigma \).

(\( \Rightarrow \)) If \( D \) is \( \prec_A \)-saturated, then \( D \subseteq \alpha \) follows by definition. If \( D \not\subseteq \beta \), then there is \( (A_1; B_1) \in D \) satisfying \( (A_1; B_1) \prec_A (A'; B') \) for some \( (A'; B') \in \alpha \) with \( A' \cap A_1 \not= \emptyset \). But then, for \( D \), \( (A_1; B_1) \) violates condition (2) in Definition 2, a contradiction. Hence, \( D \subseteq \beta \). If \( D \not\subseteq \sigma \), since \( D \subseteq \beta \) and the \( A_1, \ldots, A_n \) are pairwise disjoint by assumption, either there is a nonempty set \( D' \subseteq \beta \setminus D \) such that \( D \cup D' \not\subseteq \sigma \) or there is a nonempty set \( D'' \subseteq \beta \) such that \( D' \cup D'' \not\subseteq \sigma \). If \( D \cup D' \not\subseteq \sigma \) and \( D' \) is nonempty, any pair \( (A'; B') \in D' \) violates condition (3.ii) in Definition 2, contradicting the \( \prec_A \)-maximality of \( D \). If \( D'' \not\subseteq \sigma \) and \( D'' \subseteq \sigma \), then for any pair \( (A'; B') \in D'' \), the set \( C = D'' \cup \{ (A'; B') \} \) falsifies the test in line 14 of Algorithm 1 and hence \( C \in \sigma \). Since \( D'' \subseteq \sigma \) and \( D'' \subseteq \sigma \), this contradicts the last remark aforementioned. Therefore, as desired, \( D \in \sigma \).

(\( \Leftarrow \)) If \( D \in \sigma \subseteq \varphi(\alpha) \), then \( A_1, \ldots, A_n \) are pairwise disjoint \( \rightarrow \) redexes, i.e., subsets, of \( A \). Thus, condition (1) in Definition 2 is satisfied. For condition (2), since \( D \in \sigma \), it follows that \( D \subseteq \beta \). Hence, any \( (A_1; B_1) \in D \) satisfies condition (2) in Definition 2. For condition (3), assume that there is \( (A'; B') \in \alpha \) with \( (A'; B') \notin \beta \). Then, either \( (A'; B') \in (\beta \setminus D) \) or \( (A'; B') \in (\alpha \setminus \beta) \). If \( (A'; B') \in (\beta \setminus D) \), then \( D \cup \{ (A'; B') \} \not\subseteq \sigma \), as previously stated. However, \( (A'; B') \in \beta \), so it must be the case that \( A' \cap A_i \not= \emptyset \) for some \( 1 \leq i \leq n \). If \( (A'; B') \in (\alpha \setminus \beta) \), then \( (A'; B') \prec_A (A'; B'') \) for some \( (A'; B'') \in \alpha \). In either case, \( D \) satisfies condition (3) in Definition 2. Thus, \( D \) is \( \prec_A \)-saturated. \( \square \)

**Algorithm 1:** The \( \prec_{\mathcal{U}} \)-maximal \( \rightarrow \)-strategy.

The definitions of strategy and maximal strategy used in this paper are more general than those in [16, 22]. In that paper, the only possible non-determinism in \( \rightarrow_5 \) arises from \( \rightarrow \). In the formalization presented in this paper, as illustrated by strategies \( s_1 \) and \( s_3 \), the synchronous relation \( \rightarrow \) can be non-deterministic even when the relation \( \rightarrow_5 \) is deterministic. The practical implication of this generalization is twofold. First, it makes simpler the task of specifying a synchronous system because there are fewer assumptions to check on its atomic relation. In some cases, proving that the atomic relation is deterministic can be a daunting task. Second, more general synchronous systems can be formally modeled, executed, and analyzed by allowing a non-deterministic atomic relation. For example, the extension from a deterministic atomic relation to a non-deterministic one allows for the study of variants of languages such as PLEXIL, where the changes to the semantics may lead to non-determinism and race conditions. In these cases, any analysis without the extension to a non-deterministic atomic relation could be unsound or incomplete, or simply bogus.

3. Overview of rewriting logic

3.1. Order-sorted rewrite theories

An order-sorted signature [2] is a triple \( \Sigma = (S, \leq, F) \), where \( (S, \leq) \) is a finite poset of sorts and \( F \) is a finite set of function symbols. Set \( X = \{ X_s \}_{s \in S} \) is an \( S \)-sorted family of disjoint sets of variables with each \( X_s \) countably infinite. The set of terms of sort \( s \) is denoted by \( T_\Sigma(X)_s \) and the set of ground terms of sort \( s \) is denoted by \( T_\Sigma.s \). It is assumed that

Please cite this article in press as: C. Rocha, C. Muñoz, Synchronous set relations in rewriting logic, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.07.008
for each sort \( s \), \( T_{\Sigma,s} \) is nonempty; this assumption simplifies the proof system to follow. Algebras \( T_{\Sigma}(X) \) and \( T_{\Sigma} \) denote the respective term algebras. The set of variables of a term \( t \) is written \( \text{vars}(t) \) and is extended to sets of terms in the natural way. A term \( t \) is called ground if \( \text{vars}(t) = \emptyset \). A substitution \( \theta \) is a sorted map from a finite subset \( \text{dom}(\theta) \subseteq X \) to \( \text{ran}(\theta) \subseteq T_{\Sigma}(X) \) and extends homomorphically in the natural way. Substitution \( \theta \) is called ground if \( \text{ran}(\theta) \) is ground. Expression \( \theta(t) \) denotes the application of \( \theta \) to term \( t \).

A \( \Sigma \)-equation is a sentence \( t = u \) if \( \text{cond} \), where \( t = u \) is a \( \Sigma \)-equality with \( t, u \in T_{\Sigma}(X) \), for some sort \( s \in S \), and the condition \( \text{cond} \) is a finite conjunction of \( \Sigma \)-equalities. An equational theory is a pair \( (\Sigma, E) \) with order-sorted signature \( \Sigma \) and finite set of \( \Sigma \)-equations \( E \). For a \( \Sigma \)-equation \( \phi \), the judgment \( (\Sigma, E) \vdash \phi \) states that \( \phi \) can be derived from \( (\Sigma, E) \) by the deduction rules in [11]. In this case, it holds that \( \phi \) is valid in all models of \( (\Sigma, E) \). An equational theory \( (\Sigma, E) \) induces the congruence relation \( =_E \) on \( T_{\Sigma}(X) \) defined for any \( t, u \in T_{\Sigma}(X) \) by \( t=_{E} u \) if and only if \( (\Sigma, E) \vdash t = u \). The equivalence class of a term \( t \in T_{\Sigma}(X) \) induced by the equivalence relation \( =_E \) is denoted by \( [t]_E \). The \( \Sigma \)-algebras \( T_{\Sigma,E}(X) \) and \( T_{\Sigma,E} \) denote the quotient algebras induced by \( =_E \) over the algebras \( T_{\Sigma}(X) \) and \( T_{\Sigma} \). The algebra \( T_{\Sigma,E} \) is called the initial algebra of \( (\Sigma, E) \).

A \( \Sigma \)-rule is a sentence \( \phi : t \Rightarrow u \) if \( \text{cond} \), where \( \phi \) is its name, \( t \Rightarrow u \) is a \( \Sigma \)-sequent with \( t, u \in T_{\Sigma}(X) \), for some sort \( s \in S \), and the condition \( \text{cond} \) is a finite conjunction of \( \Sigma \)-equalities. Note that sequents are excluded from the condition of a sequent. A rewrite theory is a tuple \( \mathcal{R} = (\Sigma, E, R) \) with equational theory \( \mathcal{E}_R = (\Sigma, E) \) and a finite set of \( \Sigma \)-rules \( R \). For \( \mathcal{R} = (\Sigma, E, R) \) and \( \phi \) a \( \Sigma \)-rule, the judgment \( \mathcal{R} \vdash \phi \) states that \( \phi \) can be derived from \( \mathcal{R} \) by the deduction rules in [2]. In this case, it holds that \( \phi \) is valid in all models of \( \mathcal{R} \). For \( \mathcal{R} = (\Sigma, E, R) \), the set of atomic predicates on \( \Sigma \)-states. The set of atomic propositions \( \Xi \) contains \( \Sigma \) and a sort \( \mathbb{B} \) with constant symbols \( \bot \) and \( \top \), predicate symbols \( \phi : \text{State} \rightarrow \mathbb{B} \) for each \( \phi \in \Phi \), and optionally some auxiliary function symbols. Equations in \( E_\Phi \) define the predicate symbols in \( \Sigma_\Phi \) and auxiliary function symbols, if any, including the Boolean operations on the sort \( \mathbb{B} \). For \( \phi \in \Phi \) and a ground term \( t \in T_{\Sigma,\text{State}} \), the semantics of \( \phi \) in \( \mathcal{T}_R \) is defined by \( \mathcal{E}_\Phi \) as follows: \( \phi(t) \) holds in \( \mathcal{T}_R \) if and only if \( \mathcal{E}_\Phi \vdash \phi(t) = \top \). This defines the Kripke structure

\[
K^\Phi_{\mathcal{T}_R} = (T_{\Sigma/E,\text{State}}, \Rightarrow_{\mathcal{T}_R,\text{State}}, L_{\Phi})
\]

with transition relation \( \Rightarrow_{\mathcal{T}_R,\text{State}} \) denoting the restriction of \( \Rightarrow_{\mathcal{T}_R} \) to terms in \( T_{\Sigma/E,\text{State}} \) and labeling function \( L_{\Phi} \) defined for any \( t \in T_{\Sigma,\text{State}} \) by \( \phi \in L_{\Phi}(t) \), written \( K^\Phi_{\mathcal{T}_R} \vdash t \models \phi \), if and only if \( \phi(t) \) holds in \( \mathcal{T}_R \). All formulas of the Linear Temporal Logic (LTL) can be interpreted in \( K^\Phi_{\mathcal{T}_R} \) in the standard way.

### 3.4. Executability conditions

Because rewriting logic’s rules of deduction [2] are based on equational deduction, it may be undecidable to check membership in \( \Rightarrow_{\mathcal{T}_R} \) for a rewrite theory \( \mathcal{R} = (\Sigma, E, R) \). Furthermore, even if deduction with \( E \) is decidable, there may be an infinite number of terms in \( E \)-equivalence classes; so, an infinite search may be needed to find a term \( t' \in [t]_E \) that can be rewritten with \( \Rightarrow_{\mathcal{T}_R} \).

The following conditions on a rewrite theory \( \mathcal{R} = (\Sigma, E, R) \) make rewriting with equations \( E \) and with rules \( R \) modulo \( E \) computable, and are assumed throughout this paper. First the set of equations \( E \) of \( \mathcal{R} \) can be decomposed into a disjoint union \( E \cup A \), with \( A \) a collection of axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a matching algorithm modulo \( A \) producing a finite number of \( A \)-matching substitutions, or failing otherwise. The
4. Synchronous set relations in rewriting logic

This section presents the infrastructure \((\Sigma, E, R)\) for specifying and executing in Maude a synchronous relation defined from a language \(\mathcal{L}\).

4.1. The synchronous language \(\mathcal{L}\)

Recall that definitions in Section 2 are given for an abstract set \(T\), an abstract relation \(\rightarrow\), and an abstract priority relation \(\prec\). The language \(\mathcal{L}\) is given by the user as an order-sorted rewrite theory \((\Sigma, E, \mathcal{L})\) that enables the definition of concrete mathematical objects \(T_{\Sigma, \mathcal{L}}\), \(\rightarrow\), and \(\prec\) that implement \(T\), \(\rightarrow\), \(\prec\), respectively. The rewrite theory \((\Sigma, E, \mathcal{L})\) extends the rewrite theory \((\Sigma, E, R)\), which provides an infrastructure with definitions of basic sorts and data structures that are suitable for specifying set rewriting systems. This rewrite theory exploits rewriting logic’s reflection capabilities available in Maude to soundly and completely simulate the synchronous relation \(\rightarrow\mathcal{L}\), with \(s\) the \(\prec\)-maximal strategy for \(\rightarrow\mathcal{L}\).

The infrastructure in Maude contains several Maude files. The following is a description of the files meaningful to the exposition in this section:

- **theory-closure.maude**: general purpose functionality for meta-level operations on modules.
- **base.maude**: specifies the constructs supporting the user-definable data. It includes several modules defining the sorts and function symbols explained in Section 4.11.
- **synchr.maude**: specifies, in several modules, the realization of Algorithm 1, which includes the functions explained in Sections 4.1.2, 4.1.3, and 4.1.4.
- **smaude.maude**: the infrastructure’s main file. It contains module SMAUDE specifying the rewrite rule \textit{sync}, explained in Section 4.2, whose main purpose is to invoke the implementation of Algorithm 1, resulting in the simulation of synchronous set rewriting in Maude.

4.1.1. The set \(T_{\Sigma, \mathcal{L}, \text{Elem}}\)

The set of ground terms \(T_{\Sigma, \mathcal{L}, \text{Elem}}\) of the rewrite theory \((\Sigma, E, \mathcal{L})\) implements the abstract set \(T\) of Section 2. The sort \(\text{Elem}\) represents elements in \(\Sigma\) having the form

\[
\langle m | a_1 : e_1, \ldots, a_n : e_n \rangle,
\]

where \(m\) is an identifier of sort \textit{Eid} and \(a_1 : e_1, \ldots, a_n : e_n\) is a map of sort \textit{Map}. A map is a collection of attributes. An attribute is a pair \(a : e\) where \(a\) is an attribute identifier of sort \textit{Aid} and \(e\) is an expression of sort \textit{Expr}. Attributes are a flexible way of defining the internal state of an element. Sorts \textit{Aid} and \textit{Eid} are declared as subsorts of \textit{Expr}.

The set \(\mathcal{U}\) of Section 2 corresponds to the set of ground terms \(T_{\Sigma, \mathcal{L}, \text{Ctx}}\), where the sort \(\text{Ctx}\) represents sets of elements of sort \textit{Elem}. A context is an element of sort \textit{Ctx}. The infrastructure \((\Sigma, E, R)\) includes as part of its signature \(\Sigma\) the definition of the constant

\[
\#\text{CTX}\# :\rightarrow \text{Ctx},
\]

which is internally maintained by the infrastructure. This constant is used at runtime to reference, when extended by the language \(\mathcal{L}\), the context identifying the current state of execution in the synchronous semantics of \(\mathcal{L}\). The name of this constant was chosen deliberately to contain the number symbol "#" as delimiter in order to minimize the chances of naming clashes with user-defined constants and variables in the signature \(\Sigma\).

The sort \textit{Val} is defined in \(\Sigma\) as a subsort of \textit{Expr} and represents built-in values such as Boolean and numerical values. Function symbols

\[
eval : \textit{Expr} \rightarrow \textit{Val},
\]

\[
eval : \textit{Ctx} \times \textit{Expr} \rightarrow \textit{Val}
\]

are part of the signature \(\Sigma\). The unary version of function eval is introduced as a short-hand for evaluating an expression in the current state of execution, which at runtime, is identified by the context \(#\text{CTX}\#\), as explained above. This is specified in the infrastructure by the following equation in \(E\), for any expression \(e\) of sort \textit{Expr}:

\[
eval(e) = \text{eval}(\#\text{CTX}\#, e).
\]
For the binary version of eval no defining equations are given in $E$ since its definition comes later, when defined by the user in $E_L$.

The signature $\Sigma$ also defines the constant

\[
\text{MODULE } - \text{NAME } : \longrightarrow \text{ Qid}
\]

for identifying the module name that implements the language $L$. The value of this constant is maintained by the user when defining the language $L$. It is used by the infrastructure to obtain a representation of $L$ in order to compute the synchronous relation of $L$ at the meta-level. The sort Qid is used in Maude for representing quoted identifiers and it is part of the standard prelude.

The user is free to extend the signature $\Sigma$ in $\Sigma_L$ with any syntax and subsorts for element identifiers, attribute identifiers, and expressions. However, it is assumed that attribute identifiers within a map and element identifiers within a context are unique. It is also assumed that the theory $(\Sigma_L,E_L)$ includes a complete equational interpretation of eval’s binary version for the set of expressions in $\Sigma_L$.

4.1.2. The relation $\Rightarrow_L$

The synchronous relation in Definition 1 is given for an abstract atomic relation $\rightarrow$. In a concrete language, such as $L$, this relation represents atomic computational steps that are synchronously executed. For that reason, the concrete relation $\Rightarrow_L$ is called the atomic relation. As shown in [16], the atomic relation is usually parametric with respect to a context that, in this infrastructure, provides global information to the function eval. Henceforth, the atomic relation with respect to a context $\Gamma$ of sort Ctx will be denoted $\Rightarrow^\Gamma_L$.

The atomic relation $\Rightarrow_L$ is specified in $R_L$ through atomic rules.

Definition 4 (Atomic rules). Let $\Sigma_L$ be an order-sorted signature extending $\Sigma$. An atomic $\Sigma_L$-rule is a $\Sigma_L$-rule $\triangleright l \Rightarrow r$ if cond such that:

- rule name $\triangleright l$ has the form $c\triangleright n$, where $c$, the component of $l$, is an identifier, and $n$, the rank of $l$, is a natural number;
- $l$ does not contain attribute identifier variables, i.e., $\text{vars}(l) \cap X_{\text{attr}} = \emptyset$; and
- attribute names appearing in an element term in $r$ are named for that same element term in $l$, i.e., if $(i \mid m') \in r$ and $(a:e') \in m'$, then there is $(i \mid m) \in l$ such that $(a:e) \in m$ for some $e \in T_{\Sigma_L}(X)_{\text{Expr}}$.

An atomic $\Sigma_L$-rule specifies transitions of contexts (possibly) constrained by a condition that may involve expressions in the syntax of $L$. The component and rank of $\Sigma_L$-rules are used to define the priority relation $\ll_L$. The restriction on attribute identifier names and variables is to prevent the user from defining an atomic relation $\Rightarrow_L$ for which computing a $\Rightarrow_L$-reduction could be highly inefficient or even incorrect.

Definition 5 (Atomic relation $\Rightarrow_L$). Let $L = (\Sigma_L,E_L,R_L)$ be a rewrite theory with $(\Sigma_L,E_L)$ extending $(\Sigma,E)$ and $R_L$ a collection of atomic $\Sigma_L$-rules with different names. For a rule $\triangleright l \Rightarrow r$ if cond in $R_L$, the (parametric) relation $\Rightarrow_{\Sigma_L}^\Gamma$, with parameter $\Gamma \in T_{\Sigma_L,\text{ctx}}$, denotes the set of pairs $(A;B)$ in $T_{\Sigma_L,\text{ctx}} \times T_{\Sigma_L,\text{ctx}}$ such that there is a ground substitution $\theta : T_{\Sigma_L}(X) \longrightarrow T_{\Sigma_L}$ satisfying $\text{cond} \triangleright l$, $(A,A') = \theta$, and $B = r \theta$ in $L$, where any expression is evaluated in $\Gamma$ and any element in $A$ is an element in $B$ (but not necessarily with the same attribute values). The atomic relation $\Rightarrow_L$ is the indexed set

\[
\{\Gamma \mapsto \triangleright l \Rightarrow r \text{ if } \text{cond} \}\Gamma \in T_{\Sigma_L,\text{ctx}},\theta \in R_L.
\]

In Definition 5, $A$, $A'$, $B$, and $\Gamma$ are ground terms of sort Ctx. Furthermore, the term $B$ is a variant of $A$ in which some expressions and attributes have been modified. In particular, $A$ and $B$ have the same number of elements with the same element and attribute identifiers. This means that the atomic relation does not delete or create elements or attributes in $A$. This restriction simplifies the technical development of $(\Sigma,E,R)$. On the other hand, the term $A'$ corresponds to a context, possibly empty, that may contain information needed for the atomic transition but that remains unchanged during the transition. That is, $A$ and $A'$ can be seen, respectively, as the ‘write once’ and ‘read many times’ terms of an atomic transition. In any case, creation and deletion of elements and attributes can be encoded by using additional attributes. Also observe that, due to the syntactical restrictions of atomic rules in Definition 4, equational sentences $\text{cond} \triangleright l$, $(A,A') = \theta$, and $B = r \theta$ can be checked in $(\Sigma_L,E_L)$ because they are equational expressions that, although may depend on context $\Gamma$, do not depend on $R_L$. In this case, since $A$, $A'$, $B$, and $\Gamma$ are ground, and $\theta$ is a ground substitution, equational deduction with $(\Sigma_L,E_L)$ in Definition 5 coincides with validity in the initial model $T_{\Gamma\Sigma_L,E_L}$.

It is noted that the atomic relation $\Rightarrow_L$ and the rewrite relation $\Rightarrow_L$ induced by the rewrite theory $L$, in general, do not coincide for ground context terms. In particular, $\Rightarrow_L$ is defined as the top-most application of the atomic rules, while $\Rightarrow_L$ is defined as the congruence closure of those rules.

4.1.3. The priority $\ll_L$

For a given context $\Gamma$, the elements in $\Rightarrow^\Gamma_L$ can be regarded as tuples of the form $(A,B,c,m)_\Gamma$ as a short-hand for $A \Rightarrow^\Gamma_L B$, with $c \leftarrow m \in R_L$. The set $\ll_L = \{\ll_L(\Gamma)\Gamma \in T_{\Sigma_L,\text{ctx}}$ is defined automatically by the infrastructure as
(A', B', c', m') \triangleright L(\Gamma) (A, B, c, m) \triangleright L(\Gamma) \quad \text{if} \quad A \subseteq \Gamma \land A' \subseteq \Gamma \land c = c' \land m < m',

where $\triangleright$ is the usual order on natural numbers.

**Theorem 2.** The indexed set $<$ is a priority for $\rightarrow L$.

**Proof.** It is enough to prove that $<_{L(\Gamma)}$ is a strict partial order, for any $\Gamma \in T_{S\mathcal{L},Ctx}$. Irreflexivity of $<_{L(\Gamma)}$ follows from the irreflexivity of $\prec$. Transitivity of $<_{L(\Gamma)}$ follows from the fact that if $(A', B', c', m') \triangleright L(\Gamma) (A, B, c, m) \triangleright L(\Gamma)$ and $(A', B', c', m') \triangleright L(\Gamma) (A, B, c, m) \triangleright L(\Gamma)$, then $A'' \subseteq \Gamma$, $A \subseteq \Gamma$, $c'' = c' = c$, and $m < m' < m''$. Therefore, $(A'', B'', c'', m'') \triangleright L(\Gamma) (A, B, c, m) \triangleright L(\Gamma)$.

The priority $<_L$ is an indexed collection of strict partial orders. In particular, for each $\Gamma \in T_{S\mathcal{L},Ctx}$, priority $<_{L(\Gamma)}$ compares two elements of $\rightarrow L$ if they are computed with the same context and they originate from atomic $\Sigma_L$-rules having the same component. It assigns a higher priority to elements with smaller rank.

4.1.4. Implementing Algorithm 1

The rewrite theory $(\Sigma, E, R)$ includes the function

\[
\text{max-strat} : \text{Module} \times \text{Ctx} \rightarrow \text{TransitionSet}
\]

that computes the $<_L$-maximal $\rightarrow L$-strategy, where $\Gamma \in T_{S\mathcal{L},Ctx}$ is the parameter of the relation $\rightarrow L$. That function implements Algorithm 1 (see Section 2) in Maude using the meta-level capabilities of the system. It takes as input the meta-representation $\overline{L}$ of language $\mathcal{L}$ and ground context $\Gamma$, and returns the collection $s(\Gamma)$, where $s$ is the $<_L$-maximal $\rightarrow L$-strategy. The sort $\text{Module}$ is used in Maude for terms denoting meta-representations of modules and it is part of Maude’s standard prelude. The sort $\text{Transition}$ denotes sets of pairs in $T_{S\mathcal{L}}(\overline{X})_{Ctx}$ and sort $\text{TransitionSet}$ denotes collections of transitions with multiset union operator ‘$|$’.

The function $\text{max-strat}(\overline{L}, \Gamma)$ reduces the problem of computing the $<_L$-maximal $\rightarrow L$-strategy into four smaller problems that are handled sequentially by different functions. Namely, it first updates the current state of execution of $\mathcal{L}$ to $\Gamma$, obtaining a module $\mathcal{L}'$ meta-represented by $\overline{L}'$. Second, it computes the one-step atomic relation $\rightarrow L$ on $\Gamma$ by using $\overline{L}'$. Third, it uses the strategy $<_L$ on the resulting collection from the second step to keep only maximal redexes. Fourth, it computes the collection of all saturated sets from the output of the third step.

4.1.4.1. Updating the current state of execution of $\mathcal{L}$ As explained in Section 4.1.1, constant $\#\text{CTX}$ is used by the infrastructure to reference the context identifying the current state of execution. The function

\[
\text{set-state} : \text{Module} \times \text{Ctx} \rightarrow \text{Module}
\]

on input $\overline{L}$ of sort $\text{Module}$ and $\Gamma$ of sort $\text{Ctx}$ adds the equation $\#\text{CTX} = \Gamma$

producing a module $\overline{L}$. This has the effect that the evaluation of any expression in $\overline{L}$ with eval’s unary version will use $\Gamma$ as the default context. In fact, the task of adding the equation above to $\overline{L}$ is achieved by using Maude’s meta-level and the meta-representation of such equation.

4.1.4.2. Computing the atomic relation $\rightarrow L$ The function

\[
\text{compute-atomic} : \text{Module} \times \text{Ctx} \rightarrow \text{LabeledTransition}
\]

on input $\overline{L}$ and $\Gamma$ computes the atomic relation $\rightarrow L$ on the context $\Gamma$. This function implements the part of Algorithm 1 in lines 3–5. The sort $\text{LabeledTransition}$ represents collections of labeled transitions, i.e., collections of tuples of the form $(A, B, c, m) \rightarrow L$. A labeled transition $(A, B, c, m) \rightarrow L$ is obtained from a rule

\[
c \cdot m : l \Rightarrow r \quad \text{if} \quad \text{cond} \in R_L
\]

that satisfies the conditions in Definition 5. The task of computing the ground substitutions $\theta$ in Definition 5 is achieved by using Maude’s metaXmatch meta-level function for each rule in $R_L$. Also, each tuple $(A, B, c, m) \rightarrow L$ computed by the function compute-atomic is such that if there is an element elem $\in A$ with identifier id, then there is an element elem' $\in B$ with identifier id, and vice versa.

Please cite this article in press as: C. Rocha, C. Muñoz, Synchronous set relations in rewriting logic, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.07.008
4.1.4.3. Applying the strategy $\prec_L$

The function

$$\text{apply-strategy : LabeledTransition}\rightarrow\text{Transition}$$

on input $\Gamma'$ of sort LabeledTransition, computed from $\text{compute-atomic}(\overline{L}, \Gamma)$, uses the strategy $\prec_L$ to filter out those labeled transitions that cannot be part of any $\prec_L$-saturated set for $\Gamma$ (see Definition 2), and returns a term of sort Transition consisting of those pairs $(A; B)$ such that $(A, B, c, m)$ has not been filtered out. This function implements the part of Algorithm 1 in lines 6–10 and does not make use of Maude’s meta-level facilities.

4.1.4.4. Computing all $\prec_L$-saturated sets

The function

$$\text{compute-saturated : Transition}\rightarrow\text{TransitionSet}$$

implements recursively the part of Algorithm 1 in lines 12–19 and does not make use of Maude’s meta-level facilities.

Henceforth, the strategy $s$ will denote the $\prec_L$-maximal $\rightarrow L$-strategy as computed by max-strat.

4.2. Simulation of $s\rightarrow L$

The set of $\Sigma$-rules $R$ of the order-sorted rewrite theory $(\Sigma, E, R)$ includes only one rule: for $l, r \in \text{Ctx}, T \in \text{Transition}$, and $S \in \text{TransitionSet}$

$$\text{sync} : \{l\} \Rightarrow \{r\} \quad \text{if} \quad T, S := \text{max-strat}(\overline{L}, l) \wedge r := \text{update}(l, T).$$

This rule, along with the rules $R_L$ provided by the user, implements the serialization algorithm defined in Algorithm 1. Note in the condition of sync that the equality symbol ‘$=$’ has been replaced by the operator ‘$:\equiv$’ in both equalities, which is idiom in Maude for a matching equality. For instance, expression

$$T, S := \text{max-strat}(\overline{L}, l)$$

means that, when evaluating the condition, the right-hand side term

$$\text{max-strat}(\overline{L}, l)$$

is first evaluated to canonical form and then the left-hand side pattern

$$T, S$$

is used to match that canonical form (modulo associativity, commutativity, and identity, which are the axioms for sort TransitionSet).

The function

$$\text{update : Ctx} \times\text{Transition}\rightarrow\text{Ctx}$$

on input $A$, a ground context of sort Ctx, and a ground transition term

$$C = \{(A_1; B_1), \ldots, (A_n; B_n)\},$$

computes the ground context

$$B = \left( A \setminus \bigcup_{1 \leq i \leq n} A_i \right) \cup \bigcup_{1 \leq i \leq n} B_i.$$

It is noted that the rule sync acts on contexts that are syntactically wrapped by curly brackets, that is, terms of the form $\{A\}$ with $A$ a ground context term. Those terms are of sort Sys. The curly brackets operator prevents its context $A$ to be directly rewritten by the user defined atomic rules in $R_L$. The actual application of those rules is done by the function update.

Rule sync is non-deterministic because a matching ground substitution for $l$ and satisfying its condition depends on the choice of $T$, i.e., on all possible transitions computed by max-strat. However, there will be exactly one rewrite with sync for each transition.

**Theorem 3.** Let $L = (\Sigma_L, E_L, R_L)$ be an extension of $(\Sigma, E, R)$. For $A, B \in T_{\Sigma, \text{Ctx}}$, the following equivalence holds:

$$L \vdash \{A\} \Rightarrow \{B\} \equiv A \overset{s}{\rightarrow_L} B,$$

where $s$ denotes the $\prec_L$-maximal $\rightarrow L$-strategy as computed by max-strat.
Proof. The key observation is that because max-strat computes the \( \prec_L \)-maximal \( \rightarrow_L \)-strategy \( s \), the following equivalence holds:

\[
C \in s(A) \iff (3C' \in T_{\Sigma_L, TransitionSet}) C' \models_{E_L} max-strat(L, A).
\]

(\( \implies \)) Since \( \{A\} \) can be rewritten only by rule sync \( \in R \), there is a ground substitution \( \theta : X \rightarrow T_{\Sigma_L} \) satisfying \( A \models_{E_L} l\theta \), \( B \models_{E_L} r\theta \), \( T\theta \), \( S\theta \models_{E_L} max-strat(L, l\theta) \), and \( r\theta \models_{E_L} update(l\theta, T\theta) \). By the observation above, \( \theta \in s(A) \). Then, from the definition of update, it follows that \( \theta \models_{E_L} B \).

(\( \impliedby \)) If \( A \models_{E_L} B \), there is \( C = \{(A_1; B_1), \ldots, (A_n; B_n)\} \in s(A) \) such that \( B = (A \setminus \bigcup_{1 \leq i \leq n} A_i) \cup \bigcup_{1 \leq i \leq n} B_i \). By the observation above and the definition of update, there is \( C' \in T_{\Sigma_L, TransitionSet} \) such that \( C, C' \models_{E_L} max-strat(L, A) \) and \( B \models_{E_L} update(A, C) \). Then substitution \( \theta \) satisfying \( A \models_{E_L} l\theta \) witnesses \( L \vdash \{A\} \models \{B\} \). \( \square \)

One key advantage of this approach is that, while it offers support for the execution of a synchronous relation \( \prec_L \), it does that by simulating \( \rightarrow_L \) using the standard asynchronous semantics of Maude. Therefore, all commands available in Maude for executing and verifying rewrite relations are directly available for \( \rightarrow_L \). Sections 5 and 6 illustrate some of these features for, respectively, a simple synchronous language and a non-toy synchronous plan execution language developed by NASA.

5. Executable semantics of a simple synchronous language

Module SMAUDE implements in Maude the rewrite theory \((\Sigma, E, R)\) presented in Section 4. This section illustrates the use of SMAUDE by giving the small-step semantics of a simple synchronous language with arithmetic expressions. Code snippets of the Maude language are used to illustrate explicitly how the infrastructure in SMAUDE is extended. Therefore, some familiarity with Maude's syntax is assumed (see [4] for a reference to the Maude language and system).

Consider a language \( S \) that consists of two kinds of elements: memory elements \( Mem(m, v) \) and assignment elements \( m := e \), where \( m \) denotes a memory name, \( v \) denotes a numerical value, and \( e \) denotes an arithmetic expression. Arithmetic expressions are recursively formed using memory names, numerical values, and expressions of the form \( e_1 + e_2 \), where \( e_1 \) and \( e_2 \) are arithmetic expressions. In this case, set \( T \) consists of all elements having the form \( Mem(m, v) \) or \( m := v \).

The small-step semantics of \( S \) requires the definition of an evaluation function \( \text{eval} \) that takes as inputs a context \( \Gamma \), which is a set of elements \( T \), and an arithmetic expression \( e \). It is inductively defined on expressions:

\[
\text{eval}(\Gamma, e) = \begin{cases} 
v & \text{if } e \text{ is the numerical value } v, \\
v & \text{if } e \text{ is the memory name } m \text{ and } Mem(m, v) \in \Gamma, \\
v_1 + v_2 & \text{if } e \text{ has the form } e_1 + e_2, \quad v_i = \text{eval}(\Gamma, e_i) \text{ for } i \in \{1, 2\}.
\end{cases}
\]

The \( \{\text{parametric}\} \) atomic relation \( \rightarrow_{S} \) of the language \( S \) is defined for a context \( \Gamma \) by \( A \models_{S} B \) if and only if \( A \subseteq \Gamma \), \( A = \{Mem(m, v), m := e, B = \{Mem(m, u), m := e, u = \text{eval}(A, e)\} \}, \text{ and } u \models_{S} \text{for some memory name } m, \text{values } v \text{ and } u, \text{and expression } e \).

The semantic relation of the language is the relation \( \models_{S} \) \((\text{or } \models_{S})\), where \( s \) is the \( \prec_{-} \)-maximal \( \rightarrow_{\Gamma} \)-strategy, \( \Gamma \) is a ground context, and \( \prec_{-} \)is the empty priority.

Example 4. If \( \Gamma = \{Mem(x, 3), Mem(y, 4), x := y, y := x\} \), then:

\[
Mem(x, 3), Mem(y, 4), x := y, y := x \models_{S} \Gamma \models_{S} Mem(x, 4), Mem(y, 3), x := y, y := x.
\]

Language \( S \) is specified by the Maude system module SIMPLE, which includes system module SMAUDE, and has the following syntax:

\[
\text{mod SIMPLE}
\]

\[
\text{including SMAUDE}.
\]

\[
\text{eq MODULE-NAME} = \text{SIMPLE}.
\]

\[
\ldots
\]

\[
\text{endm}
\]

Note that constant \( MODULE-NAME \) is identified with the quoted identifier representing the name of module \( SIMPLE \), as required by the infrastructure (see Section 4.1.1).

Element identifiers include the following constructors with sort \( Eid \):

\[
\text{op a : Nat \rightarrow Eid [ctor].}
\]

\[
\text{ops x y : \rightarrow Eid [ctor].}
\]

Memory elements use constructors \( x \) and \( y \) for element identifiers, and assignment elements use constructors \( a(\_\_) \) for element identifiers.
Attribute identifiers include the following constructors with sort Aid:

\[
\text{ops mem body to : -> Aid [ctor] .}
\]

Memory elements have attribute \texttt{mem} as their only attribute, while assignment elements have attributes \texttt{body} and \texttt{to} as their only attributes. In the syntax of \textit{SIMPLE}, memory element \textit{Mem}(x, v) and an assignment element \texttt{x:=e} can be represented, respectively, by elements

\[
<x | \text{mem : v}> <a(1) | \text{body : e, to : x}>
\]

Built-in natural numbers are values of the language and addition corresponds to the built-in one in Maude. These are specified in \textit{SIMPLE} with the following subsort and operation declarations:

\[
\text{subsort} \text{Nat < Val .}
\]
\[
\text{op}_+ : \text{Expr Expr -> Expr [ditto] .}
\]

Expressions are evaluated equationally by following the definition of \texttt{eval}:

\[
\begin{align*}
\text{var C : Ctx .} & \quad \text{vars I J : Eid .} & \quad \text{vars E E' : Expr .} \\
\text{var M : Map .} & \quad \text{var N : Nat .} \\
\text{eq eval(C,N)} & = N . \\
\text{eq eval(}(<I | \text{mem : N, M > C}), I) & = N . \\
\text{eq eval(C,E + E')} & = \text{eval(C,E)} + \text{eval(C,E')} .
\end{align*}
\]

Atomic rule \texttt{r-1} specifies the atomic relation of the language:

\[
\text{rl [r-1] :}
\begin{align*}
< I | \text{mem : N} > < J | \text{body : E, to : I} > \\
\Rightarrow < I | \text{mem : eval(E)} > .
\end{align*}
\]

The specification of atomic rules is slightly different to the usual specification of rules in rewriting logic. First, in the left-hand side of an atomic rule, it is sufficient to mention only the attributes involved in the atomic transition. In this case, \texttt{SMAUDE} will complete each left-hand side term by automatically adding a variable of sort \texttt{Map}, unique for each element, before any matching is performed. Second, in the right-hand side of an atomic rule, it is sufficient to mention the elements and the attributes that can change in the atomic step. In this case, \texttt{SMAUDE} updates in the current state only the attributes of the elements occurring in the right-hand side of the rule, while keeping the other ones intact. So, in atomic rule \texttt{r-1}, the only attribute that can change is attribute \texttt{mem} of the memory element. Note also that in the right-hand side of \texttt{r-1}, a unary version of function \texttt{eval}, without mention of any particular context, is used; \texttt{SMAUDE} will automatically extend it to its binary counterpart, for the given context, when computing function \texttt{max-strat}.

The context \( \Gamma \) in \textbf{Example 4}, written in the syntax of \textit{SIMPLE}, is

\[
<x | \text{mem : 3} > < y | \text{mem : 4} > \\
<a(1) | \text{body : y, to : x} > < a(2) | \text{body : x, to : y} > .
\]

Maude's \texttt{search} command can be used to compute, for instance, the one-step synchronous semantic relation of the language in \textbf{Example 4} from context \( \Gamma \):

\[
\text{Maude> search \{ \Gamma \} } \Rightarrow 1 \text{X:Sys .}
\]
\[
\text{search in SIMPLE : ( \Gamma ) } \Rightarrow 1 \text{X:Sys .}
\]
\[
\text{Solution 1 (state 1)}
\]
\[
\text{states: 2} \quad \text{rewrites: 514 in 53ms cpu (54ms real) (9655 rewrites/second)}
\]
\[
\text{X:Sys } \rightarrow ( \langle x | \text{mem : 4} > \langle y | \text{mem : 3} > \\
\langle a(1) | \text{body : y, to : x} > \langle a(2) | \text{body : x, to : y} > )
\]
\[
\text{No more solutions.}
\]

\textbf{6. A rewriting logic semantics of PLEXIL}

This section presents a rewriting logic semantics of the Plan Execution Interchange Language (PLEXIL) developed with the infrastructure in Maude introduced in Section \textbf{4}. The overview of PLEXIL semantics and the running example of this section are borrowed from \cite{16}.
6.1. Overview of PLEXIL

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 memories, i.e., local variables. The primitive actions of a plan are specified in the leaf nodes. Leaf nodes can be assignment nodes, which assign values to memories, 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 status, which can be inactive, waiting, executing, finishing, iterationended, or finished, and an execution outcome, which can be none, success, or failure. They can declare memories, accessible to the node in which they are declared and all its descendants. In contrast to memories, which have a hierarchical scope, the execution status and the execution outcome of a node are available to all nodes in the plan. Assignment nodes have also a priority that is used to solve race conditions. The internal state of a node consists of the current values of its execution status, execution outcome, and memories.

Each node is equipped with a set of gate conditions and check conditions that govern the execution of a plan. The gate conditions are start condition, which specifies when a node starts its execution, end condition, which specifies when a node ends its execution, repeat condition, which specifies when a node can repeat its execution, and skip condition, which specifies when the execution of a node can be skipped. The check conditions signal abnormal execution states of a node and are pre-condition, post-condition, and invariant. The language includes basic Boolean, arithmetic, and string expressions. It also includes lookup expressions that read the value of external variables provided to the plan by the functional layer. Expressions appear in conditions, assignments, and arguments of commands.

The execution in PLEXIL is driven by external events 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 status. These internal modifications may trigger more changes in gate conditions, which in turn are synchronously processed until quiescence by 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.

Consider the PLEXIL plan in Fig. 1. The plan consists of a root node Exchange of type list, and leaf nodes SetX and SetY of type assignment. The node Exchange declares two memories x and y. The values of these memories are exchanged by the synchronous execution of the node assignments SetX and SetY. The node Exchange also declares a start condition and an invariant condition. The start condition states that the node waits for an external variable T to be greater than 10 before starting its execution. The invariant condition states that at any execution step the values of x and y add to 3.

6.2. Node transitions

The atomic relation of PLEXIL defines the changes in the internal state of nodes as consequences of changes in their gate conditions. It is defined by 42 individual transitions, indexed by type and execution status of nodes into a dozen groups (see [19] for a version of node transition diagrams or [6] for a version of transition rules).

For instance, Fig. 2 depicts the three atomic transitions corresponding to list nodes with status executing. The first transition, labeled with 1, updates the status and outcome of the list node to failing and parent failure, respectively, if the invariant condition of any of its ancestors does not hold. The second transition updates the status and outcome of the list node with failing and invariant failure, respectively, if its invariant condition does not hold. The third transition updates the status of the list node with finishing if its end condition does hold. If there are multiple condition changes that may happen simultaneously for a particular node, the integer labels in the transition diagram are used to represent the precedence order of its atomic transitions. A condition change with precedence order 1 gets priority over any other condition change. A condition change with precedence order 2 is processed if there is no condition change with priority 1 and so on. It is noted that the evaluation of conditions in the transitions takes place with respect to the current state of nodes, memories, and external variables of the given plan.
6.3. The rewriting logic semantics $\mathcal{L}$

The synchronous semantics of the PLEXIL language is specified by the order-sorted rewrite theory $\mathcal{L} = (\Sigma_\mathcal{L}, E_\mathcal{L} \cup A_\mathcal{L}, R_\mathcal{L})$, that extends the rewrite theory $(\Sigma, E, R)$ providing the infrastructure presented in Section 4. This semantics comprises Boolean and arithmetic values and expressions, evaluation of expressions, list and assignment nodes, memories, and external variables, which represents a significant subset of the entire PLEXIL language.

6.3.1. The set $T_{\Sigma_\mathcal{L}.\text{Elem}}$

A node, memory, or external variable is a ground term in $T_{\Sigma_\mathcal{L}.\text{Elem}}$ of the form

\[ \langle N | a_1:e_1, \ldots, a_n:e_n \rangle \]

with nonempty qualified name $N$ of sort $\text{NeQualName} \leq \text{Eid}$ that uniquely identifies the node, memory, or external variable, attribute identifiers $a_1, \ldots, a_n$, and expressions $e_1, \ldots, e_n$.

Attribute identifiers include the following constructors with sort $\text{Aid}$:

\[
\text{type} \quad \text{status} \quad \text{outcome} \quad \text{parent} \quad \text{set} \quad \text{sort} \quad \text{val} \quad \text{init-val} \quad \text{values}
\]

\[ \text{pre} \quad \text{post} \quad \text{inv} \quad \text{start} \quad \text{end} \quad \text{repeat} \quad \text{skip} \]

Node elements have attributes type, status, outcome, parent, pre, post, inv, start, end, repeat, and skip. Additionally, assignment node elements have attribute set that takes as parameter a nonempty qualified name specifying the name of the assignment’s target memory. Attributes pre, post, inv, start, end, repeat, and skip are used to identify the gate and check conditions of a node element. Memory elements have attributes parent, sort, val, and init-val. External variable elements have attributes sort, value, and values. Attributes type, status, and outcome identify the type, i.e., list or assignment, status, and outcome of a node element, respectively. Attribute parent identifies the qualified name of the parent of a node or memory element. Attributes sort, val and init-val identify the typing, i.e., Boolean or integer, current value, and initial value, respectively, of a memory element. Attribute values identifies the values of an external variable at future time steps.

Expressions are defined inductively from nominal constants, Boolean and integer values, memory bindings, lookups on external variables, and Boolean predicates. Nominal constants are used to identify a type, status, outcome, etc., of a node and have sort $\text{Val}$. For instance, list, assg, mem, and ext are nominal values identifying elements of type list, assignment, memory, and external variable, respectively. Boolean values have sort $\text{BVal}$ and integer values have sort $\text{IVal}$, both subsorts of the predefined sort $\text{Val}$. Boolean and integer constant values use constructors

\[
c : \text{Bool} \rightarrow \text{BVal},
\]

\[
c : \text{Int} \rightarrow \text{IVal},
\]

where $\text{Bool}$ and $\text{Int}$ are Maude’s built-in Boolean and integer values. Sorts $\text{BExpr}$ and $\text{IExpr}$ represent Boolean and integer expressions, respectively, and are defined satisfying the following subsort relations:

\[
\text{BVal} \leq \text{BExpr} \leq \text{Expr} \quad \text{and} \quad \text{IVal} \leq \text{IExpr} \leq \text{Expr}.
\]

Memory bindings and lookups on external variables use constructors

\[
bm, bl : \text{NeQualName} \rightarrow \text{BExpr},
\]

\[
im, ii : \text{NeQualName} \rightarrow \text{IExpr}.
\]

Boolean predicates, with sort $\text{BExpr}$, include PLEXIL’s built-in predicates. For instance, the Boolean predicate $\text{anc-inv-false}(N)$ can be used to check if the invariant condition of any ancestor of a node $N$ does not hold. Boolean and integer expressions can use most of the traditional connectives and relations including, for instance, negation and conjunction, and addition and multiplication, respectively.
6.3.2. The evaluation function

Expressions are equationally evaluated using their inductive definition with respect to a given context. For example, evaluation of integer values, memory bindings, external variable lookups, and addition is defined by the following equations, for \( \Gamma \in X_{\text{Ctx}}, \; N \in X_{\text{Eid}}, \; i \in X_{\text{Val}}, \; M \in X_{\text{Expr}}, \) and \( iE_1, iE_2 \in X_{\text{Expr}}:\)

\[
\begin{align*}
\text{eval}(\Gamma, i) & = i, \\
\text{eval}(\Gamma, \text{im}(N)) & = \text{if } (N \mid \text{type} : \text{mem} \land \text{sort} : \text{int} \land \text{val} : i \land M) \in \Gamma, \text{then } c(\text{true}), \text{else } c(\text{false}) \text{.}
\end{align*}
\]

Function \( \text{eval} \) is similarly defined for all expressions considered in \( \mathcal{L} \).

6.3.3. The synchronous relation \( \xrightarrow{s} \mathcal{L} \) and the priority \( \prec_{\mathcal{L}} \)

The atomic rules in \( R_{\mathcal{L}} \) include all transitions for \textit{list} and \textit{assignment} nodes. For example, the following atomic rules in \( R_{\mathcal{L}} \) specify the atomic transitions for \textit{list} nodes with status \textit{executing} depicted in Fig. 2, for \( N \in X_{\text{Eid}}, \; o \in X_{\text{Expr}}, \) and \( bE \in X_{\text{Expr}}:\)

\[
\begin{align*}
exlist-1 : \; (N \mid \text{type} : \text{list} \land \text{status} : \text{executing} \land \text{outcome} : o) & \Rightarrow (N \mid \text{status} : \text{failing} \land \text{outcome} : \text{fail(parent)}) \\
& \text{if } \text{eval}(\text{anc-inv-false}(N)) = c(\text{true}), \\
exlist-2 : \; (N \mid \text{type} : \text{list} \land \text{status} : \text{executing} \land \text{outcome} : o \land \text{inv} : b) & \Rightarrow (N \mid \text{status} : \text{failing} \land \text{outcome} : \text{fail(inv)}) \\
& \text{if } \text{eval}(bE) = c(\text{false}), \\
exlist-3 : \; (N \mid \text{type} : \text{list} \land \text{status} : \text{executing} \land \text{end} : b) & \Rightarrow (N \mid \text{status} : \text{finishing}) \\
& \text{if } \text{eval}(bE) = c(\text{true}).
\end{align*}
\]

As explained in Section 4.1.1, the unary application of the function \( \text{eval} \) is rewritten by the infrastructure into a binary application, where the context of evaluation is the constant \#CTX# maintained by the infrastructure. It is noted that these three atomic rules have the same component name because they correspond to the same group of atomic rules and their ranks correspond to the integer labels in Fig. 2. Atomic rules in different groups have different component names, so that conflicts are not resolved among rules from atomic rules in different groups.

6.3.4. Interaction with the external environment

The set of \( \Sigma_{\mathcal{L}} \)-rules \( R_{\mathcal{L}} \) includes rule \( \text{tick} \) that updates the values of external variables in a context with the next available value, if any. This rule is applicable only if quiescence has been reached by all nodes. It is defined for \( l, r \in X_{\text{Ctx}} \) by:

\[
\begin{align*}
\text{tick} : \; [l] \Rightarrow [r] & \text{ if } \emptyset := \text{max-strat}(\mathcal{L}, l) \\
& \land \text{has-future}(l) = \text{true} \\
& \land \; r := \text{update-ext-vars}(l).
\end{align*}
\]

This rule acts on terms of sort \( \text{Sys} \) and hence it does not specify an atomic transition. Given a ground term \( \{\Gamma\} \), with context \( \Gamma' \), rule \( \text{tick} \) is applied if \( s(\Gamma') = \emptyset \) for the \( \prec_{\mathcal{L}} \)-maximal \( \Gamma' \)-strategy \( s \) (see Section 4.1.2) and there is at least one external variable in \( \Gamma' \) to be updated (specified by auxiliary function \( \text{has-future} \)). Auxiliary function \( \text{update-ext-vars}(\Gamma') \) returns a context similar to \( \Gamma' \) in which the values of external variables with a next available value is updated or they are unchanged otherwise. It is observed that rules \( \text{sync} \) (Section 4.2) and \( \text{tick} \) have mutually unsatisfiable conditions.

6.4. Formal analysis with \( \mathcal{L} \)

As previously stated, one key advantage of using the infrastructure presented in Section 4 is that by simulating the synchronous relation \( \xrightarrow{s} \mathcal{L} \) of a language \( \mathcal{L} \) by means of standard asynchronous rewriting in Maude, all commands available in Maude for executing and verifying rewrite relations are directly accessible to formally analyze \( \xrightarrow{s} \mathcal{L} \).

6.4.1. Simulation and debugging

Maude’s rewrite and search commands can be used to compute, for instance, the \( n \)-step and run-to-completion semantics of plans in \( \mathcal{L} \). In particular, command search is useful in computing all possible execution paths of a PLEXIL plan from a given initial state. Since PLEXIL is deterministic [16], checking for a race condition between two assignment nodes is logically equivalent to checking for more than one execution path, which can be achieved with search. For example, for ground
Status ::= inactive | waiting | executing | finishing | iterationended | failing | finished
Failure ::= parent | invariant | pre | post
Outcome ::= unknown | skipped | success | fail(μ)
Cond ::= start | end | repeat | pre | post | inv
Φ := true | false | status(λ, σ) | outcome(λ, ω) | ψ(λ, δ) | eval − exp(δ)
LTLN := α | ϕ | ϕ ∨ ϕ′ | ϕ ∧ ϕ′ | ϕ ⇒ ϕ′ |
□ϕ | ◦ϕ | ○ϕ | ϕUsP′ | ϕWp′ | ϕRϕ′

with variables

μ : Failure λ : N σ : Status ω : Outcome
ψ : Cond δ : BExprN α : ΦN ϕ, ϕ′ : LTLN

Fig. 3. Parameterized atomic predicates ΦN and LTL formulas LTLN in L.

case Γ in L representing the initial configuration of plan Exchange in Fig. 1, and with external variable T having initial and only value 4, command search verifies that this plan does not have race conditions, which is equivalent to checking that there exists exactly one quiescent state, reachable from the initial state {Γ}, where all nodes have finished executing.

Maude> search { Γ } =>! X:Sys.
search in EXCHANGE : { Γ } =>! X:Sys.
Solution 1 (state 9)
states: 10 rewrites: 7220 in 565ms cpu (566ms real) (12761 rewrites/second)
X:Sys --> {
  < Exchange | type : list, status : finished, outcome : success, pre : c(true), post : c(true), inv : (c(3) == im(memX . Exchange) + im(memY . Exchange)), start : (c(10) < il(T)), end : children-finished(Exchange), repeat : c(false), skip : c(false), parent : nil >
  < x . Exchange | type : mem, sort : int, init-val : c(1), val : c(2), parent : Exchange >
  < T | type : ext, sort : int, val : c(4), values : nil >
}

No more solutions.
states: 10 rewrites: 7220 in 566ms cpu (567ms real) (12753 rewrites/second)

Maude’s strategy language [10] can be used to debug PLEXIL plans with a high degree of precision. For instance, the ‘normalizing’ strategy “!” can be used with parameter sync to obtain the quiescence synchronous reduction

(ssync)!

that computes the quiesence state {Γ′} for a given state {Γ} without updating the values of any external variable in context Γ. Similarly, two quiescence synchronous reductions can be obtained by the strategy

(ssync) ! ; tick ; (ssync)!

where is the ‘sequential composition’ strategy.

6.4.2. LTL model checking

As explained in Section 3.3, a Kripke structure can be associated with the initial reachability model T L of L = (Σ L, E L, R L) by making explicit the intended sort of states in the signature Σ L and the relevant set of atomic predicates on states. LTL model checking of PLEXIL plans in L takes place at the level of sort Sys. The set of atomic propositions ΦN is parameterized by the set of qualified names of nodes, memories, and external variables in the plan to be model checked. The BNF-like notation in Fig. 3 defines the syntax of the atomic propositions ΦN and formulas LTLN automatically available for model checking a plan P with set of qualified names N. The sort of PLEXIL Boolean expressions parameterized by N is denoted by BExprN. Atomic propositions ΦN include the constants true and false, predicates for testing the
status, outcome, and gate and checking conditions of a node. They also include the atomic proposition \( eval \) — \( exp \) for testing PLEXIL’s Boolean expressions. Formulas in \( LTL_{\Lambda} \) include the usual Boolean connectives, and the temporal connectives ‘always’ (\( \square \)), ‘eventually’ (\( \Diamond \)), ‘next’ (\( \bigcirc \)), ‘until’ (\( U \)), ‘weak until’ (\( W \)), and ‘release’ (\( R \)), all interpreted in the standard way.

Given a plan \( p \), a ground initial context \( \Gamma \), and a \( LTL_{\Lambda} \) formula \( \psi \) over the names \( \Lambda \) in \( p \), Maude’s LTL model checker can be used to check

\[
K_{\Lambda(\Gamma)}^{\Phi_{\Lambda(\Gamma)}} \models \psi,
\]

where \( K_{\Lambda(\Gamma)}^{\Phi_{\Lambda(\Gamma)}} \) is the Kripke structure associated with \( T_{\Lambda(\Gamma)} \), having set of states \( T_{\Lambda(\Gamma)}/\pi_{\Lambda(\Gamma), Sys} \), transition relation \( \rightarrow_{\Lambda(\Gamma)} \), and labeling function \( \lambda_{\Lambda(\Gamma)} \).

For example, the formula

\[
\square inv(Exchange, true) \land \Diamond status(Exchange, finished)
\]

for the plan \( Exchange \) in Fig. 1, tests the invariant of node \( Exchange \) and that it will eventually transition to state \( finished \). For example, for ground context \( \Gamma \) in \( L \) representing the initial configuration of plan \( Exchange \) in Fig. 1, and with external variable \( T \) having initial and only value 4, the following result is obtained with Maude’s LTL Model Checker:

Maude> red modelCheck((\( \Gamma \)), \{\}inv(Exchange,c(true))\land\square status(Exchange,finished)) .
reduce in EXCHANGE : modelCheck((\( \Gamma \)), \{\}inv(Exchange,c(true))\land\square status(Exchange,finished)) .
rewrites: 7368 in 581ms cpu (585ms real) (12661 rewrites/second)
result Bool: true

6.5. Comparing \( L \) with another rewriting logic semantics of PLEXIL

As stated in Section 1, another rewriting logic semantics of PLEXIL in Maude has been developed [7]. In that semantics the serialization procedure was manually coded as part of the atomic transitions specification. This section presents a brief comparison between that semantics of PLEXIL and the rewriting logic semantics \( L \) developed in this paper. Henceforth, the rewriting logic semantics in [7] is denoted by \( P \).

Specifications \( L \) and \( P \) comprise 600 and 2500 lines of Maude code, respectively. Approximately, 2100 lines of code in \( P \) account for the functionality present in \( L \). Table 1 presents a size comparison between \( L \) and \( P \) in terms of lines of code required to specify the functionality common to both specifications. It is noted that a significant size difference can be observed for the specification of datatypes and atomic transitions. Datatypes for the manually implemented serialization procedure account for most of the difference between the specifications. The column \( L \) in Table 1 does not include 948 lines of code that implement the generic infrastructure described in Section 4.

In terms of efficiency, \( P \) outperforms \( L \) on average by one order of magnitude. This is to be expected because, although meta-level computations in Maude are efficient, they are not as efficient as object level computations. Therefore, the on-the-fly implementation of the serialization procedure at the meta-level adds overhead to the overall computation of \( L \)’s atomic relation \( \rightarrow_{\Lambda} \). Also, the computation of the maximal reducts strategy for \( \rightarrow_{\Lambda} \) is essentially exponential in the size of the input ground context. However, as it is usually the case for many formal analysis tools in the realm of the Maude system, the idea is to use \( (\Sigma, E, R) \) as a prototype for a future extension of Maude at the C++ level. A complementary idea is to investigate specific-purpose serialization procedures that use language-specific information, such as determinism of the atomic relation in the case of PLEXIL, to obtain more efficient synchronous reductions.

7. Conclusion

Rewriting logic has been used previously as a test bed for specifying and animating synchronous rewrite relations. M. AlTurki and J. Meseguer [1] have studied the rewriting logic semantics of the language Orc, which includes a synchronous reduction relation. T. Serbanuta et al. [18] and C. Chira et al. [3] define the execution of \( P \)-systems with structured data with continuations. The focus of the former is to use rewriting logic to study the (mainly) non-deterministic behavior of Orc programs, while the focus of the latter is to study the relationship between \( P \)-systems and the existing continuation

<table>
<thead>
<tr>
<th>Specifications</th>
<th>L</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datatypes</td>
<td>110</td>
<td>450</td>
</tr>
<tr>
<td>Function eval</td>
<td>270</td>
<td>282</td>
</tr>
<tr>
<td>Atomic relation</td>
<td>155</td>
<td>408</td>
</tr>
<tr>
<td>Executing lists</td>
<td>14</td>
<td>29</td>
</tr>
<tr>
<td>Total</td>
<td>549</td>
<td>1169</td>
</tr>
</tbody>
</table>
framework for enriching each with the strong features of the other. D. Lucanu [9] studies the problem of the interleaving semantics of concurrency in rewriting logic for synchronous systems from the perspective of P-systems. More recently, T. Serbanuta [17] advances the rewriting-based framework \( K \) with resource sharing semantics that enables some kind of synchronous rewriting. J. Meseguer and P. Olveczky [12] present a formal specification of the physically asynchronous logically synchronous architectural pattern as a formal model transformation that maps a synchronous design, together with performance bounds on the underlying infrastructure, to a formal distributed real-time specification that is semantically equivalent to the synchronous design.

The work presented in this paper is closely related to those works in that it presents techniques for specifying and executing synchronous rewrite relations. However, the work presented here is a first milestone towards the development of symbolic techniques for the analysis of synchronous set relations. In particular, the authors strongly believe that the infrastructure presented in Section 4 can be extended with rewriting and narrowing based techniques, in the style of [15], to obtain a deductive approach for verifying symbolic safety properties, such as invariance or race conditions, of synchronous set relations. Another feature that distinguishes this work from related work is the idea of priorities as a mechanism for controlling non-determinism of synchronous relations. Of course, in some cases priorities can be encoded in the condition of rewrite rules, but the treatment here seems to be more convenient and simpler for the end-user. One interesting exercise would be to study how best to implement this feature in the framework \( K \) and for real-time specifications in rewriting logic.

The contribution of this paper to rewriting logic research is the implementation of general synchronous set relations via asynchronous set rewrite systems. This work extends previous work reported in [16] by giving an on-the-fly implementation of the serialization procedure for rewrite theories that supports execution and verification of more general synchronous set relations. The infrastructure exploits rewriting logic’s reflective capabilities and its implementation in Maude, to soundly and completely simulate the synchronous relation associated with an atomic relation and a maximal strategy specified by atomic rules. This work also generalizes the concept of priority, so that more general synchronous set relations are supported both theoretically and in the Maude infrastructure. A priority, as treated in this work, enables non-deterministic synchronous relations even when the atomic relation is deterministic. In [16], the only possible non-determinism in a synchronous relations arises from its atomic relation. A direct benefit to the user from using the infrastructure presented in this paper, is the wealth of Maude’s ground analysis tools for rewrite theories such as its rewrite and search commands, and its LTL Model Checker.

This paper has shown that the current implementation of the Maude infrastructure supports non-trivial synchronous languages such as PLEXIL. The current infrastructure is being integrated into the PLEXIL Interactive Verification Environment (PLEXIL5) [14], a graphical software environment for the validation and verification of PLEXIL programs.

The infrastructure proposed in this paper can help designers of synchronous languages to focus on the synchronous semantic design without shifting their attention to details in the serialization procedure implementation. A size comparison to another rewriting logic semantics of PLEXIL in Maude, which manually implemented the serialization procedure, evidenced the benefits of the infrastructure in terms of elegance and succinctness. However, more cases studies that stress the infrastructure capabilities are needed to streamline the core algorithms and data structures. The ultimate goal is for the infrastructure to be used as a prototype in a future extension of Maude, say Synchronous Maude, that natively supports efficient execution of synchronous set-based rewrite relations.

The examples provided in this paper do not exploit non-deterministic synchronous strategies. This feature is useful in symbolic reachability analysis techniques in rewriting logic for synchronous set relations [15]. The combination of these techniques with rewriting modulo SMT solving and narrowing-based techniques is a promising area of research on symbolic analysis of safety properties of synchronous set relations.

Acknowledgements

The authors would like to thank the anonymous referees for their comments. The first author was partially 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 awarded to the National Institute of Aerospace, and also by grants NSF CCF09-05584 and NSF CCF09-05584.

References


