A Graphical Environment for the Semantic Validation of a Plan Execution Language

Camilo Rocha*, César Muñoz† and Héctor Cadavid‡
*Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61820
hrochan2@cs.uiuc.edu
† National Institute of Aerospace
Hampton, VA 23666, USA
munoz@nianet.org
‡ Escuela Colombiana de Ingeniería
Bogotá, Colombia
hcadavid@escuelaing.edu.co

***Abstract*—This paper presents PLEXIL5, a graphical environment providing an user-friendly interface to the formal operational semantics of PLEXIL. PLEXIL is a synchronous plan execution language developed by NASA to support autonomous space operations. PLEXIL5 serves as a testbed for designers, developers and users of PLEXIL's executive system to validate, maintain, and debug the implementation of the system against the formal semantics of the language. In PLEXIL5, the executable formal semantics of PLEXIL is implemented as a rewriting logic theory in Maude's language.**

I. Introduction

The Plan Execution Interchange Language (PLEXIL) [1] is a high-level plan execution language developed by NASA. PLEXIL belongs to the family of synchronous reactive languages such as Esterel [2], Lustre [3], and Signal [4], where the only non-determinism allowed originates from the interaction with the environment. However, in contrast to these general purpose languages, PLEXIL has been specifically designed to support autonomous spacecraft operations. In particular, the PLEXIL Universal Executive (UE), the software system that interprets and executes PLEXIL plans, could be deployed in multiple platforms, usually with limited computational resources and under uncertain physical conditions.

Given the critical nature of spacecraft operations, PLEXIL's operational semantics has been formally defined [5] and several properties of the language have been mechanically verified [6] in the Program Verification System (PVS) [7]. This semantics has been also implemented in the formal notation of Maude [8], a high-performance implementation of the rewriting logic framework [9]. Rewriting logic is a computational logic that has good properties as a general and flexible logical and semantic framework, in which a wide range of logics and models of computation can be faithfully represented. In rewriting logic, concurrent computations and deduction coincide: rewriting logic's inference system allows to derive as proofs all the concurrent computations of the system axiomatized by a rewriting logic theory [10].

Camilo Rocha and César Muñoz were supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NCC-1-02043. Héctor Cadavid was supported by the Escuela Colombiana de Ingeniería, Bogotá, Colombia. César Muñoz (Cesar.A.Munoz@nasa.gov) is currently affiliated to NASA Langley.

The formal semantics of PLEXIL is organized as a stack of abstract relations, which range from an atomic relation describing the evolution of a single computational element of PLEXIL to an execution relation describing the evolution of the whole plan after the occurrence of a series of external events. For efficiency reasons, an executive system may profit from properties of the language, such as determinism and compositionality, to implement these relations in a different way. Therefore, there may not be a one-to-one relation between the formal semantics and the executive implementation. A discrepancy between the executive and the formal semantics does not necessarily mean that the executive is *incorrect*. After all, the language is still evolving and the executive serves as an implementation of the intended semantics.

We have developed the graphical environment PLEXIL5, PLEXIL's Formal Interactive Visual Environment, where designers and developers can validate the formal semantics of the language against an intended semantics, and plan developers can execute and debug PLEXIL plans. The graphical environment consists of a graphical component written in Java, the formal executable semantics of PLEXIL written in Maude, and a bidirectional translator from Maude syntax to Java objects and vice-versa. PLEXIL5 provides an user-friendly interface that enables step-by-step execution of PLEXIL plans for recorded sequences of external events. It also allows for the inspection of the internal state and the execution status, backtracking, traceability, and cross-reference to the formal semantics.

The rest of this paper is structured are follows. Section II gives an overview of PLEXIL and its formal semantics. Section III presents the graphical interface. Section IV discusses related work. The last section concludes this work and gives perspectives on future work.

II. PLEXIL and Its Semantics: An Overview

Plan execution is a cornerstone in systems involving intelligent software agents such as robotics, unmanned vehicles, and habitats. The Plan Execution Interchange Language (PLEXIL) is an open source synchronous language developed by NASA to support automation of spacecraft operations.[1] In this Section, we present a brief overview of PLEXIL and use an example to illustrate many constructs of the language. For a detailed description of its formal semantics, and more examples, we refer the reader to [5], [6].

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 call 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 state* of a node is given by states such as `Inactive`, `Waiting`, `Executing`, etc. The *external state* is accessed through *lookups* on environment variables.

```
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;
    }
  }
}
```

Fig. 1. SafeDrive: A PLEXIL Plan Example

In Figure 1 we present SafeDrive, an example of a PLEXIL plan. In this particular example, the plan tasks are represented by the interior nodes `SafeDrive` and `Loop`, and the leaf nodes `OneMeter`, `TakePic` and `Counter`. The root of the plan is the node `SafeDrive`. `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` has finished its execution, while `Pre` is 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 `Wheel`.

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.

The execution of a plan proceeds in discrete time steps, called *macro steps*. All the external events are processed in the order in which they are received. An external event and all its cascading effects are processed until *quiescence* before next event is processed; this behavior is known as *run-to-completition* semantics. A macro step of execution consists of a number of *micro steps*. A micro step is the parallel synchronous execution of *atomic steps* of individual nodes. Consequently, the semantics of PLEXIL is regarded as a compositional stack of four execution relations, namely, atomic, micro, quiescence and macro relations. Moreover, the semantics of PLEXIL is given in terms of the semantics for atomic and micro relations, as the other relations are based on well-known abstract relations.

The atomic relation defines the individual evolution of the elements in the internal state of the plan, i.e., nodes and variables, at a given time. Those evolutions are originally described by a dozen of state transition diagrams, in an ad-hoc graphic notation, indexed by the type and state and of nodes. Figure 2 shows the state transition diagram[2] for a node of type `Assignment` in state `Executing`. Oval vertices in the diagram represent states of nodes, rectangular yellow ones represent conditions changes with their respective enabling value, rectangular green/red and white vertices represent success/failure codes and respective actions, and rhombus lyle vertices represent path decisions on check conditions (like an if-then-else construct). The state transition diagram in Figure 2 shows that an `Assignment` node evolves from state `Executing` to either state `Finished` in the case the invariant of any of its ancestors is false (in the hierarchical structure of the plan), or state `IterationEnded` in the case its invariant condition is false or its termination condition is true. The labeling on the edges leaving the vertex `Executing` in the diagram, induces an evaluation strategy for the conditions in the diagram, where lower the number, higher the priority.

For example, recalling our plan SafeDrive in Figure 1,

[1] PLEXIL is electronically available from http://plexil.sourceforge.net.

[2] Taken from http://plexil.wiki.sourceforge.net.

the node `Counter` in any particular time when in state `Executing` and its `EndCondition` changes to true while the other two conditions are not enabled, will increment the variable `pictures` in one unit, and then evolve to state `IterationEnded`.

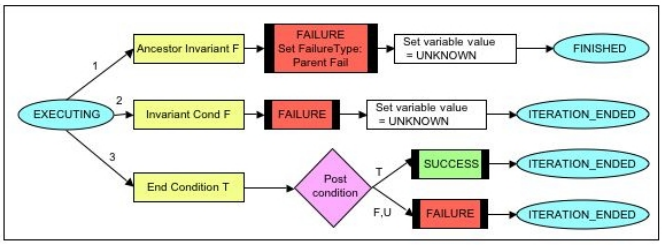

Fig. 2. Transitions from state `Executing` for nodes of type `Assignment`.

The micro relation, as already mentioned in the preceding paragraphs, is the parallel synchronous execution of the atomic relation. It relates internal states to internal states by means of the atomic relation, namely, it records synchronously changes in the state of nodes and variables. If two different nodes in the internal state write to the same variable in a particular time, only the update of the node with the higher priority is considered.

The modular definition of the PLEXIL semantics is considerably advantageous for both designing the language itself and developing plans in the language. On one hand, features of the language can certainly be identified with a layer of the stack of execution relations; then evolving or adding new features to the language becomes often local to a particular layer. Moreover, properties of each execution relation can be locally studied, taming the inherent complexity associated to the power of the entire language. For example, it has been formally proven in the Program Verification System –PVS- that the micro relation is deterministic with respect to the environment [6], that is, once the environment is known the execution of the micro relation is determined. Then, by the modularity of PLEXIL semantics, the determinism is inherited by quiescence and macro. On the other hand, plan developers are able to focus their testing and debugging efforts to a particular execution relation. For example, to debug a plan for which no interaction with the external environment is required, it would suffice to debug such plan at the level of quiescence.

In the scenarios described above, it would be convenient and beneficial to have a fully functional graphical environment supporting the interaction with the modular semantics of PLEXIL, namely, one allowing designers to visualize with ease changes to the execution relations of the language and developers to recreate plans at different execution levels.

III. PLEXIL5: A Formal Interactive Visual Environment for the Symbolic Execution of PLEXIL

Both language designers and plan developers demand capable supporting tools when interacting with a language. In this Section, we present PLEXIL5, PLEXIL's Formal Interactive Visual Environment, for visual and symbolic execution of PLEXIL plans. In PLEXIL5 users can visualize and interact with plans and their execution. We first describe the architecture of PLEXIL5 and then state the main features provided by the graphical environment.

A. Architecture

PLEXIL5 consists of three components, as depicted in Figure 3:

- A *graphical component* written in Java with an object oriented model representing plans and their execution behavior.
- A *symbolic interpreter* written in Maude [8], a Rewriting Logic [9] interpreter, which executes PLEXIL plans written in PLEXIL-MAUDE, an intermediate language between PLEXIL syntax and Maude's syntax.
- A *bidirectional translator* from the object oriented model to PLEXIL-MAUDE.

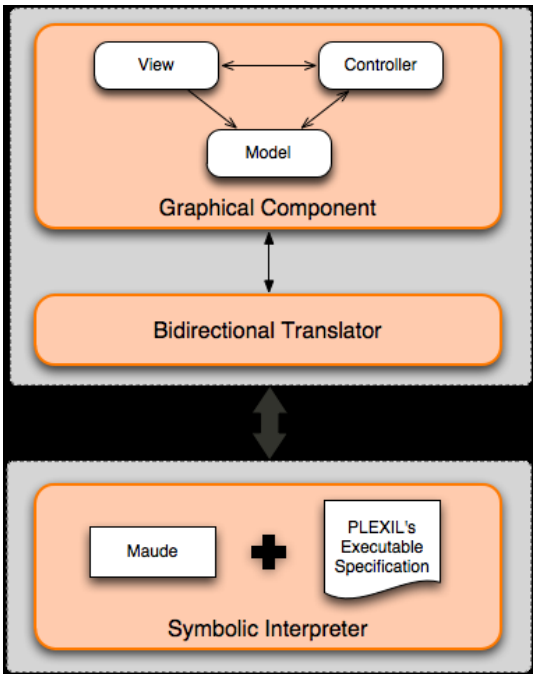

Fig. 3. PLEXIL5 Architecture

The graphical component was designed and developed using the Model-View-Controller (MVC) architectural/design pattern (see [11], [12] for a comprehensive definition of MVC). The model corresponds to the Java objects representing a plan. The view is, currently, a single view consisting of a tree-like structure and a indexed set of list. The tree structure represents plans, while the indexed set of lists represent the external state of plans at particular times. The controller consists of a custom "controller-facade" class and "listener" classes extending the Java framework. JGraph[3], an open source Java

[3]Available at http://www.jgraph.com.

Swing diagramming library, is used to display the internal state. Due to the adherence of the design to the MVC pattern, it would be easy to add new types of view to the graphical component.

The symbolic interpreter is an executable specification in rewriting logic of PLEXIL semantics. Rewriting logic is a computational logic that has good properties as a general and flexible logical and semantic framework, in which a wide range of logics and models of computation can be faithfully represented. A rewriting logic theory is a tuple $\mathcal{R} = (\Sigma, E, R)$ where (Σ, E) is an membership equational theory with signature Σ and equations E, and a set of rewrite rules R. From the language semantics viewpoint, a theory $\mathcal{R}$ axiomatizes a *concurrent system*, whose *states* are E-equivalence classes of ground Σ-terms (i.e. elements of the abstract data type (Σ, E)), and whose *atomic* transitions are specified by the rules R. In rewriting logic concurrent computations and deduction coincide: rewriting logic's inference system allows to derive as proofs all the concurrent computations of the system axiomatized by $\mathcal{R}$ [10]. Therefore, the specification of PLEXIL is a rewriting logic theory defining the operational semantics of PLEXIL that is directly executable in the Maude language[4]. We refer the reader to [8], [9] for a self-contained definition of rewriting logic and the use of Maude as a specification language and execution environment.

The bidirectional translator is automatically generated from a BNF specification of PLEXIL-MAUDE by JavaCC[5]. JavaCC is a Java parser generator written in Java that generates top-down (recursive descent) parsers.

The symbolic interpreter and the bidirectional translator communicate as processes at the operating system's level. There is also the possibility of connecting these two components through sockets (both Maude and Java have communication support through sockets). The latter configuration allows PLEXIL5 to execute in a client-server fashion, in addition to the stand-alone fashion imposed by the former configuration. In PLEXIL5, the location of the symbolic interpreter can be configured by the user. Once this is done, the integration of the different components is seamless for the end user.

The installation process of PLEXIL5 is as simple as unpacking a compressed file. PLEXIL5 requires a version of the Java Virtual Machine for Java 5 or greater. Once PLEXIL5 is installed, the application can be launched from the command line by a script (included in the distribution). Once launched, the user may select the version of the symbolic interpreter to be used (selecting the symbolic interpreter is mandatory the first time after installation); PLEXIL5 distribution comes with the current executable semantics of the language(the symbolic interpreter) and Maude's 2.4 distribution. The user can browse the file system for files containing PLEXIL plans. In order to inspect and execute a plan, the user interacts with PLEXIL5 vía buttons: the controller in the graphical component takes the events from the user interaction, sending the current state of execution and the execution command associated to the event to the bidirectional translator, that in turn will call and receive information from the interpreter, obtaining a new state in the execution which is translated to the object model and finally displayed to the end user by the graphical component. Figure 4 depicts a sequence diagram representing the interaction of the user with PLEXIL5.

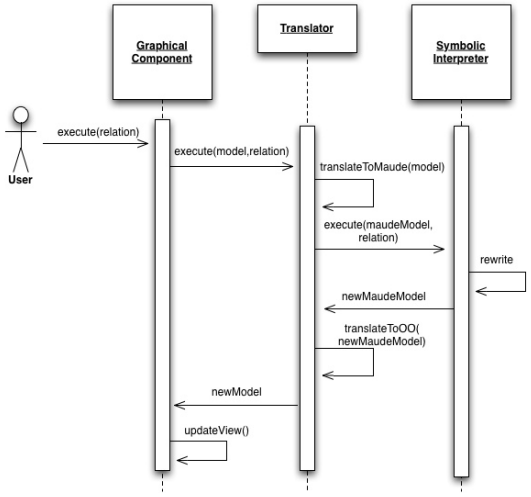

Fig. 4. Diagram sequence for the execution of a relation in PLEXIL5.

B. Main Features

PLEXIL5 represents the internal state of a plan as a tree and the external state as lists of (the external) variables; Figure 5 shows the graphical rendering of SafeDrive in PLEXIL5.

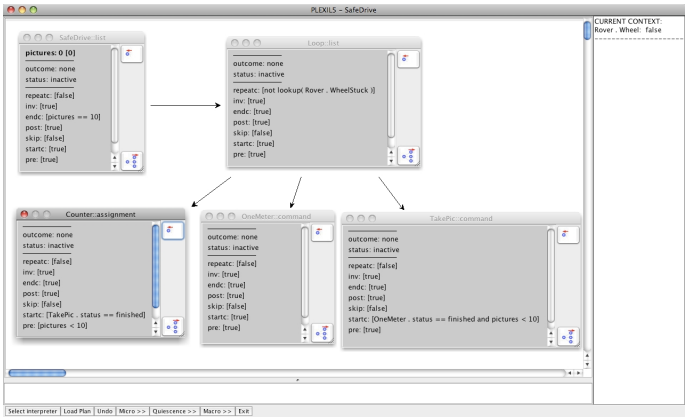

Fig. 5. SafeDrive in PLEXIL5.

The visualization of a node consists of its name, gate and check conditions, and state (including its variables). The user can visualize a node at different levels of detail by choosing to show or hide parts of its information. The environment also supports toggling back and forth between showing or hiding descendants of a node. Nodes can also be repositioned (using the mouse).

The external state of a plan can consist of several environment instances. Each instance is represented as a list of

[4]Available at http://maude.cs.uiuc.edu/.
[5]Available at https://javacc.dev.java.net.

variables with their corresponding value. The user can browse through all environment instances; it is also possible to sort alphabetically an environment instance. Both variable names and values can be edited.

PLEXIL5 allows the user to browse for plans in the file system. Loaded plans can be symbolic executed via the micro, quiescence, and macro execution relations. In each case, there is an undo feature returning the state of the the plan to the state previous to the last execution step, if any. Nodes are colored according to their particular state and success/failure status associated by the execution. Figure 6 depicts the execution of SafeDrive in which `SafeDrive` is in state `Waiting` and `Counter` is in state `Finishing`.

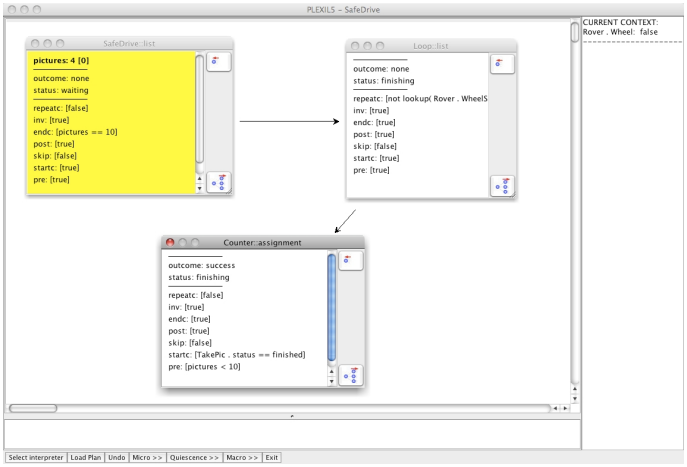

Fig. 6. Executing SafeDrive.

The execution trace is kept by PLEXIL5 and its content is displayed at the bottom of the graphical component. This trace indicates which atomic steps have been taken by the nodes at any particular time of the execution. The labels used to identify such steps map faithfully to the transition diagrams defining the semantics of the atomic execution relation described in Section II.

PLEXIL5 is highly portable as the graphical component is written in Java and the symbolic execution engine in Maude, for which there are virtual machines and interpreters, respectively, for several operating systems. The authors have tested the current version of PLEXIL5 in Mac Os X 10.4 and 10.5, Linux and Windows XP and Vista.

IV. Related Work

The natural candidate to compare PLEXIL5 to is the Universal Executive, the official implementation and execution environment of PLEXIL. In this Section we give a brief comparison between these tools from the perspective of relevant features to both language designers and plan programmers. Visual environments for other synchronous languages are also mentioned, and the reader is referred to formal developments supporting the validation of synchronous languages in a more general setting.

The Universal Executive (UE) is a Unix-based implementation and execution environment for PLEXIL. It comes with the Test Executive (TE), a plan execution simulator, and the Lightweight Universal Executive Viewer (LUV), a graphical interface to PLEXIL plans. Given a plan file (in PLEXIL-XML) and a simulation script (also in XML) representing the external world, the plan is executed in the TE. The TE is parametrized by a debug configuration file. A typical debug configuration file shows the state transitions and final outcomes of every node in the plan.

The LUV is a graphical front-end to the TE, in which plans are presented in a hierarchical table-like view, and status and output of each node during execution is visible in the main window. By clicking in the row corresponding to a particular node in the hierarchical structure of the plan, auxiliary tabbed windows show the gate and check conditions, assignments, variables, command actions, etc. In the LUV it is also possible to hide/show nodes in the plan's hierarchy and search for nodes. For debugging, it is possible to edit debug configuration files and add interruption /breakpoints directly from the LUV.

With respect to the Graphical Environment presented in this paper, the TE of the UE is the analogous to the executable semantics in Maude of PLEXIL5, in the same way the graphical and translator components are analogous to the LUV. The graphical interface of LUV is more developed that the one offered by PLEXIL5. However, the PLEXIL5 is directly supported by a formal semantics of a subset of the language. From a functional point of view, the most significant difference between the LUV and PLEXIL is that PLEXIL5 supports step-by-step and undo/redo operations with different execution relations. This feature is highly convenient for designers as step-by-step and undo/redo operations allows for the discovery of mismatches in the semantics and the localization of logical errors. In future work, we expect to integrate LUV and PLEXIL5 to provide powerful graphical execution environment of PLEXIL plans supported by a formal executable semantics.

Graphical environments for the specification and verification of reactive systems commonly rely on synchronous languages as they are suitable for the development of reactive systems [13]. Esterel and Lustre are commercial synchronous languages well positioned in this niche with powerful graphical model-based design and verification tools (see [14], [15] for a survey). There are, nevertheless, two main facts worth noticing. First, the companion graphical environments for these commercial languages are, in general, only suitable for supporting program development but not for validating language design. Second, by having Rewriting Logic and Maude as PLEXIL5's underpinnings, out-of-the-box formal tools already available for Maude become available to PLEXIL5 for free. For example, having an interface in PLEXIL5 for Maude's Model Checker is the only development needed in order to have a model checker for PLEXIL. As a matter of fact, this is the first task the authors will pursue in the immediate future. In the same spirit, language designers and program developers can hope for a theorem prover too.

The executable specification of PLEXIL semantics implementing the symbolic interpreter benefits from the previous developed formal specification of PLEXIL operational semantics in PVS [5]. In particular, the property of *determinism* satisfied by the micro execution relation accounts for an efficient implementation of the semantics, as this property is sufficient for the confluence of the rewriting relation induced by the micro relation: rewrite rules can be turned into oriented equations and search with rules then becomes "blind" reduction with equations while maintaining completeness. This is important not only for efficiency reasons at execution time, but vital for model checking as the state space is greatly reduced.

V. Conclusion and Future Work

PLEXIL5, PLEXIL's Formal Interactive Visual Environment, is an open source and portable graphical environment for the validation of PLEXIL semantics. PLEXIL5 consists of three components: a graphical component for visualizing plans, a symbolic formal interpreter by means of an executable specification in rewriting logic of PLEXIL semantics, and bidirectional translator between the graphical component and the symbolic interpreter.

The graphical component allows users of the tool to visualize the internal state of plans in a tree-like configuration and the external state as an indexed list of sets of pairs of variables and values. It endows the user with the functionality for executing PLEXIL plans with different execution relations in a step-by-step fashion, and to undo/redo steps of execution, among other things. Because of these features, PLEXIL5 supports the rapid and user-friendly validation of the language. Besides being a tool supporting the validation of PLEXIL semantics, PLEXIL5 is also a tool suitable and convenient for plan developers.

Our current implementation is a proof of concept. Although PLEXIL5 does not support yet all the syntactic elements of PLEXIL, we have already discovered, and fixed, a semantic rule in the language that deals with assignment of local variables. During the initial evolution of the language, we expect that PLEIXL5 will become an important tool for PLEXIL's designers.

In the near future, we intend to extend PLEXIL5 in the following directions:

1) support for *formal analysis* by incorporating Maude's native verification tools such as Maude's LTL Model Checker;
2) improve user experience by defining new views for displaying plans and I/O support for other idioms used in PLEXIL's community for the definition of plans;
3) integrate PLEXIL5 with the Universal Executive, so it is an option for the user to execute plans using the formal semantics of the language or its standard and optimized implementation.

We believe that integrating Maude's model checker in PLEXIL5 is a relatively simple development, thanks to Maude and the modularity of its vast companion of formal analysis tools. As for adding new views to the graphical component, it is virtue of the MVC meta-pattern to make it simple and effortless. We believe that the integration of PLEXIL5 with the Universal Executive is key to the more ambitious but necessary goal of having formal analysis tools available for the Universal Executive.

Finally, it is important to remark that the definition and implementation of PLEXIL's executable semantics as a rewriting logic theory can be seen as a template for the formal specification of other synchronous languages in rewriting logic. In particular, the work presented in this paper directly contributes to the rewriting logic semantics project [16] in which rewriting logic semantics has been extensively used as a logical framework for operational semantics definitions of programming languages. On top of that, we are contributing a capable user interface enriching user experience for the validation of the semantics, a combination resulting in a very strong and yet simple approach for the formal validation of PLEXIL semantics.

Acknowledgment

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] V. Verma, A. Jónsson, C. S. Păsăreanu, and M. Iatauro, "Universal executive and PLEXIL: Engine and language for robust spacecraft control and operations," in *American Institute of Aeronautics and Astronautics Space 2006 Conference*, 2006.

[2] G. Berry, "The foundations of Esterel," in *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. [Online]. Available: citeseer.ist.psu.edu/62412.html

[3] 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.

[4] P. L. Guernic, T. Gautier, M. L. Borgne, and C. L. M. e, "Programming real-time applications with SIGNAL," in *Proceedings of the IEEE, Volume 79(9)*, 1991, pp. 1321–1336.

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

[6] ——, "A formal analysis framework for PLEXIL," in *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*, September 2007.

[7] S. Owre, J. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, Jun. 1992, pp. 748–752.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science, vol. 4350. Springer, 2007.

[9] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, no. 1, pp. 73–155, 1992.

[10] T.-F. Serbanuta, G. Rosu, and J. Meseguer, "A rewriting logic approach to operational semantics," *Inf. Comput.*, vol. 207, no. 2, pp. 305–340, 2009.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. Wiley, 1996, vol. 1.

[12] M. Fowler, *Patterns of Enterprise Application Architecture*. Reading, Massachusetts: Addison Wesley, Nov. 2002.

[13] A. K. Bhattacharjee, S. D. Dhodapkar, S. A. Seshia, and R. K. Shyamasundar, "A graphical environment for the specification and verification of reactive systems," in *SAFECOMP*, ser. Lecture Notes in Computer Science, M. Felici, K. Kanoun, and A. Pasquini, Eds., vol. 1698. Springer, 1999, pp. 431–444.

[14] S. Ramesh and P. Sampath, *SCADE: Synchronous Design and Validation of Embedded Control Software*. Springer Netherlands, 2007.

[15] G. Berry, "Synchronous design and verification of critical embedded systems using scade and esterel," in *FMICS*, ser. Lecture Notes in Computer Science, S. Leue and P. Merino, Eds., vol. 4916. Springer, 2007, p. 2.

[16] J. Meseguer and G. Rosu, "The rewriting logic semantics project," *Theor. Comput. Sci.*, vol. 373, no. 3, pp. 213–237, 2007.