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*

Abstract

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.

Keywords: synchronous set relations, synchronous semantics, abstract set relations, simulation, verification, rewriting logic, the Maude system, PLEXIL

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

Henceforth, it is assumed that $\mathcal{U}$ is the family of all *nonempty* finite sets over an abstract and possibly infinite set T, i.e., $\mathcal{U} \subseteq \wp(T)$, $\emptyset \notin \mathcal{U}$, and if $A \in \mathcal{U}$ then $\mathrm{card}(A) \in \mathbb{N}$ (therefore, $\rightarrow$ is a binary relation on finite sets of T). The elements of $\mathcal{U}$ 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., $a, b \rightarrow b$ denotes $\{a, b\} \rightarrow \{b\}$. Because of this abuse of notation, the symbol ',' 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 $\{d, e\}$, notation $A, B \rightarrow B, D$ denotes $\{a, b, c, d\} \rightarrow \{c, d, e\}$.

The *parallel* relation $\rightarrow^{\|}$ of $\rightarrow$ is the relation defined as the parallel closure of $\rightarrow$, i.e., the set of pairs $\langle A\,;B\rangle$ in $\mathcal{U} \times \mathcal{U}$ such that $A \rightarrow^{\|} B$ 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 $A_i \rightarrow B_i$ 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 $\rightarrow$ 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 $A \in \mathcal{U}$ into a set $s(A) \subseteq \wp(\to)$ such that if $\{\langle A_1 ; B_1\rangle, \ldots, \langle A_n ; B_n\rangle\} \in s(A)$, 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 $\langle A ; B\rangle$ in $\mathcal{U} \times \mathcal{U}$ 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 $\{\langle A_1 ; B_1\rangle, \ldots, \langle A_n ; B_n\rangle\} \in s(A)$.

Example 1. *Let T be the set of distinct elements a, b, c, d, e, and the relation $\to = \{r_1, r_2, r_3\}$, where $r_1 = \langle a, b ; b, d\rangle$, $r_2 = \langle c ; d\rangle$, and $r_3 = \langle a, c ; e\rangle$. Let s_1, s_2, and s_3 be $\to$-strategies defined for $A = \{a, b, c, d\}$ as follows.*

$$s_1(A) = \{\, \{r_2\}, \{r_3\} \,\}, \quad s_2(A) = \{\, \{r_1, r_2\} \,\}, \quad s_3(A) = \{\, \{r_1, r_2\}, \{r_3\} \,\}.$$

It holds that

$$a, b, c, d \to^{s_1} a, b, d, \qquad a, b, c, d \to^{s_1} b, d, e, \qquad a, b, c, d \to^{s_2} b, d,$$
$$a, b, c, d \to^{s_3} b, d, \qquad a, b, c, d \to^{s_3} b, d, e.$$

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

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

1. the sets $A_1, \ldots, A_n$ are nonempty pairwise disjoint subsets of A,
2. each $\langle A_i ; B_i\rangle$ is such that for any $A' \to B'$ with $A' \subseteq A$ and $A' \cap A_i \neq \emptyset$, $\langle A_i ; B_i\rangle \not\prec_A \langle A' ; B'\rangle$, and
3. if there is $A' \to B'$ with $\langle A' ; B'\rangle \notin \{\langle A_1 ; B_1\rangle, \ldots, \langle A_n ; B_n\rangle\}$ and $A' \subseteq A$, then either
   (i) there is $\langle A_j ; B_j\rangle$, for some $1 \leq j \leq n$, such that $A_j \cap A' \neq \emptyset$ or
   (ii) there is $A'' \to B''$ with $A'' \subseteq A$, $A'' \cap A' \neq \emptyset$, and $\langle A' ; B'\rangle \prec_A \langle A'' ; B''\rangle$.

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 $\langle A_i ; B_i\rangle$ that are $\prec$-maximal, and condition (3) states that any possible extension of a saturated set with a pair $\langle A' ; B'\rangle \in \to$ 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* pairs $\langle A' ; B'\rangle$ in $\prec_A$, and hence $\prec_A$-saturation exclusively depends on the ordering of the finitely many subsets of $\to \cap (\wp(A) \times \mathcal{U})$.

Example 2. *Recall the relation* $\rightarrow = \{r_1, r_2, r_3\}$ *and the set* $A = \{a, b, c, d\}$ *from Example 1. Let* $\prec_A^1$ *be such that* $r_1 \prec_A^1 r_3$*. It holds that the sets* $\{r_2\}$ *and* $\{r_3\}$ *are* $\prec_A^1$*-saturated. The set* $\{r_1, r_2\}$ *is not* $\prec_A^1$*-saturated because* r_1 *falsifies condition (2) in Definition 2 with witness* r_3*. Let* $\prec_A^2$ *be such that* $r_3 \prec_A^2 r_1$*. In this case, the only* $\prec_A^2$*-saturated set is* $\{r_1, r_2\}$*. The set* $\{r_3\}$ *is not* $\prec_A^2$*-saturated because* r_3 *falsifies condition (2) in Definition 2 with witness* r_1*. For* $\prec_A^3 = \emptyset$*, the sets* $\{r_1, r_2\}$ *and* $\{r_3\}$ *are the only* $\prec_A^3$*-saturated sets.*

A maximal strategy defines the most general synchronous behavior of a relation, which is given by *all* 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,* $\prec_A^1$*-maximal,* $\prec_A^2$*-maximal, and* $\prec_A^3$*-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 γ 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,
- $\beta = \alpha \backslash \{\langle A' ; B' \rangle \in \alpha \mid (\exists \langle A'' ; B'' \rangle \in \alpha) A' \cap A'' \neq \emptyset \wedge \langle A' ; B' \rangle \prec_A \langle A'' ; B'' \rangle\}$, i.e., β is the subset of α in which all conflicting pairs in α that are not maximal elements in $\prec_A$ have been omitted;
- $\sigma \subseteq \wp(\beta)$ is the collection of largest non-conflicting subsets of β; and
- if $C \in \sigma$, then for any nonempty $C' \subseteq (\beta \setminus C)$, $C \cup C' \notin \sigma$.

Let $D = \{\langle A_1 ; B_1 \rangle, \ldots, \langle A_n ; B_n \rangle\}$. 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 $D \in \sigma$.

($\Longrightarrow$) If D is $\prec_A$-saturated, then $D \subseteq \alpha$ follows by definition. If $D \not\subseteq \beta$, then there is $\langle A_i ; B_i \rangle \in D$ satisfying $\langle A_i ; B_i \rangle \prec_A \langle A' ; B' \rangle$ for some $\langle A' ; B' \rangle \in \alpha$ with $A' \cap A_i \neq \emptyset$. But then, for D, $\langle A_i ; B_i \rangle$ violates condition (2) in

Definition 2, a contradiction. Hence, $D \subseteq \beta$. If $D \notin \sigma$, since $D \subseteq \beta$ and the $A_1, \ldots, A_n$ are pairwise disjoint by assumption, either there is a nonempty set $D' \subseteq \beta \backslash D$ such that $D \cup D' \in \sigma$ or there is a nonempty set $D'' \subsetneq D$ such that $D'' \in \sigma$. If $D \cup D' \in \sigma$ and since D' is nonempty, any pair $\langle A' ; B' \rangle \in D'$ violates condition (3.ii) in Definition 2, contradicting the $\prec_A$-maximality of D. If $D'' \in \sigma$, then for any pair $\langle A'' ; B'' \rangle \in D \backslash D''$ the set $C = D'' \cup \{\langle A'' ; B'' \rangle\}$ falsifies the test in line 14 of Algorithm 1 and hence $C \in \sigma$. Since $D'' \in \sigma$ and $D'' \subsetneq C \in \sigma$, this contradicts the last remark aforementioned. Therefore, as desired, $D \in \sigma$.

($\Longleftarrow$) If $D \in \sigma \subseteq \wp(\alpha)$, 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 $\langle A_i ; B_i \rangle \in D$ satisfies condition (2) in Definition 2. For condition (3), assume that there is $\langle A' ; B' \rangle \in \alpha$ with $\langle A' ; B' \rangle \notin D$. Then, either $\langle A' ; B' \rangle \in (\beta \setminus D)$ or $\langle A' ; B' \rangle \in (\alpha \setminus \beta)$. If $\langle A' ; B' \rangle \in (\beta \setminus D)$, then $D \cup \{\langle A' ; B' \rangle\} \notin \sigma$, as previously stated. However, $\langle A' ; B' \rangle \in \beta$, so it must be the case that $A' \cap A_i \neq \emptyset$ for some $1 \leq i \leq n$. If $\langle A ; B' \rangle \in (\alpha \setminus \beta)$, then $\langle A' ; B' \rangle \prec_A \langle A'' ; B'' \rangle$ for some $\langle A'' ; B'' \rangle \in \alpha$. In either case, D satisfies condition (3) in Definition 2. Thus, D is $\prec_A$-saturated.

□

The definitions of strategy and maximal strategy used in this paper are more general than those in [16, §2]. In that paper, the only possible nondeterminism in $\rightarrow^s$ arises from $\rightarrow$. In the formalization presented in this paper, as illustrated by strategies s_1 and s_3, the synchronous relation $\rightarrow^s$ can be nondeterministic even when the relation $\rightarrow$ 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

Input : $A \in \mathcal{U}$ and priority $\prec$ for $\rightarrow$.
Output: $s(A)$, with s the $\prec_A$-maximal $\rightarrow$-strategy.

begin
  $\alpha, \beta, \gamma, \sigma \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$;
  for $A_i \rightarrow$*-redex,* $A_i \subseteq A$*, and* B_i *such that* $A_i \rightarrow B_i$ **do**
    add $\langle A_i \,;\, B_i \rangle$ to α;
  end
  for $\langle A_i \,;\, B_i \rangle \in \alpha$ **do**
    if $(\forall \langle A' \,;\, B' \rangle \in \alpha)\ (A_i \cap A') \neq \emptyset \implies \langle A_i \,;\, B_i \rangle \not\prec \langle A' \,;\, B' \rangle$ **then**
      add $\langle A_i \,;\, B_i \rangle$ to β;
    end
  end
  $\gamma \leftarrow \{\beta\}$;
  while $\gamma \neq \emptyset$ **do**
    remove any element C from γ;
    if $(\exists \langle A_i \,;\, B_i \rangle, \langle A_j \,;\, B_j \rangle \in C)$ *with* $i \neq j$ *and* $A_i \cap A_j \neq \emptyset$
    then add $C \setminus \{\langle A_i \,;\, B_i \rangle\}$ and $C \setminus \{\langle A_j \,;\, B_j \rangle\}$ to γ;
    else add C to σ;
  end
  return σ*;*
end

Algorithm 1: The $\prec$-maximal $\rightarrow$-strategy.

sort s is denoted by $T_{\Sigma,s}$. It is assumed that for each sort s, $T_{\Sigma,s}$ is nonempty; this assumption simplifies the proof system to follow. Algebras $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the respective term algebras. The set of variables of a term t is written $vars(t)$ and is extended to sets of terms in the natural way. A term t is called *ground* if $vars(t) = \emptyset$. A *substitution* θ is a sorted map from a finite subset $dom(\theta) \subseteq X$ to $ran(\theta) \subseteq T_\Sigma(X)$ and extends homomorphically in the natural way. Substitution θ is called *ground* if $ran(\theta)$ is ground. Expression $t\theta$ denotes the application of θ to term t.

A Σ-*equation* is a sentence $t = u$ **if** *cond*, where $t = u$ is a Σ-*equality* with $t, u \in T_\Sigma(X)_s$, for some sort $s \in S$, and the *condition cond* is a finite conjunction of Σ-equalities. An *equational theory* is a pair (Σ, E) with order-sorted signature Σ and finite set of Σ-equations E. For a Σ-equation φ, the judgement $(\Sigma, E) \vdash \varphi$ states that φ can be derived from (Σ, E) by the deduction rules in [11]. In this case, it holds that φ is valid in all models of (Σ, E). An equational theory (Σ, 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 Σ-algebras $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$ denote the quotient algebras induced by $=_E$ over the algebras $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$. The algebra $\mathcal{T}_{\Sigma/E}$ is called the *initial algebra* of (Σ, E).

A Σ-*rule* is a sentence $\flat : t \Rightarrow u$ **if** *cond*, where $\flat$ is its *name*, $t \Rightarrow u$ is a Σ-*sequent* with $t, u \in T_\Sigma(X)_s$, for some sort $s \in S$, and the *condition cond* is a finite conjunction of Σ-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 $\mathcal{E}_\mathcal{R} = (\Sigma, E)$ and a finite set of Σ-rules R. For $\mathcal{R} = (\Sigma, E, R)$ and $\flat$ a Σ-rule, the judgement $\mathcal{R} \vdash \flat$ states that $\flat$ can be derived from $\mathcal{R}$ by the deduction rules in [2]. In this case, it holds that $\flat$ is valid in all models of $\mathcal{R}$. For $\flat$ a Σ-equation, it can be proved that $\mathcal{R} \vdash \flat$ if and only if $\mathcal{E}_\mathcal{R} \vdash \flat$. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the rewrite relation $\Rightarrow_\mathcal{R}$ on $T_{\Sigma/E}(X)$ defined for every $t, u \in T_\Sigma(X)$ by $[t]_E \Rightarrow_\mathcal{R} [u]_E$ if and only if there is a *one-step* rewrite proof $\mathcal{R} \vdash t \Rightarrow u$. Relations $\Rightarrow_\mathcal{R}$ and $\Rightarrow^*_\mathcal{R}$ respectively denote a one-step rewrite and an arbitrary length (but finite) rewrite in $\mathcal{R}$ from t to u. Model $\mathcal{T}_\mathcal{R} = (\mathcal{T}_{\Sigma/E}, \Rightarrow^*_\mathcal{R})$ is the *initial reachability model* of $\mathcal{R} = (\Sigma, E, R)$ [2].

3.2. Reflection in Rewriting Logic

A reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. Maude's language design and implementation make systematic use of the fact that order-sorted rewriting logic is reflective [5].

Order-sorted rewriting logic being reflective precisely means that there is a finitely represented order-sorted rewrite theory $\mathcal{U}$ that is *universal* in the sense that any finitely represented order-sorted rewrite theory $\mathcal{R}$ (including $\mathcal{U}$) can be represented in $\mathcal{U}$ as a term $\overline{\mathcal{R}}$. Also, any terms t, u in $\mathcal{R}$ and any pair $(\mathcal{R}, t)$ can be represented as terms $\overline{t}, \overline{u}$ and $(\overline{\mathcal{R}}, \overline{t})$ in $\mathcal{U}$, respectively, in such a way that

the following equivalence holds:

$$\mathcal{R} \vdash t \Rightarrow u \quad \equiv \quad \mathcal{U} \vdash (\overline{R}, \overline{t}) \Rightarrow (\overline{R}, \overline{u}).$$

3.3. Linear Temporal Logic Semantics of Rewrite Theories

In general, a Kripke structure can be associated with the initial reachability model $\mathcal{T}_\mathcal{R}$ of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ by making explicit the intended sort *State* of states in the signature Σ and the relevant set Φ of atomic predicates on states. The set of atomic propositions Φ is defined by an equational theory $\mathcal{E}_\Phi = (\Sigma_\Phi, E \cup E_\Phi)$. Signature Σ_Φ contains Σ and a sort $\mathbb{B}$ with constant symbols $\bot$ and $\top$ of sort $\mathbb{B}$, predicate symbols $\phi : State \to \mathbb{B}$ for each $\phi \in \Phi$, and optionally some auxiliary function symbols. Equations in E_Φ define the predicate symbols in Σ_Φ and auxiliary function symbols, if any, including the Boolean operations on the sort $\mathbb{B}$. For $\phi \in \Phi$ and a ground term $t \in T_{\Sigma,State}$, the *semantics* of ϕ in $\mathcal{T}_\mathcal{R}$ is defined by $\mathcal{E}_\Phi$ as follows: $\phi(t)$ *holds* in $\mathcal{T}_\mathcal{R}$ if and only if $\mathcal{E}_\Phi \vdash \phi(t) = \top$. This defines the Kripke structure

$$\mathcal{K}^\Phi_\mathcal{R} = (T_{\Sigma/E,State}, \Rightarrow_{\mathcal{R},State}, L_\Phi)$$

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

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 second condition is that the equations E' can be oriented into a set of *ground sort-decreasing*, *ground confluent*, and *ground terminating* rules $\overrightarrow{E'}$ modulo A. The expression $[\mathrm{can}_{\Sigma,E'/A}(t)]_A \in T_{\Sigma/A,s}$ will denote the E'*-canonical form* of $[t]_A$. The rules R in $\mathcal{R}$ are assumed to be *ground coherent* relative to the equations E' modulo A [20]. Intuitively, ground coherence means that any rewriting with R modulo $E \cup A$ can be equivalently achieved by adopting the strategy of first simplifying a term to canonical form with E modulo A, and then applying a rule in R modulo A.

4. Synchronous Set Relations in Rewriting Logic

This section presents the infrastructure (Σ, 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 $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ that enables the definition of concrete mathematical objects $T_{\Sigma_{\mathcal{L}},Elem}$, $\rightarrow_{\mathcal{L}}$, and $\prec_{\mathcal{L}}$ that implement T, $\rightarrow$, $\prec$, respectively. The rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ extends the rewrite theory (Σ, 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 $\rightarrow^{s}_{\mathcal{L}}$, where s is the $\prec_{\mathcal{L}}$-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 4.1.1.

`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 *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 $T_{\Sigma_{\mathcal{L}},Elem}$.

The set of ground terms $T_{\Sigma_{\mathcal{L}},Elem}$ of the rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ implements the abstract set T of Section 2. The sort *Elem* represents *elements* in Σ having the form

$$\langle m \mid a_1 : e_1, \ldots, a_n : e_n \rangle,$$

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

The set $\mathcal{U}$ of Section 2 corresponds to the set of ground terms $T_{\Sigma_{\mathcal{L}},Ctx}$, where the sort *Ctx* represents sets of elements of sort *Elem*. A *context* is an element

of sort *Ctx*. The infrastructure (Σ, E, R) includes as part of its signature Σ the definition of the constant

$$\#CTX\# : \longrightarrow Ctx,$$

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 $\Sigma_{\mathcal{L}}$.

The sort *Val* is defined in Σ as a subsort of *Expr* and represents built-in values such as Boolean and numerical values. Function symbols

$$eval : Expr \longrightarrow Val,$$
$$eval : Ctx \times Expr \longrightarrow Val$$

are part of the signature Σ. 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 $\#CTX\#$, as explained above. This is specified in the infrastructure by the following equation in E, for any expression e of sort *Expr*:

$$eval(e) = eval(\#CTX\#, e).$$

For the binary version of *eval* no defining equations are given in E since its definition comes later, when defined by the user in $E_{\mathcal{L}}$.

The signature Σ also defines the constant

$$MODULE\text{-}NAME : \longrightarrow Qid$$

for identifying the module name that implements the language $\mathcal{L}$. The value of this constant is maintained by the user when defining the language $\mathcal{L}$. It is used by the infrastructure to obtain a representation of $\mathcal{L}$ in order to compute the synchronous relation of $\mathcal{L}$ 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 Σ in $\Sigma_{\mathcal{L}}$ 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 $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ includes a complete equational interpretation of *eval*'s binary version for the set of expressions in $\Sigma_{\mathcal{L}}$.

4.1.2. The Relation $\rightarrow_{\mathcal{L}}$.

The synchronous relation in Definition 1 is given for an abstract atomic relation $\rightarrow$. In a concrete language, such as $\mathcal{L}$, this relation represents atomic computational steps that are synchronously executed. For that reason, the concrete relation $\rightarrow_{\mathcal{L}}$ 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 Γ of sort *Ctx* will be denoted $\overset{\Gamma}{\rightarrow}_{\mathcal{L}}$.

The atomic relation $\rightarrow_{\mathcal{L}}$ is specified in $R_{\mathcal{L}}$ through *atomic rules*.

Definition 4 (Atomic Rules). Let $\Sigma_{\mathcal{L}}$ be an order-sorted signature extending Σ. An *atomic* $\Sigma_{\mathcal{L}}$*-rule* is a $\Sigma_{\mathcal{L}}$-rule $\flat : l \Rightarrow r$ **if** *cond* such that:

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

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

Definition 5 (Atomic Relation $\rightarrow_{\mathcal{L}}$). Let $\mathcal{L} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ be a rewrite theory with $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ extending (Σ, E) and $R_{\mathcal{L}}$ a collection of atomic $\Sigma_{\mathcal{L}}$-rules with different names. For a rule $\flat : l \Rightarrow r$ **if** $cond \in R_{\mathcal{L}}$, the (parametric) relation $\rightarrow_{\flat}^{\Gamma}$, with parameter $\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}$, denotes the set of pairs $\langle A\,;B \rangle$ in $T_{\Sigma_{\mathcal{L}},Ctx} \times T_{\Sigma_{\mathcal{L}},Ctx}$ such that there is a ground substitution $\theta : T_{\Sigma_{\mathcal{L}}}(X) \longrightarrow T_{\Sigma_{\mathcal{L}}}$ satisfying $cond\theta$, $(A, A') = l\theta$, and $B = r\theta$ in $\mathcal{L}$, where any expression is evaluated in Γ and any element in A is an element in B (but not necessarily with the same attribute values). The *atomic relation* $\rightarrow_{\mathcal{L}}$ is the indexed set $\{\rightarrow_{\flat}^{\Gamma}\}_{\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}, \flat \in R_{\mathcal{L}}}$.

In Definition 5, A, A', B, and Γ 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 (Σ, 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 $cond\theta$, $(A, A') = l\theta$, and $B = r\theta$ can be checked in $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ because they

are equational expressions that, although may depend on context Γ, do *not* depend on $R_{\mathcal{L}}$. In this case, since A, A', B, and Γ are ground, and θ is a ground substitution, equational deduction with $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ in Definition 5 coincides with validity in the initial model $\mathcal{T}_{\Sigma_{\mathcal{L}}/E_{\mathcal{L}}}$.

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

4.1.3. The Priority $\prec_{\mathcal{L}}$.

For a given context Γ, the elements in $\overset{\Gamma}{\rightarrow}_{\mathcal{L}}$ can be regarded as tuples of the form $(A, B, c, m)_\Gamma$ as a shorthand for $A \overset{\Gamma}{\rightarrow}_{c\text{–}m} B$, with $c\text{–}m \in R_{\mathcal{L}}$. The set $\prec_{\mathcal{L}} = \{\prec_{\mathcal{L}(\Gamma)}\}_{\Gamma \in T_{\Sigma_{\mathcal{L}}, Ctx}}$ is defined automatically by the infrastructure as

$$(A', B', c', m')_\Gamma \prec_{\mathcal{L}(\Gamma)} (A, B, c, m)_\Gamma \quad \equiv \quad A \subseteq \Gamma \wedge A' \subseteq \Gamma \wedge c = c' \wedge m < m',$$

where $<$ is the usual order on natural numbers.

Theorem 2. *The indexed set $\prec_{\mathcal{L}}$ is a priority for $\rightarrow_{\mathcal{L}}$.*

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

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

4.1.4. Implementing Algorithm 1

The rewrite theory (Σ, E, R) includes the function

$$\textit{max-strat} : \textit{Module} \times \textit{Ctx} \longrightarrow \textit{TransitionSet}$$

that computes the $\prec_{\mathcal{L}}$-maximal $\overset{\Gamma}{\rightarrow}_{\mathcal{L}}$-strategy, where $\Gamma \in T_{\Sigma_{\mathcal{L}}, Ctx}$ is the parameter of the relation $\rightarrow_{\mathcal{L}}$. 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 $\overline{\mathcal{L}}$ of language $\mathcal{L}$ and ground context Γ, and returns the collection $s(\Gamma)$, where s is the $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy. The sort *Module* is used in Maude for terms denoting meta-representations of modules and it is part of Maude's standard prelude. The sort *Transition* denotes sets of pairs in $T_{\Sigma_{\mathcal{L}}}(X)_{Ctx}$ and sort *TransitionSet* denotes collections of transitions with multiset union operator ','.

The function *max-strat*$(\overline{\mathcal{L}}, \Gamma)$ reduces the problem of computing the $\prec_{\mathcal{L}}$-maximal $\rightarrow^{\Gamma}_{\mathcal{L}}$-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 Γ, obtaining a module $\mathcal{L}'$ meta-represented by $\overline{\mathcal{L}'}$. Second, it computes the one-step atomic relation $\rightarrow_{\mathcal{L}}^{\Gamma}$ on Γ by using $\overline{\mathcal{L}'}$. Third, it uses the strategy $\prec_{\mathcal{L}}$ 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.

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

$$\textit{set-state} : \textit{Module} \times \textit{Ctx} \longrightarrow \textit{Module}$$

on input $\overline{\mathcal{L}}$ of sort *Module* and Γ of sort *Ctx* adds the equation

$$\#CTX\# = \Gamma$$

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

Computing the atomic relation $\rightarrow_{\mathcal{L}}$. The function

$$\textit{compute-atomic} : \textit{Module} \times \textit{Ctx} \longrightarrow \textit{LabeledTransition}$$

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

$$c\text{-}m : l \Rightarrow r \ \textbf{if} \ cond \in R_{\mathcal{L}}$$

that satisfies the conditions in Definition 5. The task of computing the ground substitutions θ in Definition 5 is achieved by using Maude's *metaXmatch* meta-level function for each rule in $\mathcal{R}_{\mathcal{L}}$. Also, each tuple $(A, B, c, m)_{\Gamma}$ 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 viceversa.

Applying the strategy $\prec_{\mathcal{L}}$. The function

$$\textit{apply-strategy} : \textit{LabeledTransition} \longrightarrow \textit{Transition}$$

on input Γ' of sort *LabeledTransition*, computed from *compute-atomic*$(\overline{\mathcal{L}'}, \Gamma)$, uses the strategy $\prec_{\mathcal{L}}$ to filter out those labeled transitions that cannot be part of any $\prec_{\mathcal{L}}$-saturated set for Γ (see Definition 2), and returns a term of sort *Transition* consisting of those pairs $\langle A\,;B\rangle$ 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.

Computing all $\prec_{\mathcal{L}}$-saturated sets. The function

$$\mathit{compute\text{-}saturated} : \mathit{Transition} \longrightarrow \mathit{TransitionSet}$$

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 $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy as computed by *max-strat*.

4.2. Simulation of $\rightarrow^{s}_{\mathcal{L}}$

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

$$\begin{array}{lll} \mathit{sync} : \{l\} \Rightarrow \{r\} & \textbf{if} & T, S := \mathit{max\text{-}strat}(\overline{\mathcal{L}}, l) \\ & \wedge & r := \mathit{update}(l, T). \end{array}$$

This rule, along with the rules $R_{\mathcal{L}}$ 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 ':=' in both equalities, which is idiom in Maude for a *matching* equality. For instance, expression

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

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

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

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

$$\mathit{update} : \mathit{Ctx} \times \mathit{Transition} \longrightarrow \mathit{Ctx}$$

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

$$C = \{\langle A_1 \,;\, B_1 \rangle, \ldots, \langle A_n \,;\, B_n \rangle\},$$

computes the ground context

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

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 $R_{\mathcal{L}}$. The actual application of those rules is done by the function *update*.

Rule *sync* is *nondeterministic* 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 $\mathcal{L} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ be an extension of (Σ, E, R). For $A, B \in T_{\Sigma_{\mathcal{L}}, Ctx}$, the following equivalence holds:*

$$\mathcal{L} \vdash \{A\} \Rightarrow \{B\} \qquad \equiv \qquad A \rightarrow^{s}_{\mathcal{L}} B,$$

where s denotes the $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy as computed by max-strat.

Proof. The key observation is that because *max-strat* computes the $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy s, the following equivalence holds:

$$C \in s(A) \qquad \equiv \qquad (\exists C' \in T_{\Sigma_{\mathcal{L}}, TransitionSet}) C, C' =_{E_{\mathcal{L}}} \textit{max-strat}(\mathcal{L}, A).$$

($\Longrightarrow$) Since $\{A\}$ can be rewritten only by rule *sync* $\in R$, there is a ground substitution $\theta : X \longrightarrow T_{\Sigma_{\mathcal{L}}}$ satisfying $A =_{E_{\mathcal{L}}} l\theta$, $B =_{E_{\mathcal{L}}} r\theta$, $T\theta, S\theta =_{E_{\mathcal{L}}}$ *max-strat*$(\mathcal{L}, l\theta)$, and $r\theta =_{E_{\mathcal{L}}}$ *update*$(l\theta, T\theta)$. By the observation above, $T\theta \in s(A)$. Then, from the definition of *update*, it follows that $A \rightarrow^{s}_{\mathcal{L}} B$.

($\Longleftarrow$) If $A \rightarrow^{s}_{\mathcal{L}} B$, there is $C = \{\langle A_1 \,;\, B_1\rangle, \ldots, \langle A_n \,;\, B_n\rangle\} \in s(A)$ 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 $C' \in T_{\Sigma_{\mathcal{L}}, TransitionSet}$ such that $C, C' =_{E_{\mathcal{L}}}$ *max-strat*$(\mathcal{L}, A)$ and $B =_{E_{\mathcal{L}}}$ *update*(A, C). Then substitution θ satisfying $A =_{E_{\mathcal{L}}} l\theta$ witnesses $\mathcal{L} \vdash \{A\} \Rightarrow \{B\}$.

□

One key advantage of this approach is that, while it offers support for the execution of a synchronous relation $\rightarrow^{s}_{\mathcal{L}}$, it does that by simulating $\rightarrow^{s}_{\mathcal{L}}$ using the standard asynchronous semantics of Maude. Therefore, all commands available in Maude for executing and verifying rewrite relations are directly available for $\rightarrow^{s}_{\mathcal{L}}$. 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 (Σ, 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 $\mathcal{S}$ 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 $\mathcal{S}$ requires the definition of an evaluation function *eval* that takes as inputs a context Γ, which is a set of elements T, and an arithmetic expression e. It is inductively defined on expressions:

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

The (parametric) atomic relation $\rightarrow_{\mathcal{S}}$ of the language $\mathcal{S}$ is defined for a context Γ by $A \rightarrow_{\mathcal{S}}^{\Gamma} B$ if and only if $A \subseteq \Gamma$, $A = \{Mem(m, v), m{:=}e\}$, $B = \{Mem(m, u), m{:=}e\}$, and $u = eval(A, e)$, for some memory name m, values v and u, and expression e. The semantic relation of the language is the relation $\rightarrow_{\mathcal{S}}^{\Gamma,s}$ (or $\rightarrow_{\mathcal{S}}^{s}$), where s is the $\prec$-maximal $\rightarrow^{\Gamma}$-strategy, Γ is a ground context, and $\prec_{\mathcal{S}}$ is the empty priority.

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

$$Mem(x, 3), Mem(y, 4), x{:=}y, y{:=}x \ \rightarrow_{\mathcal{S}}^{\Gamma,s} \ Mem(x, 4), Mem(y, 3), x{:=}y, y{:=}x.$$

Language $\mathcal{S}$ is specified by the Maude system module *SIMPLE*, which includes system module *SMAUDE*, and has the following syntax:

```
mod SIMPLE is
  including SMAUDE .

  eq MODULE-NAME = 'SIMPLE .
  ...
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*:

```
  op a    : Nat -> Eid [ctor] .
  ops x y : -> 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*:

```
ops mem body to : -> Aid [ctor] .
```

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

```
< x | mem : v >   < a(1) | 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 *SIMPLE* with the following subsort and operation declarations:

```
subsort Nat < Val .
op _+_ : Expr Expr -> Expr [ditto] .
```

Expressions are evaluated equationally by following the definition of *eval*:

```
var  C  : Ctx .    vars I J  : Eid .  vars E E' : Expr .
var  M  : Map .    var  N    : Nat .

eq eval(C,N) = N .
eq eval(( < I | mem : N , M > C), I ) = N .
eq eval(C,E + E') = eval(C,E) + eval(C,E') .
```

Atomic rule r–1 specifies the atomic relation of the language:

```
rl [r-1] :
   < I | mem : N > < J | body : E, to : I >
   =>  < I | mem : eval(E) > .
```

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

The context Γ in Example 4, written in the syntax of *SIMPLE*, is

```
< x | mem : 3 >  < y | mem : 4 >
< a(1) | body : y, to : x >  < a(2) | body : x, to : y > .
```

Maude's *search* command can be used to compute, for instance, the one-step synchronous semantic relation of the language in Example 4 from context Γ:

```
Maude> search { Γ } =>1 X:Sys .
search in SIMPLE : { Γ } =>1 X:Sys .
Solution 1 (state 1)
states: 2  rewrites: 514 in 53ms cpu (54ms real) (9655 rewrites/second)
X:Sys --> {< x | mem : 4 > < y | mem : 3 >
            < a(1) | body : y, to : x > < a(2) | body : x, to : y > }
No more solutions.
```

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 4. The overview of PLEXIL semantics and the running example of this section are borrowed from [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 they 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 Figure 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.

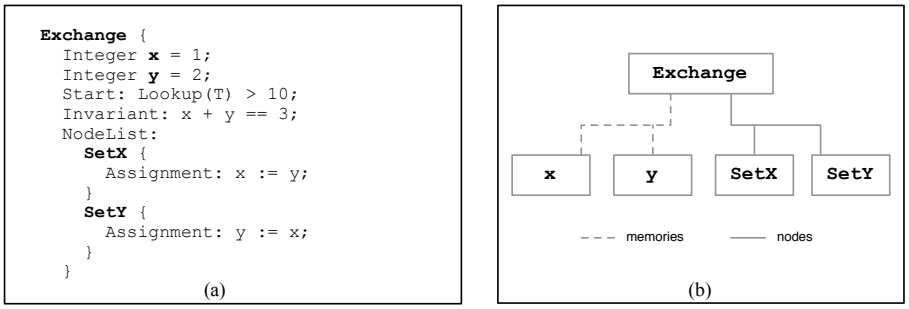

Figure 1: A PLEXIL plan.

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, Figure 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.

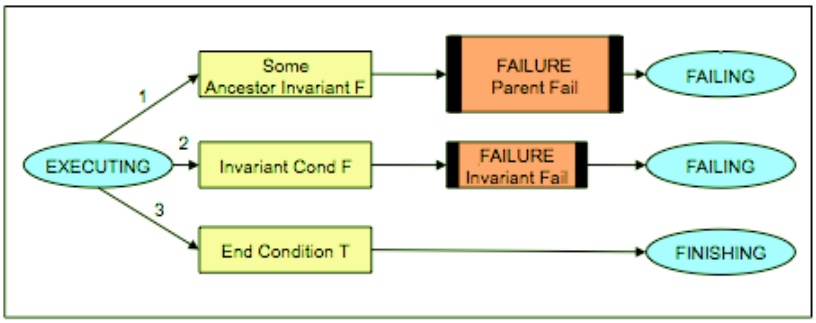

Figure 2: Atomic transitions for *list* nodes with *executing* status.

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 (Σ, 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 $T_{\Sigma_{\mathcal{L}},Elem}$

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

$$\langle N \mid a_1 : e_1, \ldots, a_n : e_n \rangle$$

with nonempty qualified name N of sort $NeQualName \leq Eid$ 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 Aid:

type	*status*	*outcome*	*parent*	*set*	*sort*	*val*	*init-val*	*values*
pre	*post*	*inv*	*start*	*end*	*repeat*	*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 *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 $BVal$ and integer values have sort $IVal$, both subsorts of the predefined sort *Val*. Boolean and integer constant values use constructors

$$c : Bool \to BVal$$
$$c : Int \to IVal$$

where *Bool* and *Int* are Maude's built-in Boolean and integer values. Sorts *BExpr* and *IExpr* represent Boolean and integer expressions, respectively, and are defined satisfying the following subsort relations:

$$BVal \leq BExpr \leq Expr \qquad \text{and} \qquad IVal \leq IExpr \leq Expr.$$

Memory bindings and lookups on external variables use constructors

$$bm, bl : NeQualName \to BExpr$$
$$im, il : NeQualName \to IExpr$$

Boolean predicates, with sort *BExpr*, include PLEXIL's built-in predicates. For instance, the Boolean predicate *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 $\Gamma \in X_{Ctx}$, $N \in X_{Eid}$, $i \in X_{IVal}$, $M \in X_{Map}$, and $iE_1, iE_2 \in X_{IExpr}$:

$$\begin{aligned} \mathit{eval}(\Gamma, i) &= i \\ \mathit{eval}(\Gamma, \mathit{im}(N)) &= i \textbf{ if } \langle N \mid \mathit{type} : \mathit{mem}, \mathit{sort} : \mathit{int}, \mathit{val} : i, M \rangle \in \Gamma \\ \mathit{eval}(\Gamma, \mathit{il}(N)) &= i \textbf{ if } \langle N \mid \mathit{type} : \mathit{ext}, \mathit{sort} : \mathit{int}, \mathit{val} : i, M \rangle \in \Gamma \\ \mathit{eval}(\Gamma, iE_1 + iE_2) &= \mathit{eval}(\Gamma, iE_1) + \mathit{eval}(\Gamma, iE_2). \end{aligned}$$

Function *eval* is similarly defined for all expressions considered in $\mathcal{L}$.

6.3.3. The Synchronous Relation $\rightarrow^{s}_{\mathcal{L}}$ and the Priority $\prec_{\mathcal{L}}$

The atomic rules in $R_{\mathcal{L}}$ include all transitions for *list* and *assignment* nodes. For example, the following atomic rules in $R_{\mathcal{L}}$ specify the atomic transitions for *list* nodes with status *executing* depicted in Figure 2, for $N \in X_{Eid}$, $o \in X_{Expr}$, and $bE \in X_{BExpr}$:

$$\begin{aligned} \mathit{exlist\text{-}1}: \quad & \langle N \mid \mathit{type} : \mathit{list}, \mathit{status} : \mathit{executing}, \mathit{outcome} : o \rangle \\ \Rightarrow \quad & \langle N \mid \mathit{status} : \mathit{failing}, \mathit{outcome} : \mathit{fail(parent)} \rangle \\ & \textbf{if } \mathit{eval}(\mathit{anc\text{-}inv\text{-}false}(N)) = c(\mathit{true}) \end{aligned}$$

$$\begin{aligned} \mathit{exlist\text{-}2}: \quad & \langle N \mid \mathit{type} : \mathit{list}, \mathit{status} : \mathit{executing}, \mathit{outcome} : o, \mathit{inv} : bE \rangle \\ \Rightarrow \quad & \langle N \mid \mathit{status} : \mathit{failing}, \mathit{outcome} : \mathit{fail(inv)} \rangle \\ & \textbf{if } \mathit{eval}(bE) = c(\mathit{false}) \end{aligned}$$

$$\begin{aligned} \mathit{exlist\text{-}3}: \quad & \langle N \mid \mathit{type} : \mathit{list}, \mathit{status} : \mathit{executing}, \mathit{end} : bE \rangle \\ \Rightarrow \quad & \langle N \mid \mathit{status} : \mathit{finishing} \rangle \\ & \textbf{if } \mathit{eval}(bE) = c(\mathit{true}) \end{aligned}$$

As explained in Section 4.1.1, the unary application of the function *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 Figure 2. Atomic rules in different groups have different component names, so that conflicts are not resolved among redexes 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 *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 $l, r \in X_{Ctx}$ by:

$$\begin{aligned} \mathit{tick}: \ \{l\} \Rightarrow \{r\} \quad \textbf{if} \quad & \emptyset := \mathit{max\text{-}strat}(\overline{\mathcal{L}}, l) \\ \wedge \quad & \mathit{has\text{-}future}(l) = \mathit{true} \\ \wedge \quad & r := \mathit{update\text{-}ext\text{-}vars}(l) \end{aligned}$$

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

6.4.1. Simulation and Debugging

Maude's *rew* 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 posible 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 context Γ in $\mathcal{L}$ representing the initial configuration of plan *Exchange* in Figure 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 $\{\Gamma\}$, 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 >
  < SetX . Exchange | type : assg, status : finished, outcome : success,
      pre : c(true), post : c(true), inv : c(true), start : c(true), end : c(true),
      repeat : c(false), skip : c(false), parent : Exchange,
      set(x . Exchange) : im(y . Exchange) >
  < SetY . Exchange | type : assg, status : finished, outcome : success,
      pre : c(true), post : c(true), inv : c(true), start : c(true), end : c(true),
      repeat : c(false), skip : c(false), parent : Exchange,
      set(y . Exchange) : im(x . Exchange) >
  < x . Exchange | type : mem, sort : int, init-val : c(1), val : c(2), parent : Exchange >
  < y . Exchange | type : mem, sort : int, init-val : c(2), val : c(1), 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

$$(sync)!$$

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

$$(sync)!\ ;\ tick\ ;\ (sync)!$$

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 $\mathcal{T}_{\mathcal{L}}$ of $\mathcal{L} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ by making explicit the intended sort of states in the signature $\Sigma_{\mathcal{L}}$ and the relevant set of atomic predicates on states. LTL model checking of PLEXIL plans in $\mathcal{L}$ takes place at the level of sort *Sys*. The set of atomic propositions $\Phi_{\mathcal{N}}$ 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 Figure 3 defines the syntax of the atomic propositions $\Phi_{\mathcal{N}}$ and formulas $\mathrm{LTL}_{\mathcal{N}}$ automatically available for model checking a plan p with set of qualified names $\mathcal{N}$. The sort of PLEXIL Boolean expressions

$$\begin{aligned}
\mathit{Status} ::= {} & \mathit{inactive} \mid \mathit{waiting} \mid \mathit{executing} \mid \mathit{finishing} \mid \mathit{iterationended} \mid \\
& \mathit{failing} \mid \mathit{finished} \\
\mathit{Failure} ::= {} & \mathit{parent} \mid \mathit{invariant} \mid \mathit{pre} \mid \mathit{post} \\
\mathit{Outcome} ::= {} & \mathit{unknown} \mid \mathit{skipped} \mid \mathit{success} \mid \mathit{fail}(\mu) \\
\mathit{Cond} ::= {} & \mathit{start} \mid \mathit{end} \mid \mathit{repeat} \mid \mathit{pre} \mid \mathit{post} \mid \mathit{inv} \\
\Phi_{\mathcal{N}} ::= {} & \mathit{true} \mid \mathit{false} \mid \mathit{status}(\lambda, \sigma) \mid \mathit{outcome}(\lambda, \omega) \mid \psi(\lambda, \delta) \mid \mathit{eval\text{-}exp}(\delta) \\
\mathit{LTL}_{\mathcal{N}} ::= {} & \alpha \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \varphi \Rightarrow \varphi' \mid \\
& \Box\varphi \mid \Diamond\varphi \mid \bigcirc\varphi \mid \varphi \mathbf{U} \varphi' \mid \varphi \mathbf{W} \varphi' \mid \varphi \mathbf{R} \varphi'
\end{aligned}$$

with variables

$$\mu : \mathit{Failure} \qquad \lambda : \mathcal{N} \qquad \sigma : \mathit{Status} \qquad \omega : \mathit{Outcome}$$
$$\psi : \mathit{Cond} \qquad \delta : \mathit{BExpr}_{\mathcal{N}} \qquad \alpha : \Phi_{\mathcal{N}} \qquad \varphi, \varphi' : \mathit{LTL}_{\mathcal{N}}$$

Figure 3: Parameterized atomic predicates $\Phi_{\mathcal{N}}$ and LTL formulas $\mathit{LTL}_{\mathcal{N}}$ in $\mathcal{L}$.

parameterized by $\mathcal{N}$ is denoted by $\mathit{BExpr}_{\mathcal{N}}$. Atomic propositions $\Phi_{\mathcal{N}}$ 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 $\mathrm{LTL}_{\mathcal{N}}$

include the usual Boolean connectives, and the temporal connectives 'always' ($\Box$), '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 Γ, and a $\text{LTL}_{\mathcal{N}}$ formula φ over the names $\mathcal{N}$ in p, Maude's LTL model checker can be used to check

$$\mathcal{K}^{\Phi_{\mathcal{N}}}_{\mathcal{L}(p)}, \{\Gamma\} \models \varphi,$$

where $\mathcal{K}^{\Phi_{\mathcal{N}}}_{\mathcal{L}(p)}$ is the Kripke structure associated with $\mathcal{T}_{\mathcal{L}(p)}$, having set of states $T_{\Sigma_{\mathcal{L}}(p)/E_{\mathcal{L}}(p),Sys}$, transition relation $\rightarrow^{s}_{\mathcal{L}(p)}$, and labeling function $L_{\Phi_{\mathcal{N}}}$.

For example, the formula

$$\Box\, inv(Exchange, true) \ \wedge \ \Diamond\, status(Exchange, finished)$$

for the plan *Exchange* in Figure 1, tests the invariant of node *Exchange* and that it will eventually transition to state *finished*. For example, for ground context Γ in $\mathcal{L}$ representing the initial configuration of plan *Exchange* in Figure 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({Γ}, []inv(Exchange,c(true))∧<>status(Exchange,finished)) .
reduce in EXCHANGE : modelCheck({Γ}, []inv(Exchange,c(true))∧<>status(Exchange,finished)) .
rewrites: 7368 in 581ms cpu (585ms real) (12661 rewrites/second)
result Bool: true
```

6.5. Comparing $\mathcal{L}$ 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 $\mathcal{L}$ developed in this paper. Henceforth, the rewriting logic semantics in [7] is denoted by $\mathcal{P}$.

Specifications $\mathcal{L}$ and $\mathcal{P}$ comprise 600 and 2500 lines of Maude code, respectively. Approximately, 2100 lines of code in $\mathcal{P}$ account for the functionality present in $\mathcal{L}$. Table 1 presents a size comparison between $\mathcal{L}$ and $\mathcal{P}$ 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 $\mathcal{L}$ in Table 1 does not include 948 lines of code that implement the generic infrastructure described in Section 4.

In terms of efficiency, $\mathcal{P}$ outperforms $\mathcal{L}$ on average by one order of magnitude. This is to be expected because, although metalevel 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 metalevel adds overhead to the overall computation of $\mathcal{L}$'s atomic relation $\rightarrow_{\mathcal{L}}$. Also, the computation of the maximal redexes strategy for $\rightarrow_{\mathcal{L}}$ is essentially exponential in the

	$\mathcal{L}$	$\mathcal{P}$
Datatypes	110	450
Function *eval*	270	282
Atomic relation	155	408
Executing lists (Figure 2)	14	29
Total	549	1169

Table 1: Size comparison between $\mathcal{L}$ and $\mathcal{P}$ for functionality common to both specifications.

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 (Σ, 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 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 $\mathbb{K}$ with resource sharing semantics that enables some kind of synchronous rewriting. J. Meseguer and P. Ölveczky [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 $\mathbb{K}$ 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.

References

[1] M. AlTurki and J. Meseguer. Reduction semantics and formal analysis of Orc programs. *Electronic Notes in Theoretical Computer Science*, 200(3):25 – 41, 2008. Proceedings of the 3rd International Workshop on Automated Specification and Verification of Web Systems (WWV 2007).

[2] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[3] C. Chira, T. F. Serbanuta, and G. Stefanescu. P systems with control nuclei: The concept. *Journal of Logic and Algebraic Programming*, 79(6):326 – 333, 2010. Membrane computing and programming.

[4] 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 *Lecture Notes in Computer Science*. Springer, 2007.

[5] 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.

[6] G. Dowek, C. Muñoz, and C. Păsăreanu. A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA, 2008.

[7] G. Dowek, C. Muñoz, and C. Rocha. Rewriting logic semantics of a plan execution language. *Electronic Proceedings in Theoretical Computer Science*, 18:77–91, 2010.

[8] T. Estlin, A. Jónsson, C. Păsăreanu, R. Simmons, K. Tso, and V. Verna. Plan Execution Interchange Language (PLEXIL). Technical Memorandum TM-2006-213483, NASA, 2006.

[9] D. Lucanu. Strategy-based rewrite semantics for membrane systems preserves maximal concurrency of evolution rule actions. *Electronic Notes in Theoretical Computer Science*, 237:107 – 125, 2009. Proceedings of the 8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008).

[10] N. Martí-Oliet, J. Meseguer, and A. Verdejo. A rewriting semantics for Maude strategies. *Electronic Notes in Theoretical Computer Science*, 238(3):227–247, 2009.

[11] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[12] J. Meseguer and P. Ölveczky. Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In J. Dong and H. Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 303–320. Springer Berlin / Heidelberg, 2010.

[13] C. Rocha. *Symbolic Reachability Analysis for Rewrite Theories*. PhD thesis, University of Illinois at Urbana-Champaign, December 2012. `https://www.ideals.illinois.edu/handle/2142/42200`.

[14] C. Rocha, H. Cadavid, C. Muñoz, and R. Siminiceanu. A formal interactive verification environment for the plan execution interchange language. In D. Latella and H. Treharne, editors, *Proceedings of 9th International Conference on Integrated Formal Methods (iFM 2012)*, volume 7321 of *Lecture Notes in Computer Science*, pages 343–357, Pisa, Italy, June 2012.

[15] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2011.

[16] C. Rocha, C. Muñoz, and G. Dowek. A formal library of set relations and its application to synchronous languages. *Theoretical Computer Science*, 412(37):4853–4866, 2011.

[17] T. Serbanuta. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois at Urbana-Champaign, December 2010. `https://www.ideals.illinois.edu/handle/2142/18252`.

[18] T. Serbanuta, G. Stefanescu, and G. Rosu. Defining and executing P systems with structured data in K. In D. Corne, P. Frisco, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer Berlin / Heidelberg, 2009.

[19] Universities Space Research Association. The Plan Execution Interchange Language, Jun 2006. `http://plexil.sourceforge.net`.

[20] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.