A Formal Interactive Verification Environment for the Plan Execution Interchange Language

Camilo Rocha[1], Héctor Cadavid[2], César Muñoz[3], and Radu Siminiceanu[4]

[1] University of Illinois at Urbana-Champaign, Urbana IL, USA
[2] Escuela Colombiana de Ingeniería, Bogotá, Colombia
[3] NASA Langley Research Center, Hampton VA, USA
[4] National Institute of Aerospace, Hampton VA, USA

Abstract. The Plan Execution Interchange Language (PLEXIL) is an open source synchronous language developed by NASA for commanding and monitoring autonomous systems. This paper reports the development of the PLEXIL's Formal Interactive Verification Environment (PLEXIL5), a graphical interface to the formal executable semantics of PLEXIL. Among its main features, PLEXIL5 provides model checking of plans with support for formula editing and visualization of counterexamples, interactive simulation of plans at different granularity levels, and random initialization of external environment variables. The formal verification capabilities of PLEXIL5 are illustrated by means of a human-automation interaction model.

1 Introduction

Plan execution is a centerpiece of systems involving intelligent software agents such as robotics, unmanned vehicles, and habitats. The *Plan Execution Interchange Language* PLEXIL [8] is a synchronous language developed by NASA to support autonomous spacecraft operations. Programs in PLEXIL, called *plans*, specify actions to be executed by an autonomous system as part of normal spacecraft operations or as reactions to changes in the environment. The computer system on board the spacecraft that executes plans is called the *executive* and is a safety-critical component of the space mission. The PLEXIL Executive [18] is an open source executive developed by NASA (`http://plexil.sourceforge.net`). PLEXIL has been used on mid-size applications such as robotic rovers, a prototype of a Mars drill, and to demonstrate automation capabilities for potential future use on the International Space Station. A summary of PLEXIL's syntax and semantics is presented in Section 2.

Spacecraft operations require flexible, efficient, and reliable plan execution. Given its critical nature, PLEXIL's operational semantics has been formally specified in the Prototype Verification System (PVS) [5]. Moreover, key meta-theoretical properties of the language, such as determinism and compositionality, have been mechanically verified in PVS [6]. Based on this formalization, a formal executable semantics of PLEXIL has been specified in the rewriting logic engine Maude [7]. The executable semantics of PLEXIL serves as an efficient formal

interpreter of the language and, as illustrated by this paper, is at the core of the PLEXIL Formal Interactive Verification Environment (PLEXIL5).

PLEXIL5 is an interactive environment for verifying and testing PLEXIL plans and for studying new features and possible variants of the language. A proof of concept of such an environment was originally presented in [11], but that tool was mainly concerned with the semantic validation of the language. This paper reports significant progress on the evolution of this proof of concept into an environment for the validation and *formal verification* of PLEXIL plans. To emphasize these new capabilities, the word "Visual" in the original acronym became "Verification" in the new system. PLEXIL5 consists of a graphical environment developed in Java that interfaces with the rewriting logic semantics in Maude. Users are not required to have any knowledge of the Maude system to take advantage of PLEXIL5's formal analysis capabilities. An overview of some architecture and design features, software metrics, and aspects of user interaction are presented in Section 3.

The formal analysis capabilities in PLEXIL5 are based on the rewriting logic semantics of the language and the formal analysis tools available in the Maude system, such as the rewriting engine, the model checker, and the strategy language [4]. The environment supports the verification of temporal properties on PLEXIL plans. These properties can be provided by the user or automatically generated from plan annotations such as preconditions, invariants, and postconditions. PLEXIL5 provides a mechanisms for modeling the interaction of plans with the external environment. Technical details on the formal analysis capabilities, i.e., simulation, model checking, and semantic validation, offered by PLEXIL5 are given in Section 4. As a case study, Section 5 presents a formalization of a simple cruise control system in PLEXIL and illustrates how PLEXIL5 can aid in discovering and correcting errors in plans. The case study presented in this paper and more information about PLEXIL5 is available from `http://shemesh.larc.nasa.gov/people/cam/PLEXIL`.

2 PLEXIL Overview

This section presents an overview of PLEXIL, a synchronous language for automation developed by NASA. The reader is referred to [8] for a detailed description of the language.

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

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

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

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

Henceforth, the notation (Γ, π) is used to represent the execution state of a plan, where Γ is a set of external variables and their current values, and π is a set of nodes and their internal states. Formally, the semantics of PLEXIL is defined on states (Γ, π) by a compositional layer of five reduction relations [8]. The *atomic relation* describes the execution of an individual node in terms of state transitions triggered by changes in the environment. The *micro relation* describes the *synchronous* reduction of the atomic relation with respect to the *maximal redexes strategy*, i.e., the synchronous application of the atomic relation to the maximal set of nodes of a plan. The remaining three relations are the *quiescence relation*, the *macro relation*, and the *execution relation* that, respectively, describe the reduction of the micro relation until normalization, the interaction of a plan with the external environment upon one external event, and the n-iteration of the macro relation corresponding to n time steps.

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 local variables x and y. The values of these variables 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 can start executing whenever the value of an external variable T is greater than 10. The invariant condition states that at any state of execution the values of x and y add up to 3.

```
Exchange: {
    Integer x = 1;
    Integer y = 2;
    StartCondition: Lookup(T) > 10;
    Invariant: x+y == 3;
    NodeList:
        SetX: { Assignment: x = y; }
        SetY: { Assignment: y = x; }
}
```

Fig. 1. A PLEXIL plan that reads the value of an external variable T and synchronously exchanges the values of internal variables x and y.

3 PLEXIL5

PLEXIL5 is a graphical environment for the formal simulation and verification of PLEXIL plans, and the validation of the intended semantics of the language against its rewriting logic semantics. This section presents an overview of its architecture and design, including some software metrics, and aspects regarding user interaction in the environment.

3.1 Architecture and Design

Figure 2 depicts PLEXIL5's key components and their interaction. The graphical user interface has been developed in Java using the *model-view-controller* pattern and, for some views on execution states, uses third-party open-source libraries such as JGraph and JGoodies. The object oriented *model* represents the hierarchical structure of plans, their execution behavior, and the external environment. The *view* consists of several classes that present the user with views of the tree-like-structure of plans. The *controller* consists of a custom *controller-facade* class and *listener* classes using and extending the Java framework.

PLEXIL5 supports a number of input formats defining plans. For this purpose, the tool links a series of parsers and translators that internally (i) generate the format supported by the rewriting logic semantics of the language implemented in Maude and (ii) construct an object oriented plan model from Maude's output. The parsers are all generated from XML Schemas and BNF-like specifications by external tools, such as ANTLR, JAXB, and JavaCC. Some of the

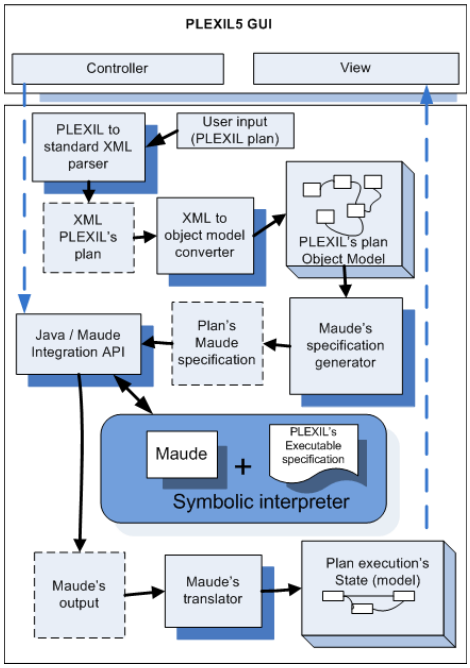

Fig. 2. PLEXIL5 logical components and their interaction.

XML schemas have been borrowed and adapted from PLEXIL's software distribution. Java and Maude communicate as processes at the operating system's level with help of the Java/Maude Integration API, developed as part of the PLEXIL5 framework.

The implementation of PLEXIL5 consists of 270 Java classes and 38 Maude modules, among other resources. The Java classes comprise 85K lines of code, of which 24K are automatically generated by the external tools. The Maude modules are 2K lines of code.

3.2 User Interaction

Once PLEXIL5 is launched for the first time, the user is required to select the folder containing PLEXIL's rewriting logic semantics. This selection is kept for future sessions and can be modified at any time through the graphical interface.

A plan is read from a file containing one of the several supported PLEXIL notations, then transformed into an object model, and ultimately presented to the user with a visual representation of the initial state of the plan. The visual representation of plans implemented in the prototype described in [11] was based on trees. That representation is only practical for plans with a small numbers of nodes. In the current version, plans are displayed by default as tables and the

hierarchical structure of plans is given by tabular indentations. The original tree representation of plans is still supported.

A plan can be edited by the user with the help of the graphical user interface. The plan can be accompanied by a *script* file, in XML format, describing the values of external variables at different macro steps. External variables can be initialized to random Boolean, integer, and floating-point values, and can be specified using an enumeration or a range. The following XML script specifies the values for the external variable T of integer type, for the plan *Exchange* in Figure 1. In the first macro step the variable T is assigned the value 2, at the second macro step it is assigned a random non-negative value, and in the third macro step it is assigned a value randomly chosen from 2 or 7.

```
<Script>
  <Step>
    <State name="T" type="int"><Value>2</Value></State>
  </Step>
  <Step>
    <State name="T" type="int"><RandomValue min="0"/></State>
  </Step>
  <Step>
    <State name="T" type="int">
      <RandomValue><Enum value="2"/><Enum value="7"/></RandomValue>
    </State>
  </Step>
</Script>
```

The translation process of a plan and its script only takes place the first time the plan is loaded and every time a plan is edited.

Plans can be executed at the level of the micro, quiescence, macro, and execution semantic relations, with undo-redo support. The tool can automatically generate formulas for checking invariant, pre, and post conditions, and the user can also define formulas from atomic predicates parameterized by the active plan. A Maude specification in the syntax of the rewriting logic semantics is generated from the object model every time the user requests to perform an action on the current state of execution. This Maude specification and the user's command are delegated to Maude via the Java/Maude integration API. The resulting output is then used to generate a new instance of the object model that is graphically presented to the user.

4 Formal Analysis in PLEXIL5

The formal analysis capabilities offered by PLEXIL5 are based on PLEXIL's rewriting logic semantics written in Maude. This section provides technical details on how these capabilities, i.e., simulation and debugging, model checking, and semantic validation, are implemented in PLEXIL5 via Maude's verification tools. This section uses standard notation and terminology of rewriting logic; the user is referred to [4] for more details.

Rewriting logic [10] is a general semantic framework that unifies a wide range of models of concurrency. Rewriting logic specifications can be executed in Maude, a high-performance rewriting logic implementation, and thus take advantage of all the formal analysis tools available in Maude. A rewriting logic specification is a tuple $\mathcal{R} = (\Sigma, E, R)$ where (Σ, E) is an order-sorted equational theory with signature Σ and equations E, and a set of rewrite rules R. The equational theory (Σ, E) induces the congruence relation $=_E$ on the set T_Σ of Σ-ground terms defined for any $t, u \in T_\Sigma$ by $t =_E u$ if and only if $(\Sigma, E) \vdash t = u$. The expression $\mathcal{T}_{\Sigma/E}$ denotes the initial algebra of (Σ, E). Similarly, a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the rewrite relation $\longrightarrow_{\mathcal{R}}$ on the set $T_{\Sigma/E}$ of E-equivalence classes of ground Σ-terms defined by any $t, u \in T_\Sigma$ by $[t]_E \longrightarrow_{\mathcal{R}} [u]_E$ if and only if $t \longrightarrow u$ can be deduced from $\mathcal{R}$ by the deduction rules in [3]. The tuple $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E}, \longrightarrow_{\mathcal{R}})$ is called the *initial reachability model* of $\mathcal{R}$. Intuitively, $\mathcal{T}_{\mathcal{R}}$ represents the concurrent system whose states are the set of E-equivalence classes of ground Σ-terms and whose concurrent transitions are specified by R.

4.1 Simulation and Debugging

The rewriting logic semantics of a synchronous language such as PLEXIL poses interesting practical challenges because Maude implements the maximal concurrency of rewrite rules by interleaving, i.e., asynchronous concurrency. To overcome this situation, the rewriting logic semantics $\mathcal{P} = (\Sigma_{\mathcal{P}}, E_{\mathcal{P}}, R_{\mathcal{P}})$ of PLEXIL implements a serialization procedure [13] that completely and correctly simulates PLEXIL's synchronous semantics. Since PLEXIL is deterministic, the serialization procedure implemented by $\mathcal{P}$ can be equationally defined in $E_{\mathcal{P}}$, thus avoiding the interleaving semantics associated with rewrite rules in Maude.

A PLEXIL node in $\mathcal{P}$ is a term object denoted $\langle O : C \mid a_1 : v_1, \ldots, a_m : v_m \rangle$, where O is the object's identifier corresponding to the node's qualified name, C is the object's class corresponding to the node's type, e.g., assignment, list, local variable, etc., and where v_1 to v_m are the current values of the attributes a_1 to a_m corresponding to the node's internal state of execution. An execution state of a plan has sort *PlxState* and the form $(\overline{\Gamma}, \overline{\pi})$, where $\overline{\Gamma}$ has the structure of a multiset of pairs representing the set Γ of external variables and their values, and $\overline{\pi}$ is a term that has the structure of a multiset of objects representing the set of nodes π. Multiset union is denoted by a juxtaposition operator that is declared associative and commutative, so that rewriting is multiset rewriting supported in Maude.

Given a PLEXIL plan p, PLEXIL5 internally generates the rewrite theory $\mathcal{P}(p)$ that extends $\mathcal{P}$ with the constructs of p. The rewrite theory $\mathcal{P}(p)$ is a formal model of p in rewriting logic and it induces the rewrite relation $\longrightarrow_{\mathcal{P}(p),m}$ that uses the equationally defined serialization procedure to soundly and completely simulate PLEXIL's synchronous micro relation for p.

Maude's strategy language [9] is used to simulate the quiescence, macro, and execution semantic relations from $\longrightarrow_{\mathcal{P}(p),m}$. By definition, the quiescence relation $\longrightarrow_{\mathcal{P}(p),q}$ is the normalized relation obtained from $\longrightarrow_{\mathcal{P}(p),m}$, namely,

$\longrightarrow_{\mathcal{P}(p),q} = \longrightarrow^{\downarrow}_{\mathcal{P}(p),m}$. Because $\longrightarrow_{\mathcal{P}(p),m}$ is deterministic, the quiescence relation $\longrightarrow_{\mathcal{P}(p),q}$ is also deterministic. For the purpose of simulating the macro and execution relations, PLEXIL5 allows for the definition of a sequence $\mathbf{\Gamma} = \Gamma_0, \Gamma_1, \ldots, \Gamma_n$ of collections of external variables indicating their value at each time step $0, 1, \ldots, n$ ($\mathbf{\Gamma}$ can be empty when the plan does not depend on external variables). For a sequence $\Gamma_0, \Gamma_1, \ldots, \Gamma_n$, the macro relation $\longrightarrow_{\mathcal{P}(p),M}$ is defined by $(\overline{\Gamma_i}, \overline{\pi}) \longrightarrow_{\mathcal{P}(p),M} (\overline{\Gamma'}, \overline{\pi'})$ if and only if $\Gamma' = \Gamma_{i+1}$ and $(\overline{\Gamma_i}, \overline{\pi}) \longrightarrow_{\mathcal{P}(p),q} (\overline{\Gamma_i}, \overline{\pi'})$, for $0 \leq i < n$. The execution relation $\longrightarrow_{\mathcal{P}(p),E}$ normalizes a given state with the macro relation and then normalizes the resulting state further with the quiescence relation in the last time step. It is formally defined by $\longrightarrow_{\mathcal{P}(p),E} = \longrightarrow^{\downarrow}_{\mathcal{P}(p),M} \circ \longrightarrow_{\mathcal{P}(p),q}$.

4.2 Model Checking

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 $\mathcal{E}_\Phi = (\Sigma_\Phi, E \uplus E_\Phi)$. Signature Σ_Φ contains Σ and a sort *Bool* with constant symbols $\bot$ and $\top$ of sort *Bool*, predicate symbols $\phi : State \rightarrow Bool$ 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 *Bool*. For $\phi \in \Phi$ and a ground term of sort *State* $t \in T_{\Sigma,State}$, 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 $\mathcal{K}^\Phi_{\mathcal{R}} = (T_{\Sigma/E,State}, \longrightarrow_{\mathcal{R}}, L_\Phi)$ 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.

PLEXIL5 supports LTL model checking of plans at the level of the micro relation on the sort *PlxState*. The set of atomic propositions is parameterized by the set of qualified names of nodes and variables (internal and external) 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 $LTL_\mathcal{N}$ for model checking a plan p with set of qualified names $\mathcal{N}$. The collection of PLEXIL Boolean expressions parameterized by $\mathcal{N}$ is denoted with $BExpr_\mathcal{N}$. They include comparison operators for Boolean and arithmetic expressions, evaluation of local variables, and lookups. 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* for testing PLEXIL's Boolean expressions. Formulas in $LTL_\mathcal{N}$ include the usual Boolean connectives, and the temporal connectives 'always' ($\mathbf{G}$), 'eventually' ($\mathbf{F}$), 'next' ($\mathbf{X}$), 'until' ($\mathbf{U}$), 'weak until' ($\mathbf{W}$), and 'release' ($\mathbf{R}$), all interpreted in the standard way.

Given a plan p, an initial state (Γ, π), and a $LTL_\mathcal{N}$ formula φ over the names $\mathcal{N}$ in p, PLEXIL5 uses Maude's LTL model checker to check $\mathcal{K}^{\Phi_\mathcal{N}}_{\mathcal{P}(p)}, (\overline{\Gamma}, \overline{\pi}) \models \varphi$,

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

with variables

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

Fig. 3. Parameterized atomic predicates $\Phi_{\mathcal{N}}$ and LTL formulas $\mathit{LTL}_{\mathcal{N}}$ in PLEXIL5.

where $\mathcal{K}^{\Phi_{\mathcal{N}}}_{\mathcal{P}(p)}$ is the Kripke structure associated with $\mathcal{T}_{\mathcal{P}(p)}$, with the set of states $T_{\Sigma(p)/E(p),PlxState}$, transition relation $\longrightarrow_{\mathcal{P}(p),m}$, and labeling function $L_{\Phi_{\mathcal{N}}}$.

PLEXIL5 provides interactive means for producing and visually inspecting counterexamples. The model checking window is equipped with three predefined checks that can be performed on any PLEXIL program: "check invariants", "check pre-conditions", and "check post-conditions". Pushing one of the three buttons generates the corresponding LTL formulas. Additionally, an input field in provided to enter custom, application specific LTL formulas specified using the above syntax. For example, the formula

$$\mathbf{G}\, \mathit{invariant}(\mathit{Exchange}, \mathit{true}) \ \wedge \ \mathbf{F}\, \mathit{status}(\mathit{Exchange}, \mathit{finished})$$

for the plan *Exchange* in Figure 1, tests the invariant of node *Exchange* and that it will eventually transition to state *finished*.

Counterexamples are displayed in a tree table with collapsible nodes, conforming to the PLEXIL program tree structure, and can be interactively navigated step-by-step for debugging and validation purposes.

4.3 Semantic Validation

The rewriting logic semantics $\mathcal{P}$ is being used to study variations and extensions of PLEXIL. This section provides examples of such variants and extensions that have been studied in PLEXIL5.

PLEXIL's macro relation is especially important because it is the semantic relation defining the interaction of a plan with the external environment. On the one hand, it is reasonable to have access to the external state as often as possible so that lookups in each atomic reduction can use the latest information available. On the other hand, it can be computationally expensive to implement such a policy because sensors or similar artifacts can significantly delay the execution of a plan. Another dimension of the problem arises when a guard of an internal loop depends on external variables: should the loop run-to-completion

regardless of the possible updates to the value of the variable in its guard, or should it stop at each iteration so that the value of the external variable can be updated? The rewriting logic semantics $\mathcal{P}$ has been modified to accommodate alternative specifications of PLEXIL's semantics with different definitions of the macro relation. These semantic variants of PLEXIL have been studied and exercised using PLEXIL5. Thanks to its modular design, PLEXIL5 can integrate the alternative semantics with a click of a button: the user has the freedom to choose the formal semantics of preference.

Another concrete example that illustrates the use of PLEXIL5 by the designers of the language is the addition of a gate condition called *exit condition*. The exit condition provides a mechanism for a clean interruption of execution. In order to support this feature in PLEXIL5, the specification of the PLEXIL's atomic relation in Maude was modified to include the intended semantics. Given the modular definition of the formal semantics none of the other rewriting relations were modified.

5 A Case Study

A cruise control system adapted from [2] is presented to showcase the model checking capabilities implemented in PLEXIL5. Originally, the model was designed for the Enhanced Operator Function Model (EOFM) formalism, which is intended for the study of human behavior in a human-computer interaction framework. However, PLEXIL shares many characteristics with EOFM, including the hierarchical structure of tasks decomposed into sub-tasks and the execution governed by conditions (*pre*, *post*, *repeat*, *invariant*).

5.1 Model Description

The model consists of three main components: car, driver, and stoplight, which execute synchronously. The operator drives the car on a street, approaching the stoplight. Other cars may merge into the lane from a side ramp, roughly midway through. The car has three controls represented in the model: the gas and break pedals to manage speed and acceleration, and a cruise button to switch the cruise mode on/off and set the cruise speed. The human operator's plan is to safely operate the controls of the car to achieve three sub-goals: (i) drive at a desired cruise speed (ii) avoid the possible merging traffic from the ramp, and (iii) obey the traffic light at the intersection, i.e., stop the car in time if the light turns red. All three properties can be represented in PLEXIL. Here we focus on the third, which is a safety property.

The model parameters are: the geometry of the intersection, i.e., the length of each street segment; the location of the ramp along the street, in distance units; the stoplight cycle length, in time units, for each color; and the speed range, in distance per time units.

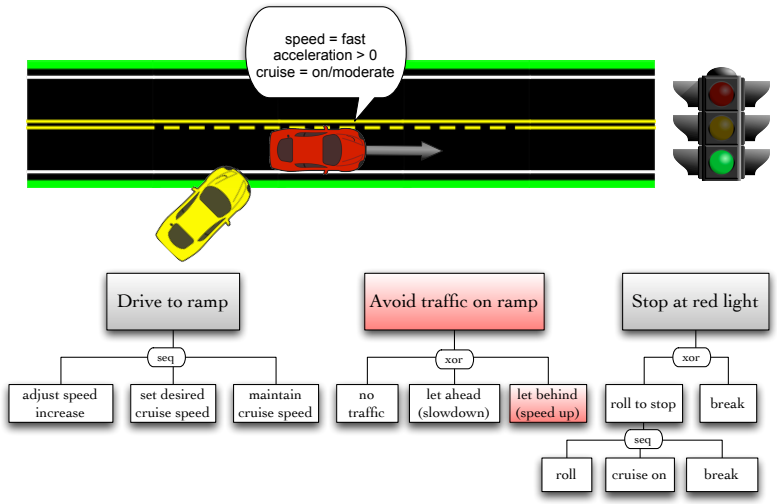

Fig. 4. Cruise control model with task hierarchy.

Model variables. The model variables and their range are selected according to an abstraction scheme that discretizes the values to allow finite state model checking, yet leaves sufficient information to make the study relevant.

- $distance \in [0 \dots 55]$, the distance of the car to the intersection;
- $time \in [0 \dots 28]$;
- $speed \in \{stopped = 0, slow = 1, moderate = 2, fast = 3\}$;
- $acceleration \in \{-1, 0, 1\}$;
- $cruise\_enabled \in \{true, false\}$;
- $cruise\_speed \in \{0, 1, 2, 3\}$;

Transitions. The car advances according to its speed until it reaches the intersection, formally, update $distance := distance - speed{*}timestep$ while the condition $speed > 0 \wedge distance > 0$ holds. The discretized speed can change by at most one unit at a time, hence the possible values for acceleration are only $\{-1, 0, 1\}$. The stoplight counts down the time units to the end of the green-yellow-red cycle by assigning $stoplight := stoplight - timestep$. The light is red in the time interval $[0 \dots 8]$, yellow in $[9 \dots 12]$, and green in $[13 \dots 28]$.

The complexity resides in capturing the decision making of the driver. In the first segment, the driver wants to set the cruise control to a desired speed (e.g., *moderate*). The driver has the choice to accelerate from slow or decelerate from *fast*, then enable the cruise control which will maintain the desired speed. On the second segment, the driver needs to react to merging traffic from the ramp. If any car is on the ramp, the driver may choose to let the other car in front by slowing down, or behind by speeding up. On the last segment, the driver has to react to the stoplight turning red. The driver may choose to maintain the speed

and then break before reaching the stoplight, or roll to a stop by releasing the gas pedal.

Comparison with the EOFM model.

- The original abstraction has been refined in PLEXIL to allow more distance and time divisions, making it more realistic; in the EOFM model the distance is heavily discretized (abstract locations 0 to 7) and not coordinated with the time to travel each segment.
- Non-determinism is introduced by lookups of environment variables. The script plays out a sequence of random choices for three Boolean environment variables: *MergingTraffic*, *LetBehind*, *RollStop*.
- Some of the concepts are essentially cognitive in nature, as they depend on the subjective (sometimes erroneous) perceptions and assessments of the situation by the human operator, hence they cannot be as naturally captured in the formal model. However, both normative and erroneous behaviors are captured in the PLEXIL model, and it is the job of the model checker to discover violations.
- The synchronous behavior is natural in PLEXIL, no further instrumentation is necessary, while in EOFM synchrony has to be expressly specified, using appropriate decomposition operators.

5.2 Verification

The property of interest can be expressed either as a global invariant in the PLEXIL model itself and checked with the generic "check invariants" button, or entered in the LTL Model Checking dialog window. The safety property is specified in the top level task node *Main* as the invariant condition:

$$not(stoplight <= red\ and\ distance == 0\ and\ speed > 0),$$

stating that it is not the case that the vehicle is moving at the intersection when the light is red.

The PLEXIL5 simulator shows that the execution of the plan ends with the outcome *invariantFail* for the root node (and *parentFail* for the successor nodes) when the environment variables *MergingTraffic*, *LetBehind*, and *RollStop* are all *true*. The result of model checking the safety property is an execution trace where the formula is violated. The counter example can be described as follows:

1. the car enters at *low* speed at $distance = 55$ and $time = 28$;
2. the driver accelerates to the desired *moderate* speed and sets the cruise on at $time = 20$ and $distance = 42$;
3. at the ramp, with $distance = 33$, the driver decides to let the merging car behind by accelerating to *fast* at $time = 12$ and $distance = 25$;
4. the stoplight light turns yellow, the driver chooses to roll to a stop (assessing there is sufficient distance to the intersection to do so, by releasing the gas pedal);

5. with the acceleration negative, the driver does not disengage the cruise mode, the cruise control kicks in and maintains the cruise *speed* to *moderate* for one execution cycle at $time = 6$ and $distance = 10$;
6. the effect of the automation is that the (now necessary) breaking is too late to decrease the speed from *moderate* to *low* at $time = 2$ and $distance = 2$, and then *stopped* in two execution cycles; and
7. when time expires, the car is moving in the intersection on the red light.

The PLEXIL5 model checking environment provides the means for detecting the aforementioned error using the predefined "check invariant" test. To correct the problem, the node corresponding to the "roll to stop" action has to be rectified, in order to include a check on the status of the cruise control. The driver either has to make sure it is disabled before initiating the "roll to stop" option or manually disable it. In PLEXIL, this can be instrumented via a *start condition* or, by duality, with the corresponding negated *skip condition*. No other combination of environment lookup variables leads to violations in this model.

The full model of the cruise control system consists of 252 lines of PLEXIL code. The generated Maude file is 929 lines long.

6 Related Work and Conclusion

An executable semantics of PLEXIL has been developed by P. J. Strauss in the Haskell language [16] with the aim of analyzing features of the language regarding the plan interaction with the environment. As a result, new data types representing the external world have been proposed for more dynamic runtime behavior of PLEXIL plans. More recently, D. Balasubramanian et al. have proposed Polyglot, a framework for modeling and analyzing multiple Statechart formalisms, and have initiated research towards the formal analysis of a Statechart-based semantics of PLEXIL [1]. In rewriting logic literature, similar approaches to the one used in PLEXIL5 have been proposed for other languages and protocol analysis. In particular, A. Verdejo and N. Martí-Oliet [17] have explored the idea of having easy-tool-building techniques from operational semantics specified in Maude. S. Santiago et al. [15] have developed a graphical user interface that animates the Maude-NPA verification process, displaying the complete search tree and allowing users to display graphical representations of final and intermediate nodes of the search tree. Maude-NPA is a crypto protocol analysis tool developed in Maude that takes into account algebraic properties of crypto-systems.

This paper reported significant progress on the evolution of PLEXIL5, an environment for the verification and validation of NASA's synchronous language PLEXIL. The environment uses the formal semantics of the language written in Maude to formally analyze PLEXIL plans. Maude is a rewriting logic formalism that provides advanced verification tools such as a fast rewriting engine and a LTL model checker. In PLEXIL5, the user is presented with the option to execute any combination of the micro, quiescence, macro, and execution reduction relations. In this way, the user has the freedom to determine the level of detail for

simulating and debugging plans. The verification tools are available in PLEXIL5 through a graphical interface that does not require knowledge of rewriting logic or the Maude system.

The formal environment has been used by the developers of the language to investigate semantic variations and extensions of PLEXIL. These include a new semantics for the execution of loops and a new feature in the language to handle exit conditions. Furthermore, several minor issues in the original intended semantics of PLEXIL have been identified and corrected. PLEXIL5 has become a formal benchmark for executives and will be part of PLEXIL's distribution in an upcoming release.

An important subset of PLEXIL's core language is currently supported by PLEXIL5. The main features of the language that are not supported by the formal semantics are array variables (arrays are not directly supported in Maude), PLEXIL's resource model, which enables the specification of resource requirements for commands, and Update nodes, which provide an importing mechanism to the language. Regarding research on PLEXIL5's rewriting logic semantics, future work will explore the possibility of having an operational semantics of the language using the framework presented in [12], so that the dependency between the rewrite rules specifying the atomic relation and the serialization procedure can be eliminated. Another interesting alternative is to study the extension of the K framework [14] with priorities for state transitions, so that it can accommodate the specification of the atomic relation. Regarding the verification and validation capabilities in PLEXIL5, future work will add support for symbolic execution and concolic testing of PLEXIL plans, and will study scalability issues with mid-size and large plans.

Acknowledgments. The authors would like to thank Michael Dalal and the Planning and Scheduling group at NASA Ames for fruitful discussions on PLEXIL and suggestions for the PLEXIL5 tool. They are also grateful to the anonymous referees for comments that helped to improve the paper. This work is supported by NASA's Autonomous Systems and Avionics Project, Software Verification Algorithms. The first author has been partially supported by NSF grant CCF 09-05584. The first, second, and fourth authors have been partially supported by the National Aeronautics and Space Administration at Langley Research Center under Research Cooperative Agreement No. NNL09AA00A awarded to the National Institute of Aerospace.

References

1. Balasubramanian, D., Păsăreanu, C., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple Statechart formalisms. In: Dwyer, M.B., Tip, F. (eds.) ISSTA. pp. 45–55. ACM (2011)
2. Bolton, M.L., Bass, E.J., Siminiceanu, R.I.: A systematic approach to model checking human-automation interaction using task analytic models. IEEE Transactions on Systems, Man, and Cybernetics–Part A: Systems and Humans 41(5), 961–976 (2011)
3. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360(1-3), 386–414 (2006)

4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
5. Dowek, G., Muñoz, C., Păsăreanu, C.: A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA (2008)
6. Dowek, G., Muñoz, C., Păsăreanu, C.: A formal analysis framework for PLEXIL. In: Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems (September 2007)
7. Dowek, G., Muñoz, C., Rocha, C.: Rewriting logic semantics of a plan execution language. In: Klin, B., Sobocinski, P. (eds.) SOS. EPTCS, vol. 18, pp. 77–91 (2009)
8. Estlin, T., Jónsson, A., Păsăreanu, C., Simmons, R., Tso, K., Verma, V.: Plan Execution Interchange Language (PLEXIL). Technical Memorandum TM-2006-213483, NASA (2006)
9. Martí-Oliet, N., Meseguer, J., Verdejo, A.: A rewriting semantics for maude strategies. Electronic Notes in Theoretical Computer Science 238(3), 227–247 (2009)
10. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
11. Rocha, C., Muñoz, C., Cadavid, H.: A graphical environment for the semantic validation of a plan execution language. In: IEEE International Conference on Space Mission Challenges for Information Technology. pp. 201–207. IEEE Computer Society, Los Alamitos, CA, USA (2009)
12. Rocha, C., Muñoz, C.: Simulation and verification of synchronous set relations in rewriting logic. In: da Silva Simão, A., Morgan, C. (eds.) SBMF. Lecture Notes in Computer Science, vol. 7021, pp. 60–75. Springer (2011)
13. Rocha, C., Muñoz, C., Dowek, G.: A formal library of set relations and its application to synchronous languages. Theoretical Computer Science 412(37), 4853–4866 (2011)
14. Rosu, G., Serbanuta, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397 – 434 (2010)
15. Santiago, S., Talcott, C.L., Escobar, S., Meadows, C., Meseguer, J.: A graphical user interface for Maude-NPA. Electronic Notes in Theoretical Computer Science 258(1), 3–20 (2009)
16. Strauss, P.J.: Executable semantics for PLEXIL: simulating a task-scheduling language in Haskell. Master's thesis, Oregon State University (2009)
17. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in Maude: CCS and LOTOS. Formal Methods in System Design 27(1-2), 113–172 (2005)
18. Verma, V., Jónsson, A., Păsăreanu, C., Iatauro, M.: 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)