NASA/TM-2009-215770

Rewriting Logic Semantics of a Plan Execution Language

Gilles Dowek
Ecole Polytechnique, France

César A. Muñoz
Langley Research Center, Hampton, Virginia

Camilo Rocha
University of Illinois at Urbana-Champaign, Illinois

June 2009

NASA/TM-2009-215770

Rewriting Logic Semantics of a Plan Execution Language

Gilles Dowek
Ecole Polytechnique, France

César A. Muñoz
Langley Research Center, Hampton, Virginia

Camilo Rocha
University of Illinois at Urbana-Champaign, Illinois

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

June 2009

Acknowledgments

This work was supported by the National Aeronautics and Space Administration at Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace. The authors would like to thank the members of the NASA's Automation for Operation (A4O) project and, specially, the PLEXIL development team led by Michael Dalal at NASA Ames for their technical support.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

The *Plan Execution Interchange Language* (PLEXIL) is a synchronous language developed by NASA to support autonomous spacecraft operations. In this paper, we propose a rewriting logic semantics of PLEXIL in Maude, a high-performance logical engine. The rewriting logic semantics is by itself a formal interpreter of the language and can be used as a semantic benchmark for the implementation of PLEXIL executives. The implementation in Maude has the additional benefit of making available to PLEXIL designers and developers all the formal analysis and verification tools provided by Maude. The formalization of the PLEXIL semantics in rewriting logic poses an interesting challenge due to the synchronous nature of the language and the prioritized rules defining its semantics. To overcome this difficulty, we propose a general procedure for simulating synchronous set relations in rewriting logic that is sound and, for deterministic relations, complete. We also report on the finding of two issues at the design level of the original PLEXIL semantics that were identified with the help of the executable specification in Maude.

1 Introduction

Synchronous languages were introduced in the 1980s to program *reactive systems*, i.e., systems whose behavior is determined by their continuous reaction to the environment where they are deployed. Synchronous languages are often used to program embedded applications and automatic control software. The family of synchronous languages is characterized by the *synchronous hypothesis*, which states that a reactive system is arbitrarily fast and able to react immediately in no time to stimuli from the external environment. One of the main consequences of the synchronous hypothesis is that components running in parallel are perfectly synchronized and cannot arbitrarily interleave. The implementation of a synchronous language usually requires the simulation of the synchronous semantics into an asynchronous computation model. This simulation must ensure the validity of the synchronous hypothesis in the target asynchronous model.

The *Plan Execution Interchange Language* (PLEXIL) [9] is a synchronous language developed by NASA to support autonomous spacecraft operations. Space mission operations require flexible, efficient and reliable plan execution. The computer system on board the spacecraft that executes plans is called the *executive* and it is a safety-critical component of the space mission. The *Universal Executive* (UE) [20] is an open source PLEXIL executive developed by NASA[1]. PLEXIL and the UE have been used on mid-size applications such as robotic rovers and a prototype of a Mars drill, and to demonstrate automation for the International Space Station.

Given the critical nature of spacecraft operations, PLEXIL's operational semantics has been formally defined [8] and several properties of the language, such as determinism and compositionality, have been mechanically verified [7] in the Prototype Verification System (PVS) [13]. The formal small-step semantics is defined

[1] `http://plexil.sourceforge.net`.

using a compositional layer of five reduction relations on sets of *nodes*. These nodes are the building blocks of a PLEXIL plan and represent the hierarchical decomposition of tasks. The *atomic relation* describes the execution of an individual node in terms of state transitions triggered by changes in the environment. The *micro relation* describes the *synchronous* reduction of the atomic relation with respect to the *maximal redexes strategy*, i.e., the synchronous application of the atomic relation to the maximal set of nodes of a plan. The remaining three relations are the *quiescence relation*, the *macro relation* and the *execution relation* which describe the reduction of the micro relation until normalization, the interaction of a plan with the external environment, and the n-iteration of the macro relation corresponding to n time-steps, respectively. From an operational point of view, PLEXIL is more complex than general purpose synchronous languages such as Esterel [2] or Lustre [4]. PLEXIL is designed specifically for flexible and reliable command execution in autonomy applications.

In this paper, we propose a rewriting logic semantics of PLEXIL in Maude [5] that complements the small-step structural operational semantics written in PVS. In contrast to the PVS higher-order logic specification, the rewriting logic semantics of PLEXIL is executable. Indeed, the Maude specification is by itself an interpreter of the language. This interpreter is intended to be a semantic benchmark for validating the implementation of PLEXIL executives such as the Universal Executive and a testbed for designers of the language to study new features or possible variants of the language. Additionally, by using a graphical interface [15], PLEXIL developers will be able to exploit the formal analysis tools provided by Maude to verify properties of actual plans.

Rewriting logic is a logic of concurrent change in which a wide range of models of computation and logics can be faithfully represented. The rewriting semantics of a synchronous language such as PLEXIL poses interesting practical challenges because Maude implements the maximal concurrency of rewrite rules by interleaving concurrency. That is, although rewriting logic allows for concurrent synchronous specifications at the mathematical level, Maude executes the rewrite rules by interleaving concurrency. To overcome this problem, we develop a *serialization procedure* that allows for the simulation of a synchronous relations via set rewriting systems. This procedure is presented in a library of abstract set relations that we have written in PVS. The procedure is sound and complete for the *synchronous closure* of any deterministic relation under the *maximal redexes strategy*.

We have used the rewriting logic semantics of PLEXIL to validate the semantics of PLEXIL against a wide variety of plan examples. We report on the findings of two issues of PLEXIL's original semantics that were discovered with the help of the rewriting logic semantics of PLEXIL presented in this paper: the first was found at the level of the atomic relation for which undesired interleaving semantics were introduced in some computations, and the second was found at the level of the micro relation for which spurious infinite loops were present in some computations. Solutions to both issues were provided by the authors, and have been adopted in the latest version of PLEXIL semantics.

Summarizing, the contributions presented in this paper are:

- The rewriting logic specification of PLEXIL semantics.
- A library of abstract set relations suitable for the definition and verification of synchronous relations.
- A serialization procedure for the simulation of synchronous relations by rewriting, and an equational version of it in rewriting logic for deterministic synchronous relations.
- The findings on two issues in the design of the original PLEXIL semantics, and the corresponding solutions that were adopted in an updated version of the language semantics.

Outline of the paper. This paper is organized as follows. Background on rewriting logic and its relation to Structural Operational Semantics are summarized in Section 2. In Section 3, we present the library of set relations, including the soundness and completeness proof of the serialization procedure. Section 4 describes the rewriting logic semantics of PLEXIL. In Section 5, we discuss preliminary results. Related work and concluding remarks are presented in Section 6.

2 Rewriting Logic and Structural Operational Semantics

Rewriting logic [11] is a general semantic framework that unifies in a natural way a wide range of models of concurrency. Language specifications can be executed in Maude, a high-performance rewriting logic implementation, and benefit from a wide set of formal analysis tools available to it, such as Maude's LTL Model Checker, for instance.

A *rewriting logic specification* or *theory* is a tuple $\mathcal{R} = (\Sigma, E \cup A, R)$ where:

- $(\Sigma, E \cup A)$ is a membership equational logic theory with Σ a signature having a set of kinds, a family of sets of operators, and a family of disjoint sets of sorts; E a set of Σ-sentences, which are universally quantified Horn clauses with atoms that are equations ($t = t'$) and memberships ($t : s$), with t, t' terms and s a sort; A a set of "structural" axioms (typically associativity and/or commutativity and/or identity) such that there exists a *matching algorithm modulo A* producing a finite number of A-matching substitutions; and
- R a set of universally quantified *conditional rewrite rules* of the form

$$(\forall X)\ r : t \longrightarrow t' \text{ if } \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_l w_l \longrightarrow w_l'$$

  where X is a set of sorted variables, r is a label, $t, t', u_i, u_i', v_j, w_j$ and w_j' are terms, and s_j are sorts. A condition of a rule is called an *equational condition* if it does not contain atoms of the form $w_l \longrightarrow w_l'$.

Intuitively, $\mathcal{R}$ specifies a *concurrent system*, whose states are elements of the initial algebra $T_{\Sigma/E \cup A}$ specified by the theory $(\Sigma, E \cup A)$ and whose *concurrent transitions* are specified by the rules R. Concurrent transitions are deduced according to the set of inference rules of rewriting logic, which are described in detail in [3] (together with a precise account of the more general forms of rewrite theories and their models). Using these inference rules, a rewrite theory $\mathcal{R}$ proves a statement of the form $(\forall X)\ t \longrightarrow t'$, written as $\mathcal{R} \vdash (\forall X)\ t \longrightarrow t'$, meaning that, in $\mathcal{R}$, the state term t can transition to the state term t' in a finite number of steps. A detailed discussion of rewriting logic as a unified model of concurrency and its inference system can be found in [11].

We have a one-step rewrite $[t]_{E \cup A} \longrightarrow_{\mathcal{R}} [t']_{E \cup A}$ in $\mathcal{R}$ iff we can find a term $u \in [t]_{E \cup A}$ such that u can be rewritten to v using some rule $r : a \longrightarrow b$ if $C \in R$ in the standard way (see [6]), denoted $u \longrightarrow_{R} v$, and we furthermore have $v \in [t']_{E \cup A}$. For arbitrary E and R, whether $[t]_{E \cup A} \longrightarrow_{\mathcal{R}} [t']_{E \cup A}$ holds is in general *undecidable*, even when the equations in E are confluent and terminating modulo A. Therefore, the most useful rewrite theories satisfy additional executability conditions under which we can reduce the relation $[t]_{E \cup A} \longrightarrow_{\mathcal{R}} [t']_{E \cup A}$ to simpler forms of rewriting just modulo A, where both equality modulo A and matching modulo A are decidable.

The first condition is that E should be *ground confluent* and *terminating* modulo A [6]. This means that in the rewrite theory $\mathcal{R}_{E/A} = (\Sigma, A, E)$, (i) all rewrite sequences terminate, that is, there are no infinite sequences of the form $[t_1]_A \longrightarrow_{\mathcal{R}_{E/A}} [t_2]_A \cdots [t_n]_A \longrightarrow_{\mathcal{R}_{E/A}} [t_{n+1}]_A \cdots$, and (ii) for each $[t]_A \in T_{\Sigma/A}$ there is a *unique* A-equivalence class $[\mathsf{can}_{E/A}(t)]_A \in T_{\Sigma/A}$ called the *E-canonical form* of $[t]_A$ modulo A such that there exists a terminating sequence of zero, one, or more steps $[t]_A \longrightarrow^{*}_{\mathcal{R}_{E/A}} [\mathsf{can}_{E/A}(t)]_A$.

The second condition is that the rules R should be *coherent* [21] relative to the equations E modulo A. This precisely means that, if we decompose the rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ into the simpler theories $\mathcal{R}_{E/A} = (\Sigma, A, E)$ and $\mathcal{R}_{R/A} = (\Sigma, A, R)$, which have decidable rewrite relations $\longrightarrow_{\mathcal{R}_{E/A}}$ and $\longrightarrow_{\mathcal{R}_{R/A}}$ because of the assumptions of A, then for each A-equivalence class $[t]_A$ such that $[t]_A \longrightarrow_{\mathcal{R}_{R/A}} [t']_A$ we can always find a corresponding rewrite $[\mathsf{can}_{E/A}(t)]_A \longrightarrow_{\mathcal{R}_{R/A}} [t'']_A$ such that $[\mathsf{can}_{E/A}(t')]_A = [\mathsf{can}_{E/A}(t'')]_A$. Intuitively, coherence means that we can always adopt the strategy of first simplifying a term to canonical form with E modulo A, and then apply a rule with R modulo A to achieve the effect of rewriting with R modulo $E \cup A$.

The conceptual distinction between equations and rules has important consequences when giving the rewriting logic semantics of a language L as a rewrite theory $\mathcal{R}_L = (\Sigma_L, E_L \cup A_L, R_L)$. Rewriting logic's *abstraction dial* [12] captures precisely this conceptual distinction. One of the key features of Structural Operational Semantics is that it provides a step-by-step formal description of a language's evaluation mechanisms [14]. Setting the level of abstraction in which the interleaving behavior of the evaluations in L is observable, corresponds to the special case in which the dial is *turned down to its minimum position* by having $E_L \cup A_L = \emptyset$. The abstraction dial can also be *turned up to its maximal position* as the special case in which $R_L = \emptyset$, thus obtaining an equational semantics of the language. In general, we can make a specification as *abstract as we want* by identifying a subset $R'_L \subseteq R_L$ such that the rewrite theory $(\Sigma_L, E_L \cup A_L \cup R'_L, R_L \setminus R'_L)$ satisfies the executability conditions aforementioned. We refer the reader to [12, 17, 19] for an in-depth presentation of the relationship between structural operational semantics and rewriting logic semantics, and the use of equations and rules to capture in rewriting logic the dynamic behavior of language semantics.

The conceptual distinction between equations and rules also has important practical consequences for *program analysis*, because it affords massive *state space reduction* which can make formal analyses such as breadth-first search and model checking enormously more efficient. Because of state-space explosion, such analyses could easily become infeasible if we were to use a specification in which all computation steps are described with rules.

3 A Rewriting Library for Synchronous Relations

When designing a programming language, it is useful to be able to define its semantic relation, to formally prove properties of this relation and to execute it on particular programs. However, defining such a semantic relation and formally reasoning about it is generally difficult, time consuming, and error-prone. This would be a major endeavor if it had to be done from scratch for each language. Moreover, since programming languages tend to evolve constantly, tools must allow reusing parts of former developments to support rapid yet correct prototyping.

Fortunately, small-step operational semantic relations are, in general, built from simple relations with a limited number of operations, such as reflexive-transitive extension, reduction to normal form, parallel extension, etc. As a minimum, the framework should include a library containing the definitions of these operations and formal proofs of their properties. This will considerably reduce the amount of work needed to define the semantic relation of *particular* programming languages and formally prove their properties. Defining the semantic relation of synchronous languages requires defining the synchronous extension of an atomic execution relation, an operation that has been much less formally studied than other relation operations such as the reflexive-transitive extension or the parallel extension.

We present in this section a first attempt to design a framework for rapid yet correct prototyping of semantic relations, in particular of synchronous languages. This framework allows for defining semantic relations, executing them on particular programs and formally proving some of their properties using general theorems about the operations that permit building relations from relations. We have been experimenting with this framework using various versions of the PLEXIL language (see Section 4).

The definitions and properties presented in Section 3.1 have been developed in PVS. The Maude engine is used for executing the semantic relations on particular programs. The full development of the framework, including the formal semantics of PLEXIL, is available from `http://research.nianet.org/fm-at-nia/PLEXIL`.

3.1 Set Relations and Determinism

Let $\rightarrow$ be a binary relation on a set T. We say that $a \in T$ is a *redex* if there exists $a' \in T$ such that $a \rightarrow a'$, and that it is a *normal form* otherwise. We denote by $\rightarrow^0$, $\rightarrow^n$, and $\rightarrow^*$, the *identity* relation, *n-fold composition*, and *reflexive-transitive closure* of $\rightarrow$, respectively.

In addition to the above relations, we also define the *normalized reduction relation* $\rightarrow^{\downarrow}$ of $\rightarrow$.

Normalized reduction $a{\rightarrow^{\downarrow}}a'$ if and only if $a{\rightarrow^{*}}a'$ and a' is a normal form.

Henceforth, we assume that the relation $\rightarrow$ is defined on sets over an abstract type T, i.e., $\rightarrow \subseteq \mathcal{P}(T) \times \mathcal{P}(T)$. We define the *asynchronous* extension of $\rightarrow$, denoted $\overset{\square}{\rightarrow}$, as the congruence closure of $\rightarrow$ and the *parallel* extension of $\rightarrow$, denoted $\overset{||}{\rightarrow}$, as the parallel closure of $\rightarrow$.

Asynchronous extension $a \overset{\square}{\rightarrow} a'$ if and only if there exist sets b and b' such that $b \subseteq a$, $b \neq \varnothing$, $b \rightarrow b'$ and $a' = (a \setminus b) \cup b'$.

Parallel extension $a \overset{||}{\rightarrow} a'$ if and only if there exist $b_1, \ldots, b_n$, nonempty, pairwise disjoint subsets of a, and sets $b'_1, \ldots, b'_n$ such that $b_i \rightarrow b'_i$ and $a' = (a \setminus \bigcup_i b_i) \cup \bigcup b'_i$.

The definition of a synchronous reduction requires the definition of a strategy that selects the redexes to be synchronously reduced.

Strategy A *strategy* is a function mapping elements $a \in \mathcal{P}(T)$ into $b_1, \ldots, b_n$, nonempty, pairwise disjoint subsets of a such that all b_i are redexes for $\rightarrow$.

Synchronous extension Let s be a strategy, $a \overset{s}{\rightarrow} a'$ if and only if there exist $b'_1, \ldots, b'_n$ such that $s(a) = \{b_1, \ldots, b_n\}$, $b_i \rightarrow b'_i$ and $a' = (a \setminus \bigcup_i b_i) \cup \bigcup_i b'_i$.

A natural way of defining strategies is via priorities. A *priority* is a function p that maps elements $a \in \mathcal{P}(T)$ into natural numbers.

Maximal redex Let $a \in \mathcal{P}(T)$ and let p be a priority function. A nonempty subset b of a is said to be a *maximal redex* of a if it is a redex, and for all nonempty subsets c of a such that c is a redex, $c \neq b$ and $c \cap b \neq \varnothing$, we have $p(b) > p(c)$. By construction, the set of maximal redexes of a set are pairwise disjoint. The *maximal redexes strategy* is the function that, given a priority function, maps elements $a \in \mathcal{P}(T)$ into the set of its maximal redexes.

In addition to the definition of the relation operators presented here, our library includes formal proofs of properties related to determinism and compositionality for abstract set relations. In this paper, we will focus on determinism as this property is fundamental to the specification of synchronous relations in rewriting logic.

Determinism A binary relation $\rightarrow$ defined on a set T is said to be *deterministic* if for all a, a' and a'' in T, $a \rightarrow a'$ and $a \rightarrow a''$ implies $a' = a''$.

Determinism is a stronger property than confluence, i.e., a deterministic relation is also confluent, but a confluent relation is not necessarily deterministic.

Proposition 3.1 (Determinism of $\to^n$, $\to^{\downarrow}$ and $\overset{s}{\to}$)**.** *If the relation $\to$ is deterministic, then so are the relations $\to^n$, $\to^{\downarrow}$, and $\overset{s}{\to}$.*

In contrast, even if the relation $\to$ is deterministic, the relations $\to^*$, $\overset{\square}{\to}$ and $\to^{\|}$ are not always deterministic.

3.2 Executing Semantic Relations

Executing the semantic relation of a programming language is desirable during the design phase of the language. In particular, it allows the designer of the features to experiment with different semantic variants of the language before implementing them.

Rewrite systems are a computational way of defining binary relations. Since our formalism is based on set relations, we consider rewrite systems on an algebra of *terms* of type T modulo associativity, commutativity, identity, and idempotence: the basic axioms for the union of sets. We denote the equality on terms of this algebra by $=_{ACUI}$. The relation $\to$ defined by a rewrite system $\mathcal{R}$ is defined as follows.

Relation defined by a rewrite system $a \to b$ if and only if there exists a rewrite rule $l \longrightarrow r$ in $\mathcal{R}$ and a substitution σ such that $a =_{ACUI} \sigma l$ and $b =_{ACUI} \sigma r$.

We remark that the previous definition uses the substitution closure of the rewrite system, rather than the more traditional definition based on the congruence closure. For example, if we consider the rewrite system

$$A(x) \quad \longrightarrow \quad B(x),$$

we have that $A(0) \to B(0)$ and $A(1) \to B(1)$. On the other hand, $A(0), A(1)$ is not a redex for $\to$.

The synchronous extension of a relation $\to$ challenges the standard asynchronous interpretation of rewrite systems. Consider again the previous example. The asynchronous extension of $\to$ defined in Section 3.1, which indeed encodes the congruence closure, relates $A(0), A(1) \overset{\square}{\to} B(0), A(1)$ and $A(0), A(1) \overset{\square}{\to} A(0), B(1)$. However, it does not relate $A(0), A(1)$ to $B(0), B(1)$, which corresponds to the parallel reduction of both $A(0)$ and $A(1)$. In this particular case, we have that $A(0), A(1) \overset{\square}{\to} B(0), A(1) \overset{\square}{\to} B(0), B(1)$.

We remark that if $a \overset{s}{\to} b$, for a strategy s, then $a \overset{\square}{\to}^* b$. However, in order to select the redexes to be reduced, we need additional machinery. In particular, we need to keep a log book of redexes that need to be reduced and redexes that have been already reduced. We propose the following procedure to implement the synchronous extension of a relation $\to$, for a strategy s, in an asynchronous rewrite engine, such as Maude.

Serialization procedure Given a term $a \in \mathcal{P}(T)$, we compute a term b as follows.

1. Reduce the pair $\langle \bigcup s(a)\ ;\ \varnothing \rangle$ to a normal form $\langle \varnothing\ ;\ a' \rangle$ using the following rewrite system:

$$\langle a_i, c\ ;\ d \rangle \quad \longrightarrow \quad \langle c\ ;\ a'_i, d \rangle,$$

where $a_i \to a'_i$.

2. The term b is defined as $a \setminus \bigcup s(a) \cup a'$.

Since a strategy is a set of redexes, and this set is finite, the procedure is well-defined, i.e., it always terminates and returns a term. However, the procedure is not necessarily deterministic.

In our previous example, we want to apply the procedure to $A(0), A(1), B(1)$ using the maximal redexes strategy (assuming that all terms have the same priority). Since $\mathtt{max}_p(\{A(0), A(1), B(1)\}) = \{A(0), A(1)\}$, we have to reduce the pair $\langle A(0), A(1)\ ;\ \varnothing \rangle$, to its normal form $\langle \varnothing\ ;\ B(0), B(1) \rangle$. Then, we compute

$$\{A(0), A(1), B(1)\}\ \setminus\ \{A(0), A(1)\}\ \cup\ \{B(0), B(1)\},$$

which is equal to $B(0), B(1)$. We check that $A(0), A(1), B(1) \xrightarrow{s} B(0), B(1)$.

Theorem 3.2 (Correctness of serialization procedure). *The serialization procedure is* sound*, i.e., if the procedure returns b from a, then $a \xrightarrow{s} b$. Furthermore, if $\to$ is deterministic, the procedure is* complete*, i.e., if $a \xrightarrow{s} b$ then the procedure returns b from a.*

Proof. **Soundness** Assume that the procedure returns $b = a \setminus \bigcup s(a) \cup a'$ from a. We have to prove that $a \xrightarrow{s} b$. Let $s(a) = \{a_1, \ldots, a_n\}$, where $a_i \subseteq a$, for $1 \leq i \leq n$. From the definition of a strategy, the elements in $s(a)$ are pairwise disjoint. Then, from the procedure, $a' = a'_1, \ldots, a'_n$, where $a_i \to a'_i$, for $1 \leq i \leq n$. Let $c \subseteq a$ be such that none of the subsets of c is in $s(a)$. Then, a has the form $a = a_1, \ldots, a_n, c$. Hence, $b = a'_1, \ldots, a'_n, c$. By definition of $\xrightarrow{s}$, we have that $a_1, \ldots, a_n, c \xrightarrow{s} a'_1, \ldots, a'_n, c$.

Completeness In this case, it suffices to note that by Proposition 3.1, if $\to$ is deterministic, then $\xrightarrow{s}$ is deterministic. Therefore, the normal form of $\langle \bigcup s(a)\ ;\ \varnothing \rangle$ is unique and the procedure returns a unique $b = a \setminus \bigcup s(a) \cup a'$. This b is the only term that is related to a in the relation $\xrightarrow{s}$.

□

4 Rewriting Logic Semantics of PLEXIL

The framework presented in Section 3 is abstract with respect to the elements in the set T and the basic set relation $\to$. If we consider that T is a set of PLEXIL nodes and $\to$ is PLEXIL's atomic relation, we can deduce by Proposition 3.1 that, since PLEXIL's atomic relation is deterministic [7] , PLEXIL's micro and quiescence relations are deterministic as well. Therefore, we can use the serialization procedure presented in Section 3.2 to implement a sound and complete formal interpreter of PLEXIL in Maude.

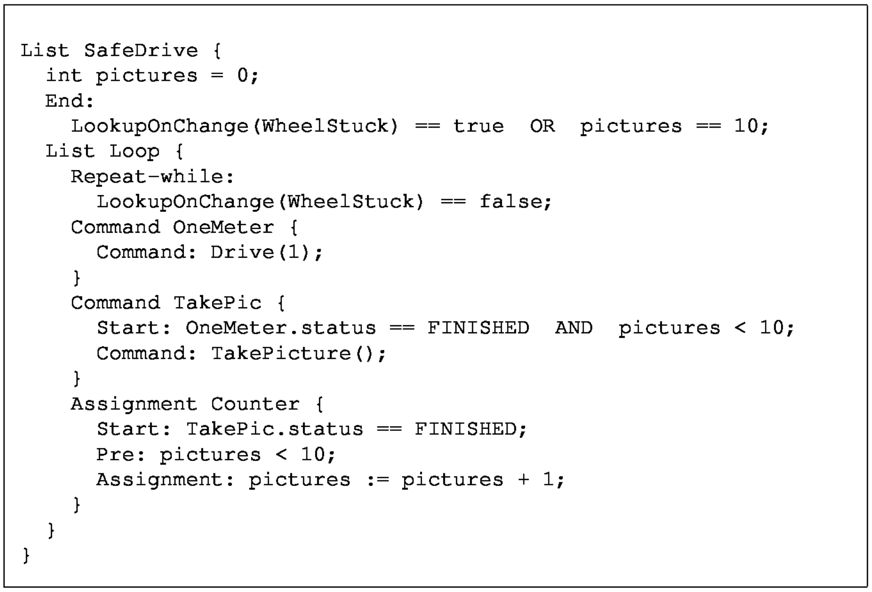
In this section, we describe in detail the specification of such an interpreter. We only discuss the atomic and micro relations since they are the most interesting for validating the synchronous semantics of PLEXIL. More precisely, we present the rewrite theory $\mathcal{R}_{\mathrm{PXL}} = (\Sigma_{\mathrm{PXL}}, E_{\mathrm{PXL}} \cup A_{\mathrm{PXL}}, R_{\mathrm{PXL}})$, specifying the rewriting logic semantics for PLEXIL's atomic and micro relations. We use the determinism property of PLEXIL's atomic relation to encode it as the computation rules in E_{PXL} because it yields a confluent equational specification. Consequently, the serialization procedure for PLEXIL's synchronous semantics into rewriting logic can be defined equationally, thus avoiding the interleaving semantics associated with rewrite rules in Maude. Of course, due to the determinism property of the language, one can as well turn up the "abstraction dial" to its maximum by making the rewrite rules R_{PXL} into computational rules. This will result in a faster interpreter, for example. Nevertheless, we are interested in PLEXIL semantics at the observable level of the micro relation. Therefore, in the rewrite theory $\mathcal{R}_{\mathrm{PXL}}$: (i) the equational theory $(\Sigma_{\mathrm{PXL}}, E_{\mathrm{PXL}} \cup A_{\mathrm{PXL}})$ defines the semantics of the atomic relation and specifies the serialization procedure for the synchronous semantics of PLEXIL, and (ii) the rewrite rules R_{PXL} define the semantics of the micro relation.

In this Section we assume the reader is familiar with the syntax of Maude [5].

4.1 PLEXIL Syntax

A PLEXIL *plan* is a tree of *nodes* representing a hierarchical decomposition of tasks. The interior nodes in a plan provide the control structure and the leaf nodes represent primitive actions. The purpose of each node determines its *type*: `List` nodes group other nodes and provide scope for local variables, `Assignment` nodes assign values to variables (they also have a *priority*, which serves to solve race conditions between assignment nodes), `Command` nodes represent calls to commands, and `Empty` nodes do nothing. Each PLEXIL node has *gate conditions* and *check conditions*. The former specify when the node should start executing, when it should finish executing, when it should be repeated, and when it should be skipped. Check conditions specify flags to detect when node execution fails due to violations of pre-conditions, post-conditions, or invariants. Declared *variables* in nodes have lexical scope, that is, they are accessible to the node and all its descendants, but not siblings or ancestors. The *execution status* of a node is given by status such as `Inactive`, `Waiting`, `Executing`, etc. The *execution state* of a plan consists of (i) the *external state* corresponding to a set of *environment variables* that are accessed through *lookups* on environment variables, and the *internal state* which is a set of nodes and (declared) variables.

Figure 1 illustrates with a simple example the standard syntax of PLEXIL. In this particular example, the plan tasks are represented by the root node `SafeDrive`, the interior node `Loop`, and the leaf nodes `OneMeter`, `TakePic` and `Counter`. `OneMeter` and `TakePic` are, for example, nodes of type `Command`. The node `Counter` has two different conditions: `Start` is a gate condition constraining the execution of the assignment to start only when the node `TakePic` is in state `Finished`, while `Pre` is



```
List SafeDrive {
  int pictures = 0;
  End:
    LookupOnChange(WheelStuck) == true  OR  pictures == 10;
  List Loop {
    Repeat-while:
      LookupOnChange(WheelStuck) == false;
    Command OneMeter {
      Command: Drive(1);
    }
    Command TakePic {
      Start: OneMeter.status == FINISHED  AND  pictures < 10;
      Command: TakePicture();
    }
    Assignment Counter {
      Start: TakePic.status == FINISHED;
      Pre: pictures < 10;
      Assignment: pictures := pictures + 1;
    }
  }
}
```

Figure 1. SafeDrive: A PLEXIL Plan Example

a check condition for the number of pictures to be less than 10. The internal state of the plan at a particular moment is represented by the set of all nodes of the plan, plus the value of the variable `pictures`, while the external state of the plan contains the (external) variable `WheelStuck`.

The external state of a plan is defined in the module `EXTERNAL-STATE-SYNTAX`. The sort **ExternalState** represents sets of elements of sort **Pair**, each of the form (*name*, *value*); we assume that the sorts **Name** and **Value**, specifying names and values, respectively, have been defined previously in the functional modules `NAME` and `VALUE`, respectively.

```
fmod EXTERNAL-STATE-SYNTAX is
 protecting Name .
 protecting Value .
 sort Pair .
 op (_,_) : Name Value -> Pair .
 sort ExternalState .
 subsort Pair < ExternalState .
 op mtstate : -> ExternalState .
 op _,_ : ExternalState ExternalState -> ExternalState [assoc comm id: mtstate] .
 eq ES:ExternalState , ES:ExternalState = ES:ExternalState .
endfm
```

The internal state of a plan is represented with the help of Maude's built-in `CONF` module supporting object based programming. The internal state has the structure of a *set* made up of objects and messages, called *configurations* in Maude, where the objects represent the nodes and (declared) variables of a plan. Therefore, we can view the infrastructure of the internal state as a configuration built up by a binary set union operator with empty syntax, i.e., juxtaposition, as $\_\_$: **Configuration** × **Configuration** ⟶ **Configuration**. The operator $\_\_$ is declared to satisfy the structural laws of associativity and commutativity and to have identity `mtconf`. Objects and messages are singleton set configurations and belong to subsorts **Object**, **Msg** < **Configuration**, so that more complex configurations are generated out of them by set union. An *object*, representing a node or a (declared) variable, in a given configuration is represented as a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's name or identifier (of sort **Oid**), C is its class (of sort **Cid**), the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*. The set of all the attribute-value pairs of an object state (of sort **Attribute**) is formed by repeated application of the binary union operator $\_,\_$ which also obeys structural laws of associativity, commutativity, and identity, i.e., the order of the attribute-value pairs of an object is immaterial. The internal state of a plan is defined in the functional module `INTERNAL-STATE-SYNTAX` by extending the sort **Configuration**; the sorts **Exp** and **Qualified**, which we assume to be defined, are used to specify expressions and qualified names, respectively.

```
fmod INTERNAL-STATE-SYNTAX is
 extending CONFIGURATION . protecting EXP .
 protecting QUALIFIED .
 subsort Qualified < Oid .    --- Qualified elements are object identifiers
 ops List Command Assignment Empty : -> Cid .   --- Types of nodes
 sort Status .
 ops Inactive Waiting Executing Finishing Failing Finished
       IterationEnded Variable : -> ExecState .
 sort Outcome .
 ops None Success Failure : -> Outcome .
 op  status: : Status -> Attribute .      --- Status of execution
 op  outcome: : Outcome -> Attribute .    --- Outcome of execution
 ops start: skip: repeat: end: : Exp -> Attribute .  --- Gate conditions
 ops pre: post: inv: : Exp -> Attribute . --- Check conditions
 op  command: : Exp -> Attribute .        --- Command of a command node
 op  assignment: : Exp -> Attribute .     --- Assignment of an assignment node
 ops initval actval: Exp -> Attribute .   --- Initial and actual values
                                          --- of a variable node
 ...
endfm
```

Using the infrastructure in `INTERNAL-STATE-SYNTAX`, the internal state of SafeDrive in Figure 1, is represented by the configuration in Figure 2. Observe that the sort **Qualified** provides qualified names by means of the operator

$$\_.\_ : \textbf{Qualified} \times \textbf{Qualified} - > \textbf{Qualified},$$

which we use to maintain the hierarchical structure of the plans. The dots at the end of each object represent the object's attributes that are not explicitly defined by the plan but that are always present in each node such as the status or the outcome. There is a "compilation procedure" from PLEXIL plans to their corresponding representation in Maude, that we do not discuss in this paper, which includes all implicit elements of a node as attributes of the object representation of the node.

We are now ready to define the sort **State** representing the execution state of the plans in the functional module `STATE-SYNTAX`, by importing the syntax of external and internal states:

```
< SafeDrive : List | end: LookupOnChange(WheelStuck) == true OR
pictures == 10,  ... >
< Loop . SafeDrive : List | repeat: LookupOnChange(WheelStuck) == false , ... >
< OneMetter . Loop . SafeDrive : Command | command: Drive(1), ... >
< TakePic . Loop . SafeDrive : Command | start: OneMeter.Status ==
Finished and pictures < 10,  command: TakePicture(), ... >
< Counter . Loop . SafeDrive : Assignment | pre: pictures < 10,
  assignment: pictures := pictures + 1, ... >
< pictures . SafeDrive : Memory | initval: 0, actval: 0 >
```

Figure 2. SafeDrive in $\mathcal{R}_{\text{PXL}}$

```
fmod STATE-SYNTAX is
 pr EXTERNAL-STATE-SYNTAX .
 pr INTERNAL-STATE-SYNTAX .
 sort State .
 op _|-_ : ExternalState Configuration -> State .
endfm
```

We adopt the syntax $\Gamma \vdash \pi$ to represent the execution state of the plans, where Γ and π are the external and internal states, respectively.

4.2 PLEXIL Semantics

PLEXIL execution is driven by external events. The set of events includes events related to lookup in conditions, e.g., changes in the value of an external state that affects a gate condition, acknowledgments that a command has been initialized, reception of a value returned by a command, etc. We focus on the execution semantics of PLEXIL specified in terms of node states and transitions between node states that are triggered by condition changes (atomic relation) and its synchronous closure under the maximal redexes strategy (micro relation).

PLEXIL's atomic relation consists of 42 rules. These rules are indexed by the type and the execution status of nodes into a dozen groups. Each group associates a priority to its set of rules which defines a linear order on the set of rules.

The *atomic relation* is defined by $(\Gamma, \pi) \vdash P \longrightarrow_a P'$, where $P \subseteq \pi$. For instance, the four atomic rules corresponding to the transitions from **Executing** for nodes of type **Assignment** are depicted in Figure 3. Rule r_3 updates the variable x to the value of the expression **e** in the given state, when the expressions associated with the gate condition **End** and the check condition **Post** of node **A**, both evaluate to **true** in the given state. In rule r_1, **AncInv(A)** is a predicate, parametric in the name of nodes, stating that none of the ancestors of **A** has changed the value associated with its invariant condition to **false**. The value $\perp$ represents the special value "Unknown". We use $(\Gamma, \pi) \vdash e \rightsquigarrow v$ to denote that expression e evaluates to value v in state $\Gamma \vdash \pi$; by abuse of notation, we write $(\Gamma, \pi) \vdash e \not\rightsquigarrow v$ to denote that expression e does not evaluate to value v in (Γ, π).

The relation $r < s$ between the labels of two different rules specifies that the rule r is only applied when the second rule s cannot be applied. That is, the binary relation on rules defines the *order of their application* when deriving atomic transitions. So, a rule r can be used to derive an atomic transition if all its premises are valid and no rule higher than r (in its group) is applicable. In the case of

$$\frac{\begin{array}{c}(\Gamma,\pi) \vdash \texttt{AncInv(A)} \rightsquigarrow \texttt{false} \qquad \texttt{A.body} \;=\; \texttt{x := e} \\ \texttt{A.type} = \texttt{Assignment} \qquad \texttt{A.status} = \texttt{Executing}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow_a \texttt{Node A with [status} = \texttt{Finished}\,,\ \texttt{x} = \bot\,,\ \texttt{outcome} = \texttt{Failure]}}\ r_1$$

$$\frac{\begin{array}{c}(\Gamma,\pi) \vdash \texttt{A.Invariant} \rightsquigarrow \texttt{false} \qquad \texttt{A.body} \;=\; \texttt{x := e} \\ \texttt{A.type} = \texttt{Assignment} \qquad \texttt{A.status} = \texttt{Executing}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow_a \texttt{Node A with [status} = \texttt{IterationEnded}\,,\ \texttt{x} = \bot\,,\ \texttt{outcome} = \texttt{Failure]}}\ r_2$$

$$\frac{\begin{array}{c}(\Gamma,\pi) \vdash \texttt{A.End} \rightsquigarrow \texttt{true} \qquad \texttt{A.body} \;=\; \texttt{x := e} \\ (\Gamma,\pi) \vdash \texttt{A.Post} \rightsquigarrow \texttt{true} \qquad (\Gamma,\pi) \vdash \texttt{e} \rightsquigarrow \texttt{w} \\ \texttt{A.type} = \texttt{Assignment} \qquad \texttt{A.status} = \texttt{Executing}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow_a \texttt{Node A with [status} = \texttt{IterationEnded}\,,\ \texttt{x} = \texttt{w}\,,\ \texttt{outcome} = \texttt{Success]}}\ r_3$$

$$\frac{\begin{array}{c}(\Gamma,\pi) \vdash \texttt{A.End} \rightsquigarrow \texttt{true} \qquad (\Gamma,\pi) \vdash \texttt{A.Post} \not\rightsquigarrow \texttt{true} \\ \texttt{A.type} = \texttt{Assignment} \qquad \texttt{A.status} = \texttt{Executing}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow_a \texttt{Node A with [status} = \texttt{IterationEnded}\,,\ \texttt{outcome} = \texttt{Failure]}}\ r_4$$

$$\{r_4 < r_3 < r_2 < r_1\}$$

Figure 3. Atomic rules corresponding to the transitions from **Executing** for nodes of type **Assignment**

PLEXIL's atomic relation, the binary relation $<$ on rules is a linear ordering. This linearity is key to the determinism of PLEXIL (see [8]).

The *micro relation* $\Gamma \vdash \pi \longrightarrow_m \pi'$, the synchronous closure of the atomic relation under the maximal redexes strategy, is defined as:

$$\frac{\begin{array}{c}(\Gamma,\pi) \vdash P_1 \longrightarrow_a Q_1 \\ \dots \\ (\Gamma,\pi) \vdash P_n \longrightarrow_a Q_n\end{array}}{\Gamma \vdash \pi \longrightarrow_m (\pi \setminus \bigcup_{1 \le i \le n} P_i) \cup \bigcup_{1 \le i \le n} Q_i}\ \textsc{Micro}$$

where $M_\pi = \{P_1, \dots, P_n\}$ is the set of node and memories in π that are affected by the micro relation. If two different processes in π, say `A` and `B`, write to the same variable, only the update of the process with higher priority is considered (assignment nodes have an associated priority always), e.g., `A` if **`A.priority > B.priority`**, `B` if **`B.priority > A.priority`**, and none otherwise.

In order to specify the PLEXIL semantics in Maude, we first define the infrastructure for the serialization procedure in the the module `SERIALIZATION-INFRASTRUCTURE`.

```
fmod SERIALIZATION-INFRASTRUCTURE is
 inc STATE-SYNTAX .
 ...
 op  [_:_|_] : Oid Cid AttributeSet -> Object .      --- New syntactic sugar for objects
 ...
 op  updateStatus    : Qualified Status  -> Msg .   --- Update status message
 op  updateOutcome   : Qualified Outcome -> Msg .   --- Update outcome message
 op  updateVariable  : Qualified Value   -> Msg .   --- Update variable message
 ...
 op  applyUpdates  : State -> State .                --- Application of updates
 var Gamma : ExternalState .  var Pi : InternalState .
 eq  applyUpdates(Gamma |- [ A:Oid : C:Cid | Att:AttributeSet status: S:Status ]
                  updateStatus(A:Oid , S':Status) Pi)
  =  applyUpdates(Gamma |- [ A:Oid : C:Cid |
   Att:AttributeSet status: S':Status ] Pi) .
 eq  applyUpdates(Gamma |- [ A:Oid : C:Cid | Att:AttributeSet ] Pi)
  =  applyUpdates(Gamma |- < A:Oid : C:Cid | Att:AttributeSet > Pi) .
 ...
 eq  applyUpdates(St:State) = St:State [owise] .
endfm
```

Following the idea of the serialization procedure, we distinguish between unprimed and primed redexes by using syntactic sugar for denoting objects in the Maude specification: unprimed redexes are identified with the already defined syntax for objects in the form of $\langle O : C \mid ... \rangle$ and primed redexes are identified with the new syntax for objects in the form of $[O : C \mid ...]$. We use messages, i.e., elements in the sort **Msg**, to denote the update actions associated with the reduction rules for the atomic relation; we accumulate these messages in the internal state of the execution state of the plans, i.e., we also use the internal state in the spirit of the log book of the serialization procedure. For example, the configuration:
`updateStatus(A,IterationEnded)`
`updateVariable(x,w)`
`updateOutcome(A,Success)`
corresponds to the update actions in the conclusion of rule r_3 in Figure 3. The function **`applyUpdates`** applies all the existing updates in the internal state and "unprimes" the "primed" nodes. In the specification above, it is shown how the status of a node is updated and how primed nodes become unprimed.

We give the equational serialization procedure in the general setting in which we consider a linear ordering on the rules.

Equational serialization procedure (with priorities) Let

$$\{r_i : (\Gamma, \pi) \vdash \texttt{Node A} \ \longrightarrow_a \ \texttt{Node A with} \ [\texttt{updates}_\texttt{i}] \ if \ \texttt{C}_\texttt{i}\}_{1 \leq i \leq n}$$

be the collection of atomic rules (in horizontal notation) defining the transition relation for nodes of type T in status S, with $r_n < \cdots < r_i < \cdots < r_1$, where $\texttt{updates}_\texttt{i}$ is the set of update actions (the order in the update actions is irrelevant) in the conclusion of r_i and $\texttt{C}_\texttt{i}$ is the set of premises of r_i. The equational serialization procedure is given by the following set of equations, in Maude notation, defining the function symbol, say, r:

```
 var Gamma : ExternalState .   var A : Oid .    var S : Status .
 var Pi : Configuration .      var T : Cid .    var Attr : AttributeSet .
 op  r : State -> State .
 eq  r( Gamma |- < A : T | status: S , Attr > Pi )
  =  if C1 == true then r( Gamma |- [ A : T | status: S , Attr ]
        messages(updates1) Pi )
     else if C2 == true
          ... else if Cn == true then r( Gamma |- [ A : T | status: S , Attr ]
                                    messages(updatesn) Pi )
                   else r( Gamma |- [ A : T | status: S , Attr ] Pi ) fi
          fi ...
     fi .
 eq  r( Gamma |- Pi ) = Gamma |- Pi [owise] .
```

where $\mathtt{messages}(\mathtt{updadates}_\mathtt{i})$ represents the message configuration associated with the update actions in the conclusion of rule r_i.

The equational serialization procedure defines a fresh function symbol, say, r : **State** $\longrightarrow$ **State**. The first equation for r tries to apply the atomic rules in the given order, by first evaluating the condition and then marking the affected node. If the condition evaluates to true, then update messages are generated. The second equation, removes the function symbol r when there aren't any more possible atomic reductions with the rules $\{r_i\}$.

The atomic relation is defined in the functional module ATOMIC-RELATION by instantiating the equational serialization procedure for each one of the twelve groups of atomic rules with a different function symbol for each one.

Finally, the micro relation is defined by the rule micro in the system module PLEXIL-RLS, which materializes the rewrite theory $\mathcal{R}_{\mathrm{PXL}}$ in Maude:

```
mod PLEXIL-RLS is
 pr ATOMIC-RELATION .
 pr SERIALIZATION-INFRASTRUCTURE .
 rl [micro] : Gamma |- Pi  => Gamma |- applyUpdates(a1(...a12( Gamma |- Pi )...)) .
endm
```

where $\mathtt{a}_1, \ldots, \mathtt{a}_{12}$ are the function symbols in ATOMIC-RELATION defining the serialization procedure for each one of the twelve groups of rules.

5 Preliminary Results

We have used $\mathcal{R}_{\mathrm{PXL}}$ to validate the semantics of PLEXIL against a wide variety of plan examples. We report on the following two issues of the original PLEXIL semantics that were discovered with the help of $\mathcal{R}_{\mathrm{PXL}}$:

1. *Non-atomicity of the atomic relation.* A prior version of the atomic rules r_3 and r_4 for Assignment nodes in state Executing, presented in Figure 3, introduced an undesired interleaving semantics for variable assignments, which invalidated the synchronous nature of the language.

2. *Spurious non-termination of plans.* Due to lack of detail in the original specification of some predicates, there were cases in which some transitions for List nodes in state IterationEnded would lead to spurious infinite loops.

Although the formal operational semantics of PLEXIL in [8] has been used to prove several properties of PLEXIL, neither one of the issues was previously found. As as matter of fact, these issues do not compromise any of the proven properties of the language. Solutions to both issues were provided by the authors, and have been adopted in the latest version of the formal PLEXIL semantics. We are currently using $\mathcal{R}_{\mathrm{PXL}}$ as the formal interpreter of *PLEXIL's Formal Interactive Visual Environment* [15] (PLEXIL5), a prototype graphical environment that enables step-by-step execution of plans for scripted sequence of external events, for further validation of the language's intended semantics.

We have also developed a variant of $\mathcal{R}_{\mathrm{PXL}}$ in which the serialization procedure was implemented with rewrite rules, instead of equations, and rewrite strategies. In general, $\mathcal{R}_{\mathrm{PXL}}$ outperforms that variant by two orders of magnitude on average, and by three orders of magnitude in some extreme cases.

The rewrite theory $\mathcal{R}_{\mathrm{PXL}}$ has approximately 1000 lines of code, of which 308 lines correspond to the module `ATOMIC-RELATION`. The rest corresponds to the syntax and infrastructure specifications.

6 Related Work and Conclusion

Rewriting logic has been used previously as a testbed for specifying and animating the semantics of synchronous languages. 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. [16] 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. Our approach is more on exploiting the determinism of a synchronous relation to tackle the problem associated with the interleaving semantics of concurrency in rewriting logic. Lucanu [10] studies the problem of the interleaving semantics of concurrency in rewriting logic for synchronous systems from the perspective of P-systems. The determinism property of the synchronous language Esterel [2] was formally proven by O. Tardieu in [18].

We have presented a rewriting logic semantics of PLEXIL, a synchronous plan execution language developed by NASA to support autonomous spacecraft operations. The rewriting logic specification relies on the determinism of PLEXIL's atomic relation and a serialization procedure that enables the specification of a synchronous relation in an asynchronous computational model. Two issues in the original design of PLEXIL were found with the help of the rewriting logic specification of the language: (i) there was an atomic rule with the potential to violate the atomicity of the atomic relation, thus voiding the synchronous nature of the language, and (ii) a set of rules introducing spurious non-terminating executions of plans. We proposed solutions to these issues that were integrated into the current semantics of the language.

Although we have focused on PLEXIL, the formal framework that we have developed is presented in a general setting of abstract set relations. In particular, we think that this framework can be applied to other deterministic synchronous languages. To the best of our knowledge there was no mechanized library of abstract set relations suitable for the definition and verification of synchronous relations; neither are there a soundness and completeness proof of a serialization procedure for the simulation of synchronous relations by rewrite systems.

To summarize, we view this work as (i) a step forward in bringing the use of formal methods closer to practice, (ii) a contribution to the modular and mechanized study of semantic relations, and (iii) yet another, but interesting contribution to the rewriting logic semantics project.

We intend to continue this work with the goal of arriving at a formal environment for the validation of PLEXIL. Such an environment would provide a rich formal tool to PLEXIL enthusiasts for the experimentation, analysis and verification of PLEXIL programs, which could then be extended towards a rewriting-based PLEXIL implementation with associated analysis tools. Part of our future work is also to investigate the modularity of the equational serialization procedure with prioritized rules.

Acknowledgments. This work was supported by the National Aeronautics and Space Administration at Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace. The authors would like to thank the members of the NASA's Automation for Operation (A4O) project and, specially, the PLEXIL development team led by Michael Dalal at NASA Ames, for their technical support.

References

1. M. AlTurki and J. Meseguer. Reduction semantics and formal analysis of Orc programs. *Electr. Notes Theor. Comput. Sci.*, 200(3):25–41, 2008.

2. G. Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.

3. R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1-3):386–414, 2006.

4. P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Symposium on Principles of Programming Languages (POPL)*, 1987.

5. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude - A High-Performance Logical Framework.* Springer LNCS Vol. 4350, 1st edition, 2007.

6. N. Dershowitz and J. P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

7. G. Dowek, C. Muñoz, and C. Păsăreanu. A formal analysis framework for PLEXIL. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*, September 2007.

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

9. 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.

10. D. Lucanu. Strategy-based rewrite semantics for membrane systems preserves maximal concurrency of evolution rule actions. *Electr. Notes Theor. Comput. Sci.*, 237:107–125, 2009.

11. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

12. J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theor. Comput. Sci.*, 373(3):213–237, 2007.

13. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

14. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

15. C. Rocha, C. Muñoz, and H. Cadavid. A graphical environment for the semantic validation of a plan execution language. In *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2009.

16. T. Serbanuta, G. Stefanescu, and G. Rosu. Defining and executing p systems with structured data in k. In David W. Corne, Pierluigi Frisco, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer, 2008.

17. Traian-Florin Serbanuta, Grigore Rosu, and José Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

18. O. Tardieu. A deterministic logical semantics for pure esterel. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.

19. Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in maude. *J. Log. Algebr. Program.*, 67(1-2):226–293, 2006.

20. V. Verna, A. Jónsson, C. Păsăreanu, and M. Latauro. Universal executive and plexil: Engine and language for robust spacecraft control and operations. In *Proceedings of the American Institute of Aeronautics and Astronautics Space Conference*, 2006.

21. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01- 06 - 2009	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE
Rewriting Logic Semantics of a Plan Execution Language

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Dowek, Gilles; Muñoz, César A.; Rocha, Camilo

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
015792.04.01.0423

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-19700

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM-2009-215770

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Plan Execution Interchange Language (PLEXIL) is a synchronous language developed by NASA to support autonomous spacecraft operations. In this paper, we propose a rewriting logic semantics of PLEXIL in Maude, a high-performance logical engine. The rewriting logic semantics is by itself a formal interpreter of the language and can be used as a semantic benchmark for the implementation of PLEXIL executives. The implementation in Maude has the additional benefit of making available to PLEXIL designers and developers all the formal analysis and verification tools provided by Maude. The formalization of the PLEXIL semantics in rewriting logic poses an interesting challenge due to the synchronous nature of the language and the prioritized rules defining its semantics. To overcome this difficulty, we propose a general procedure for simulating synchronous set relations in rewriting logic that is sound and, for deterministic relations, complete. We also report on the finding of two issues at the design level of the original PLEXIL semantics that were identified with the help of the executable specification in Maude.

15. SUBJECT TERMS
Plan execution; Semantics; Rewriting logic; Synchronous language

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	25	(443) 757-5802