Integration of Formal Methods in the Development of a Language for Autonomous Spacecraft Operations

Lauren M. White
NASA's Langley Research Center
Hampton, VA
lauren.m.white@nasa.gov

César Muñoz
NASA's Langley Research Center
Hampton, VA
cesar.a.munoz@nasa.gov

Marco A. Feliú
Analytical Mechanics Associates
Hampton, VA
marco.feliu@nasa.gov

Mariano Moscato
Analytical Mechanics Associates
Hampton, VA
mariano.m.moscato@nasa.gov

Laura Humphrey
NASA's Langley Research Center
Hampton, VA
laura.r.humphrey@nasa.gov

***Abstract*—The Plan Execution Interchange Language (PLEXIL) is an open-source event-driven synchronous language developed by NASA to support autonomous spacecraft operations. This paper presents a workflow that integrates formal methods, including theorem proving and model checking, into the development process of the PLEXIL language. This workflow includes (1) iteratively updating the formalization of the language semantics to match the proposed changes and checking that significant language properties are preserved, (2) the automatic generation of a graphical representation of the semantics to check against the intended behavior, and (3) performing differential testing to check conformance of PLEXIL executions to the formalized PLEXIL operational semantics. The utilization of formal methods in the development process of the language aims to preserve semantic properties as new features are added to the language, improve traceability of language changes, and enable continuous development of both the language and tools used to perform validation and verification of PLEXIL programs.**

***Index Terms*—synchronous programming languages, formalization, formal methods, autonomous planning**

I. Introduction

Synchronous languages are suitable for programming reactive systems, i.e., systems whose behavior is determined as a reaction to environmental inputs. The Plan Execution Interchange Language (PLEXIL) [1] is an open-source event-driven synchronous language developed by NASA to provide flexible, efficient, and reliable plan execution in space mission operations. Programs in PLEXIL, called plans, specify actions to be executed by the system in response to external events happening in the environment of operation. PLEXIL has been used in space applications[1], aeronautics applications for unmanned aircraft operations [2], [3] and robotics applications [4].

Because of the safety- and mission-critical nature of these applications, PLEXIL was designed with formal methods in mind. Its operational semantics, i.e. the description of the processes that define the execution of programs in the language, has been formally defined in the Prototype Verification System (PVS) [5], an interactive theorem prover. This formalization is used to verify meta-theoretical properties of the language [6]. An executable semantics was also defined in the rewriting logic engine Maude [7]. This executable semantics serves as a reference implementation of the PLEXIL executive, an implementation of the PLEXIL language that provides a runtime environment for executing PLEXIL plans [8]. This executable semantics of PLEXIL is a core component of PLEXIL-V, a verification tool for PLEXIL plans [9]. As part of the workflow presented in this paper, PLEXIL-V is used to check conformance between the PLEXIL executive and the formal executable semantics via differential testing.

PLEXIL has constantly evolved since its original release. The current version of the language is PLEXIL 6. Because of this constant evolution, the PLEXIL language has grown organically, which has led to a state in which the language has increased in complexity and expressivity, but it has departed from the original intended semantics. This is true especially for very nuanced behaviors of the interactions between the different components at each of the different relational steps. This complexity leads to difficulty in both writing plans and model-checking those plans for desired properties. The effort described in the current work started as an attempt to remedy this difficulty.

This paper shows how the formalizations of the PLEXIL semantics in PVS and Maude drive the design of the language and help to uncover unexpected behaviors of the intended semantics. The use of formal methods allows for an agile

This work was funded under NASA's Certification for Autonomy Mission Environment (CARMEL) project. The authors are grateful to the PLEXIL development team and, in particular to Michael Dalal and Charles Fry, for the multiple discussions on the PLEXIL language that made this formalization work possible

[1]See https://plexil-group.github.io/plexil_docs/

language development workflow where the language and its formalization evolve together, and new features can be added without breaking fundamental properties of the language.

II. Related work

Integrating formal methods into the development cycle is a common practice when the resulting work necessitates a high level of assurance. Previous works describing the integration of formal methods into the development process focuses on the development of individual products and the formal verification process within that development cycle, see [10]. Many theoretical and pedagogical languages, such as WHILE [11], Guarded Command Language [12], and UNITY [13], were designed with a formal semantics in mind. Some industrial languages for developing high-reliability general purpose software, notably Eiffel [14] and SPARK/Ada [15], have a clear semantics and support a built-in *design by contract* principle, which enables formal verification and rigorous validation of programs written in those languages. From the family of reactive languages, Esterel [16], Lustre [17], and Signal [18] are synchronous languages with a well-defined semantics. Similar to these languages, the only non-determinism that is allowed in PLEXIL originates from external events occurring in the environment. PLEXIL differs from these languages in that its processes (called execution nodes in PLEXIL) do not communicate through data flows or signals but through shared memory. PLEXIL, similar to Eiffel and SPARK/Ada, allows plan developers to write contracts in the form of pre-/post-conditions and invariants. These conditions, called *check-conditions* in PLEXIL, are verified at run-time by the executive, but they can also be statically checked on user-specified environments through PLEXIL-V.

The types of workflows commonly used for integrating formal methods in development cycles are studied in [19]. In that work, the authors discuss different workflow styles of formal methods integration in development, which they call "after-the-fact", "parallel", and "integrated" approaches. The "after-the-fact" approach uses a standard development approach, and verification occurs after the system or product is complete. In this approach, formal verification is used to prove properties about the design of the system and potentially highlight inconsistencies in the design for further review. This approach, while common, is the costliest option, with an estimated 30 to 50 percent added cost to the development. Also, any errors caught by the formal verification require re-development of the system, which often means the formal specification done on the older version is no longer valid and needs to be redone. The "parallel" approach has both formal verification and development happening at the same time. This approach does require regular communication between the development team and the formalization team. The "parallel" approach speeds up the process of development but with two highly specialized teams, the cost could be high. In the "integrated" approach, as the name suggests, the formal verification effort is completely integrated into the system development process so that both development and specification efforts are one and the same. This workflow has its own unique challenges, mainly related to ensuring the development and formalization tools are highly integrated in such a way as to make this workflow even possible.

The current work concerns the development of the PLEXIL language and its formal semantics using two different notations. As explained in Section IV, this development has many interweaving components and, regarding the formal methods integration, follows a modified parallel approach. The use of formal methods here aims to make improvements to the language development process that ease the evolution of the language and its understandability by PLEXIL plan developers. During the development of a new version of the PLEXIL language, the formalization of the semantics is a key part of the workflow. The formalized semantics allows the PLEXIL developers to better understand changes to the original semantics and also ensure desired properties of the semantics persist through the changes in the language.

III. PLEXIL language background

PLEXIL is used to write plans for the execution of autonomous tasks in reactive systems. Intuitively, a PLEXIL plan reads the state of the external environment and send commands to actuators based on its internal state. These plans are written using different *nodes*, which contain data on the possible execution states and rules for execution and transition between the given states. The state of a node at any given point in the execution of a plan is one of the following: Inactive, Waiting, Executing, Finishing, Iteration_Ended, Failing, or Finished.

An operational small-step semantics is given for PLEXIL based on five relational steps:

- atomic, which describes the state transition of a single node,
- micro, the synchronous execution of the atomic steps of all nodes in the plan,
- quiescence, the iteration of micro steps until no more evolutions are possible,
- macro, which handles the external events and triggers another round of micro steps, and
- execution, which is the repeated iteration of macro steps.

In the nominal execution of a plan, nodes transition between the different states based on the semantics given by state transition diagrams. When no new transitions can occur in a micro step, the plan reaches quiescence. As the external environment changes, macro steps are induced and external environment information is read, and value updates are made visible to the internal state of the plan.

Since the semantics of PLEXIL is defined using internal state transitions, most of the behavior is understood at the atomic level where nodes transition from one state to the next. The intended semantics of these transitions are captured in node state diagrams. An example of this is given in Fig. 1, which illustrates the transition of an assignment node from the Executing state to either the Iteration_Ended state or the Failing state.

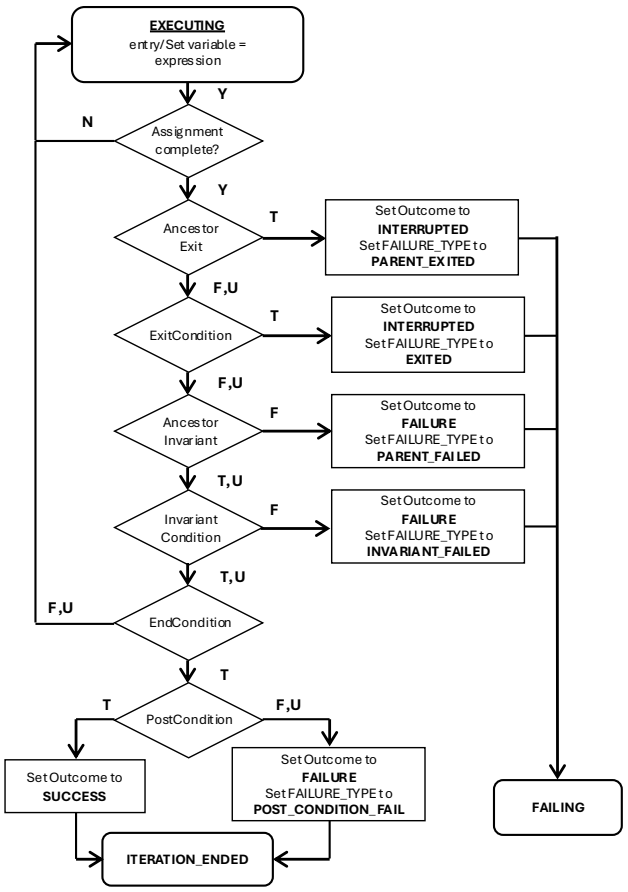

Fig. 1. Node state diagram for the intended semantics of an assignment node in the executing state.

The original semantics of the language was documented using flow chart diagrams. Historically, these graphical representations helped the PLEXIL development team agree on the intended behavior of the language and then guided the implementation of the PLEXIL executive. The same diagrams were used as the basis for the formalization of the language semantics in PVS and Maude discussed in the current work.

IV. Workflow

The setup of the project was initiated with two sub-teams. One sub-team consisted of the PLEXIL developers. This team, referred to as the *development team*, has members who are experts on the PLEXIL language, its development history, and experts in the practical use of PLEXIL and PLEXIL-V. The other sub-team, referred to as the *formalization team*, has experts on the formal semantics of the language in both PVS and Maude. They are also the developers of the PLEXIL-V toolset.

The high-level objectives were as follows:

1) Improve the verification technology for PLEXIL, in particular enhancing PLEXIL-V.
2) Make PLEXIL simpler, easier to use, and more intuitive.
3) Document the current semantics of the PLEXIL language and ensure that the intended semantics is reflected in the formal semantics in PVS and Maude.

The two sub-teams agreed that simplifying the semantics of the PLEXIL language would both improve its ergonomics and reduce its execution state space, thus making it more efficient to model-check. They also agreed that this simplification could be achieved by reducing the number of constructs in the language while preserving power and expressiveness. As will be seen in later sections, the process of thoroughly formally specifying the semantics helped the development team to uncover specific behaviors in the language that are not obvious even to expert users. This process was also used to find examples that highlight both the behavior of the language and its expressivity. These examples were used to discuss different semantic options and are intended to be used in regression tests as the language evolves. One of these examples is discussed in detail in Section V-C.

In addition to updating the formal semantics of PLEXIL, the verification team enhanced PLEXIL-V with a unified command line interface (CLI) that provides a simple point of entry to the following features:

- `plexilv-run`, a test runner that uses the executable semantics to run PLEXIL plans in batch mode.
- `plexilv-diff`, a differential testing tool that compares the execution of a batch script using plexilv-run and the standard PLEXIL executive-based test utility.
- `plexilv-check`, a utility to model-check PLEXIL plans over user-specified external environments.

To ease their adoption, the PLEXIL-V CLI utilities do not expose to their users the internal representation of PLEXIL plans in PLEXIL-V. Future improvements include lessening the requirement for familiarity with the Maude system. The development of these formalization tools is not the subject of this paper, but this work was done concurrently with the development of the PLEXIL language semantics.

The workflow presented in Fig. 2 summarizes the joint efforts for the development of PLEXIL and the formal semantics in PVS and Maude. The formalization in PVS is the early design verification step for the development of the PLEXIL language semantics. The development of the tools for verification are independent of the specific semantics development. The specification of the new semantics in Maude allows for a final round of differential testing to show adherence of new language features to the semantics specified in the formal verification tools within PLEXIL-V. This workflow is a combination of the "parallel" and "after-the-fact" workflows presented in [19]. That work goes into detail explaining that an integrated workflow is ideal for the development process in that an integrated workflow can both save time and reduce costs related to the development process. That type of workflow relies on a few conditions that are not present in the current effort. The system being specified would need to have a design that lends itself well to being fully specified with a formalization. This may be possible for PLEXIL and

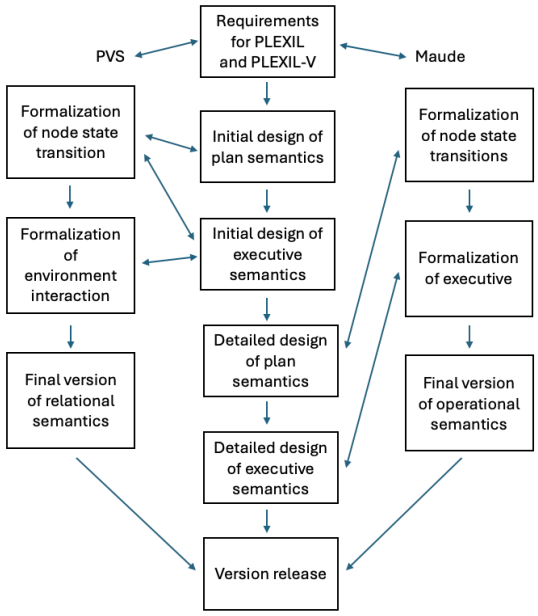

Fig. 2. Proposed workflow for the combined development efforts of PLEXIL and its two formalizations.

is the subject of future work. To do this, there would need to be a way to extract the formalization and make it executable. While possible for certain types of formalizations, that process is not suitable here given the size and complexity of the PLEXIL language. Another factor in this blended workflow is that the formalization tools are used to verify the functionality of the language in two different ways. The formalization of the semantics in PVS is the official formal specification of the language of PLEXIL, and the necessity of the formalized semantics in Maude is for the operation of the verification tools in PLEXIL-V. These tools allow plan developers to verify and test properties of specific plans. While conformance of all of these representations of the semantics is essential for the success of the effort described in this work, the two formal specifications serve different but complementary functions. A more detailed discussion of the unique features of each formal semantics can be found in Section VI.

V. PLEXIL LANGUAGE DEVELOPMENT CYCLE

The integration of formal methods into the development of a language is first and foremost used to guarantee the language satisfies its design specifications. The work began with the following design goals for the next version of the PLEXIL language semantics:

- Reduce the plan execution state space.
- Remove semantic thorns, e.g. ambiguities and complexities that cause difficulties in interpretation and implementation.
- Provide simpler repetition and iteration control features.

Part of the design specification for the upcoming version of PLEXIL is to make the semantics and behavior of the language easier to understand for practitioners. The language should be formalizable with a simplified semantics that keeps the current level of expressivity. Furthermore, divergence between the formal semantics and the intended semantics implemented by the PLEXIL executive should be eliminated. This leads to a design requirement for both the PLEXIL language and the formalization tools of PLEXIL that any future additions should be traceable in such a way that the language and its formalization evolve together.

In addition to the design specifications listed above, key properties of the PLEXIL semantics are essential to maintain for the new version of the language. The properties identified for the PLEXIL language semantics are as follows:

- Operational determinism, i.e., the language input/output relation is single valued but there is no assumption on the states of the external environment,
- Run to completion, i.e., the external environment triggers a chain of synchronous internal state changes that eventually stops,
- Operational synchrony, i.e., external events are synchronized at the level of macro steps.

Under well-specified assumptions, these semantic properties are verified in the PVS specification of the semantics for each iteration of the new language design. Any violation of these properties requires either semantic changes or a strong justification of the cases that elicit such behavior.

The original PLEXIL semantics aimed to satisfy all of these high-level design requirements. This initial semantics followed a set of state transition diagrams. The formalization effort first took place in PVS, which was intended to be the "sandbox" where new designs for the semantics can be specified, and the formally abstracted properties can be proved. Iteration on the semantics is informed by the high-level objectives, input from the PLEXIL development team, e.g., what does the semantics allow plan developers to express, and the formalization team, e.g., how the different choices for updates to the semantics affect high level properties of the language.

The rest of this section describes the process to maintain the intended and formal semantics in synchrony during the evolution of the language. It also provides two concrete examples that highlight how the workflow in Fig. 2 facilitates discussion between the development and verification teams that led to adding new features to the language while preserving its core properties.

A. Maintaining the intended and formal semantics in synchrony

The specification of the state transition diagrams in PVS, as seen in Fig. 3, looks very different from the state transition diagrams presented in the documentation of the PLEXIL semantics, Fig. 1.

As the development team addresses upgrades to the language, the state transition diagrams are updated as well. Maintaining consistency between these diagrams and the formal semantics is a time-consuming and error-prone activity. In order for the development and the verification teams to communicate effectively, an automatic translation between

```
% Status Executing (Node Assignment)
% ------------------------------------
executing_node_assignment(gamma:Context,pi:State)(p:(node_process?(pi)),
                                                   m:(memory_process?(pi)),
                                                   q:NodeProcess,
                                                   n:MemoryProcess): bool =
  status(p) = Executing AND
  node_assignment?(p) AND
  qid(p) = cdr(qid(m)) AND
  if ancestor_exit_true(gamma,pi)(p)
    then q = set_outcome_status(p,Interrupted(ParentExit),Failing) AND
         n = undo_value(m)
  elsif eval(gamma,pi)(exitc(p)) = TrueValue
    then q = set_outcome_status(p,Interrupted(ConditionExit),Failing) AND
         n = undo_value(m)
  elsif ancestor_invariant_false(gamma,pi)(p) then
    q = set_outcome_status(p,Failure(ParentFail),Failing) AND
    n = undo_value(m)
  elsif eval(gamma,pi)(inv(p)) = FalseValue then
    q = set_outcome_status(p,Failure(InvariantFail),Failing)  AND
    n = undo_value(m)
  else
    if (gamma,pi) |= endc(p) then
      n = m AND
      if (gamma,pi) |= post(p) then
        q = set_outcome_status(p,Success,IterationEnded)
      else
        q = set_outcome_status(p,Failure(PostconditionFail),IterationEnded)
      endif
    else
      false
    endif
  endif
```

Fig. 3. Formal semantics state transition logic for an assignment node in the executing state.

these two representations was developed. A pretty printer was developed that automatically produces Flow Chart and UML State Machine diagrams from the PVS formal semantic rules. The pretty printer is seamlessly integrated into VSCode-PVS [20], a modern Integrated Development Environment for PVS implemented as an extension for Microsoft's Visual Studio Code. Diagram specifications are generated in the Mermaid language, which supports several image formats and diagram editing tools. During the design of new language features, these diagrams are manually modified by the developer team. These modifications are then integrated into the formal semantics, which in turn is used to produce new diagrams. Divergence between the manually modified diagrams and those automatically produced from the formal specifications highlight potential ambiguities in the intended semantics, which are resolved iteratively.

B. Validation of PVS semantics

The formalization of the PLEXIL semantics in a theorem prover requires clearly defined behaviors. The process of formalizing leads to an investigation into details for the intended semantics which can highlight ambiguities in documentation. The following example gives insights into how the differing level of detail required in the PVS semantics and intended PLEXIL semantics led to iterative changes to the intended semantics of PLEXIL.

The state diagram in Fig. 1 informally specifies how assignment nodes behave in the Executing state. At the beginning of the Executing state diagram, an "assignment complete?" condition is checked. However, there is an ambiguity here in exactly when the assignment is intended to occur, e.g., before entering Executing or at the beginning of Executing. This ambiguity might lead to assumptions on the underlying semantics that do not match the implementation. The diagram given in Fig. 4, which was generated from the PVS specification, led to discussion with the PLEXIL development sub-team on the intended behavior of the assignment node semantics. It was clarified that assignments happen before transitioning to the Executing state.

Furthermore, during this continuing iterative discussion where the formal semantics was in development, it was noted that in the current implementation of PLEXIL, this assignment action forces termination of quiescence and the execution of a new macro step. This behavior is inconsistent with the event-based nature of the macro step since, in this particular case, an assignment of a variable, which is an internal behavior, is triggering a macro step. Further investigation of this issue revealed that the introduction of this forced macro step occurred to address the possibility of non-terminating quiescence for certain types of repetitions. The PLEXIL development team, which is designing a new version of the language, made several adjustments to the intended semantics, including the introduction of a new type of node to handle repetitions. These changes eliminate the need for the forced macro step in assignments and address the issue with non-terminating quiescence.

C. Validation of Maude Semantics

The Maude semantics is validated operationally by using it to interpret PLEXIL plans. More specifically, the `plexilv-run` tool implements a Maude semantics interpreter frontend similar to the official `plexiltest` tool to test and debug PLEXIL plans. The `plexilv-run` tool receives both a PLEXIL plan and an input environment script in the `PLEXILScript` format, and it executes the plan while producing as output the atomic, micro, and macro steps performed. The `plexilv-diff` tool automates the task of performing differential testing between `plexiltest` and `plexilv-run` by parsing their outputs and showing their differences.

Since the Maude semantics is so operationally close to the official PLEXIL test executive, it is used to explore and validate under-specified semantic features of PLEXIL. In the following, an example exploration of the PLEXIL semantics for reading external variables is presented.

Interactions between a PLEXIL plan and the external environment are captured by LookupNow expressions, which simply read external values into the internal state. The formal specification of this behavior relies on understanding how the PLEXIL executive interacts with the environment and when the read values are internally visible. The documentation of this behavior led to several discussions between the development and the verification team.

Consider the scenario in Fig. 5, where $\text{LookupNow}^n(T)$ denotes the n^{th} reading of the environment variable T, possibly from different nodes, during a plan execution.

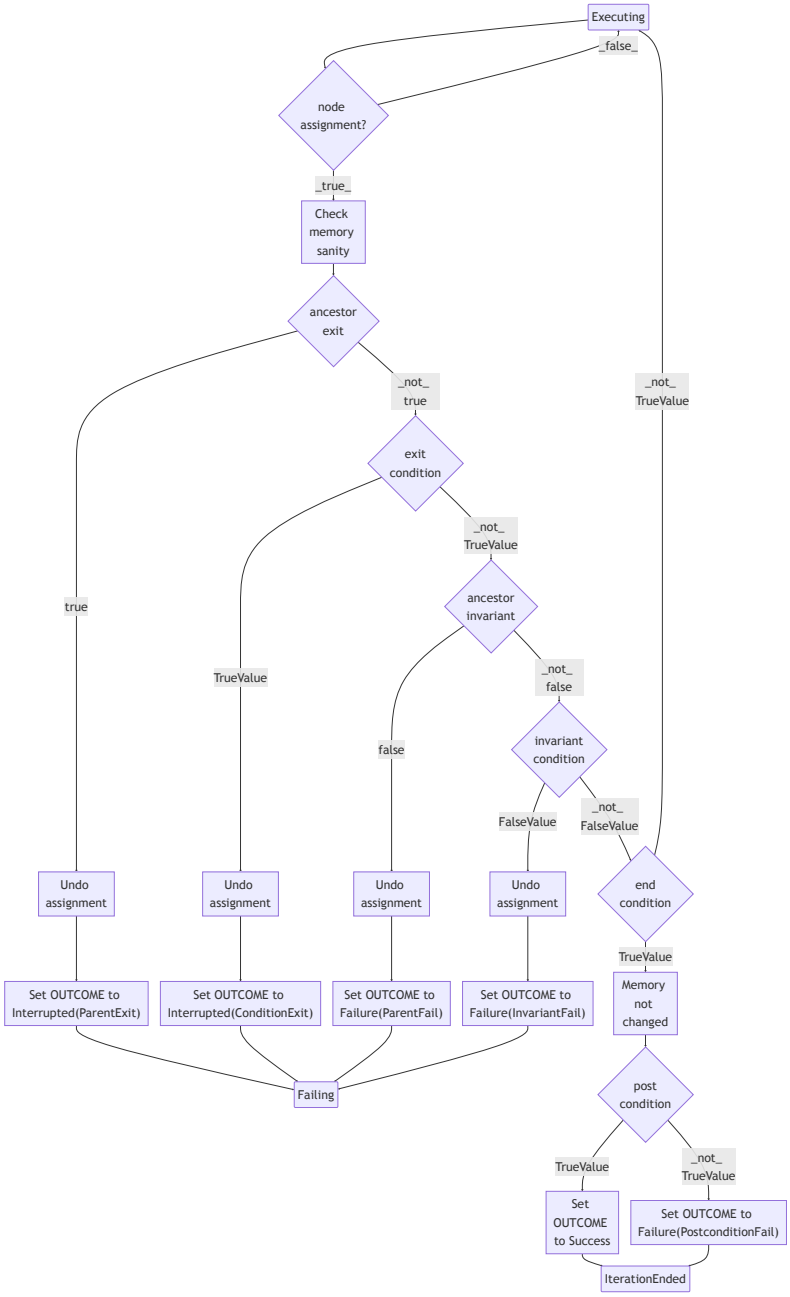

Fig. 4. Automatically generated node state diagram for an assignment node in the executing state from the formal semantics in PVS.

Henceforth, Γ represents a snapshot of the external environment that is visible to the internal state. Consider the following options for how the LookupNows in the scenario could be assigned given the relative positions of the real-time environment data assignment and the LookupNows in the plan execution.

In Option 1, shown in Fig. 6, values of external variables are cached in Γ at the beginning of the macro step. In other words, $\Gamma(T)$ is the value of T at the beginning of the macro step. Furthermore, during a macro step, all LookupNows of the same external variable return the same value. This semantics assumes that external variables used by the plan are declared, so that their values are cached at the beginning of the macro. If T is not declared, its LookupNow within the macro step is Unknown, which is a possible value, the first time the variable is read during an execution.

In Option 2, shown in Fig. 7, values of external variables are cached in Γ at first reading of the variable within a macro step. In other words, $\Gamma(T)$ is the value of T at first reading during the macro step. As in Option 1, during a macro step,

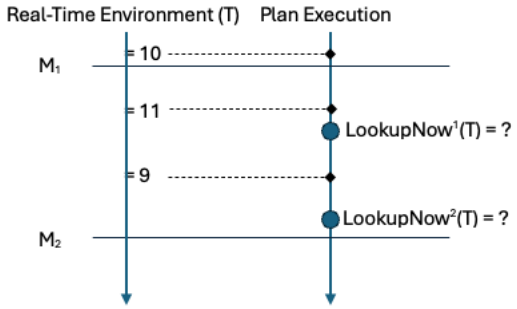

Fig. 5. A scenario with multiple LookupNows during one macro step execution.

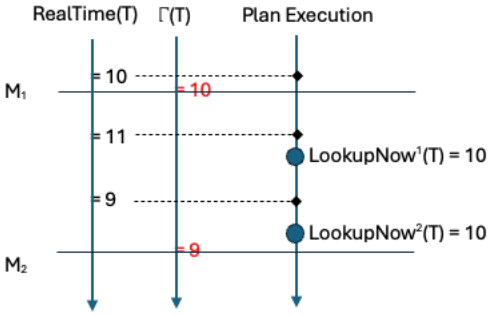

Fig. 6. In this option, the state of the environment is captured at the start of a macro step.

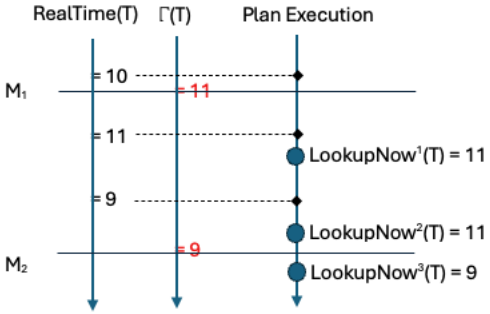

Fig. 7. The second option for how the LookupNow execution could behave. In this option, the state of the environment is updated to the first reading of the variable in the macro step.

all LookupNows of the same external variable return the same value. This approach does not require caching the value of T at the beginning of the macro step, so T could return a valid value at first reading even if it is undeclared.

In Option 3, given in Fig. 8, values of external variables are updated in Γ at each reading during a macro step. Therefore, different LookupNows of the same external variable during a macro step may see different values.

Due to the ambiguity of the existing documentation, all three interpretations of the LookupNow semantics presented above

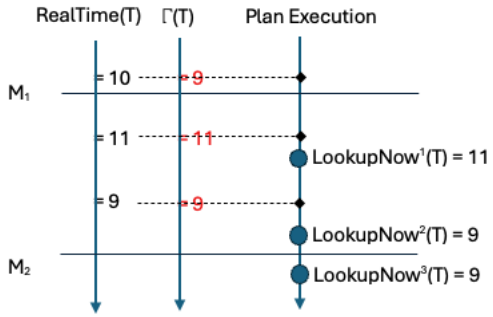

Fig. 8. The third option for how the LookupNow execution behaves. In this option, the state of the environment is continuously updated in the middle of a macro step.

are plausible. Option 1 aligns with the formal specifications in PVS and Maude. Options 2 and 3 would require a major overhaul of how the environment and its interaction with the internal state is formalized. Option 3, in particular, could be implemented by forcing a macro step whenever an environment variable is updated, but this change implies several changes to the micro and quiescence steps. Additionally, this option would violate the definition of a synchronous language, which is a key property of PLEXIL. Following discussions with the PLEXIL development team, test plans were developed to exercise these scenarios and subsequently added to the regression test suite. Collaborative exploration between the two teams confirmed that Option 1 represents the intended semantics. Consequently, the Maude semantics could be immediately and automatically validated against the test plans. Records of these conversations, the accompanying examples, and a definitive explanation of how LookupNows interact with both plan execution and the environment will be included in the documentation for the upcoming release.

VI. Why two semantics?

The semantics of PLEXIL as specified in PVS and Maude are very different in their nature. The formalized semantics of PLEXIL in PVS is specified using higher-order logic and defined in a layered structure through the five relational steps presented in Section III and illustrated in Fig. 9. This layered structure makes this semantics suitable for proving meta-theoretical properties of the PLEXIL language such as operational determinism, run to completion, and operational synchrony. Indeed, the key properties verified using PVS are proven at the atomic level and then lifted to the upper layers using abstract properties on relations [21].

On the other hand, the relational semantics in PVS is not executable. Furthermore, to verify correctness properties of PLEXIL plans using the PVS semantics, more low-level computational details would be needed, which would significantly complicate the PVS semantics and would break its compositional structure. For those reasons, an alternate formal semantics, operationally closer to the PLEXIL executive, was developed in Maude. This alternate semantics is executable

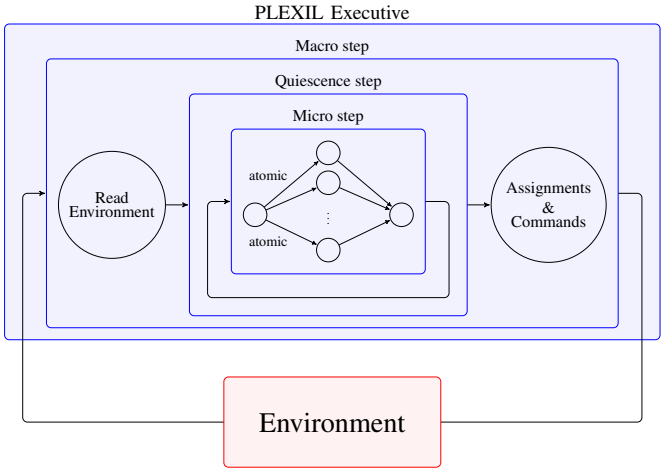

Fig. 9. The high-level view of the PLEXIL semantics in PVS.

and enables the use of model-checking for verifying concrete PLEXIL plans.

The PLEXIL semantics specified in Maude follows the structure of the *Rewriting Logic* framework [7], in which systems are specified as non-deterministic transition systems whose states are equivalence classes. These equivalence classes can be viewed as nested transition systems that represent the deterministic behaviors of a system. Transitions between equivalence classes model the non-deterministic behaviors of the system. Since the PLEXIL language has been proven to be operationally deterministic, the only non-determinism occurring during the verification of a plan originates from the environment in which the plan is executed. Hence, the Maude semantics provides a suitable framework to model the deterministic behaviors of PLEXIL plans under non-deterministic environments. This framework models non-deterministic environments by composing a set of environment-generator primitives that produce finite sets of non-deterministic options during execution. As depicted in Figure 10, while exploring the execution space of a PLEXIL plan during verification, the execution of each macro step will be deterministic (`Macro`$_i$ equivalence classes); the generation of potential environment inputs will be deterministic as well (`generateInputs`); but the inputs coming from the environment will be chosen non-deterministically (`chooseInput` transitions). This model of execution is closer to the PLEXIL executive and facilitates the interpretation of verification results by developers of PLEXIL plans.

VII. Conclusion and Future Work

By integrating formal methods into the development process of PLEXIL, new language features were implemented and divergence between the intended and formal semantics were identified through the formal specification process. These divergence created discussion between the development and formalization teams, aided by a common language of independently produced state diagrams. This discussion allowed

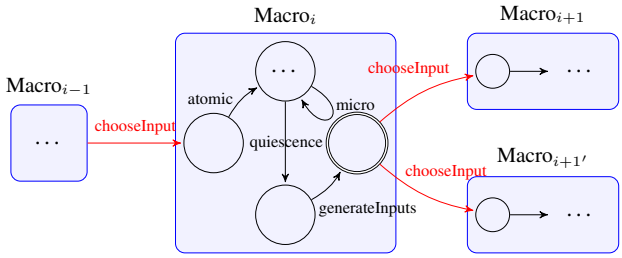

Fig. 10. The transition system view of the PLEXIL semantics in Maude.

for the development of strong reasoning to justify the changes as they were made. Through this integrated development workflow, the PLEXIL language and the PLEXIL executive evolve together while preserving core properties of the language.

The formalization of the semantics in PVS and Maude are both checked for conformance in different ways. The operational semantics in Maude is being checked by differential testing, and the PVS formalization checked using the adherence to the intended semantics by state diagram comparison. This leaves open the possibility of divergence between the intended semantics and the operational semantics. Future work includes the development of automated tools that bridge the gap of traceability and conformance between the different formalizations. This is a planned effort for the newest version of the PLEXIL language and PLEXIL-V.

References

[1] T. Estlin, A. Jonsson, C. Pasareanu, R. Simmons, K. Tso, and V. Verma, "Plan execution interchange language (PLEXIL)," NASA, Technical Memorandum TM-2006-213483, April 2006.

[2] S. Balachandran, C. Muñoz, M. Consiglio, M. Feliú, and A. Patel, "Independent configurable architecture for reliable operation of unmanned systems with distributed on-board services," in *Proceedings of the 37th Digital Avionics Systems Conference (DASC 2018)*, London, England, UK, September 2018.

[3] M. Lowry, A. Bajwa, T. Pressburger, A. Sweet, C. Fry, M. Dalal, J. Schumann, D. Dahl, G. Karsai, and N. Mahadevan, "Design considerations for a variable autonomy executive for UAS in the NAS," in *AIAA SciTech Forum*, 2018.

[4] H. F. Cadavid Rengifo and J. A. Chaparro Preciado, "Hardware and software architecture for plexil-based, simulation supported, robot automation," in *IEEE Colombian Conference on Robotics and Automation (CCRA)*, 2016.

[5] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *International Conference on Automated Deduction*. Springer, 1992, pp. 748–752.

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

[7] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott, *All About Maude - A High-Performance Logical Framework*, 1st ed., ser. LNCS. Springer, 2007, vol. 4350.

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

[9] C. Rocha, H. Cadavid, C. Muñoz, and R. Siminiceanu, "A formal interactive verification environment for the plan execution interchange language," in *International Conference on Integrated Formal Methods*. Springer, 2012, pp. 343–357.

[10] R. Chapman, C. Dross, S. Matthews, and Y. Moy, "Co-developing programs and their proof of correctness," *Communications of the ACM*, vol. 67, no. 3, pp. 84–94, 2024.

[11] H. Riis Nielson and F. Nielson, *Semantics with Applications*. John Wiley & Sons, 1999. [Online]. Available: http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.pdf
[12] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.
[13] K. M. Chandy, "Parallel program design," in *Opportunities and Constraints of Parallel Computing*. Springer, 1989, pp. 21–24.
[14] B. Meyer, *Eiffel: The Language*. Hemel Hempstead, England: Prentice Hall, 1992.
[15] B. Carré and J. Garnsworthy, "SPARK—an annotated ada subset for safety-critical programming," in *Proceedings of the Conference on TRI-ADA'90*, 1990, pp. 392–402.
[16] G. Berry, "The foundations of Esterel," in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, ser. Foundations of Computing, G. Plotkin, C. Stirling, and M. Tofte, Eds. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. [Online]. Available: https://inria.fr
[17] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1987, pp. 178–188.
[18] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming Real-Time Applications with Signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991. [Online]. Available: https://inria.hal.science/inria-00540460
[19] R. A. Kemmerer, "Integrating formal methods into the development process," *IEEE software*, vol. 7, no. 5, pp. 37–50, 2002.
[20] P. Masci and C. Muñoz, "An integrated development environment for the Prototype Verification System," *Electronic Proceedings in Theoretical Computer Science*, vol. 310, pp. 35–49, 2019.
[21] C. Rocha, C. Muñoz, and G. Dowek, "A formal library of set relations and its application to synchronous languages," *Theoretical Computer Science*, vol. 412, no. 37, pp. 4853–4866, 2011.