A Formal Analysis Framework for PLEXIL

Gilles Dowek, César Muñoz, and **Corina S. Păsăreanu**
Ecole Polytechnique, France, National Institute of Aerospace, USA, and Perot Systems/NASA Ames, USA
Gilles.Dowek@polytechnique.fr, munoz@nianet.org and pcorina@email.arc.nasa.gov

Abstract

The Plan Execution Interchange Language (PLEXIL) is a rich concurrent and reactive language developed by NASA to support autonomous commanding and monitoring for a variety of space systems. In this paper, we propose a semantic framework for the analysis of PLEXIL. In particular, the semantic framework allows for the formal study of properties such as determinism, compositionality, run to completion, termination, and stuttering, for different variants of the language. The framework is organized as a stack of abstract execution relations that define the small-step semantics of a synchronous event-driven language. This modular presentation of the language semantics enables the instantiation of the framework to different semantic variants of PLEXIL, and therefore, the framework serves as a formal designing tool. The mathematical development presented in this paper has been formalized and mechanically checked in the Program Verification System (PVS).

Introduction

Plan execution languages are specialized languages used for specifying control strategies that command and monitor a variety of systems such as spacecrafts, robots, instruments, and habitats. These languages vary in sophistication and in the degree of autonomy that they provide (Verma *et al.* 2005). The simplest systems execute linear sequences of commands at fixed times and provide little autonomy, while the most sophisticated systems use fully fledged programming languages to encode complex strategies, which take into account environment changes, and hence provide a high degree of autonomy. Autonomy is particularly critical for NASA missions, since NASA robots and spacecraft typically operate far from Earth so they must take significant responsibility for their own safe operation. However, the more complex a system is, the harder it is to predict properties about command execution, due also to the high uncertainty in the environment.

The Plan Execution Interchange Language (PLEXIL) (Estlin *et al.* 2005; Verma *et al.* 2006) is a high-level plan execution language, developed at NASA to support autonomous operations in a multi-platform environment. PLEXIL programs, called *plans*, specify actions to be performed by an *executive* system. A PLEXIL plan consists of set of processes, called *nodes*, organized in a tree structure. Each node is equipped with a set of conditions that trigger and monitor its execution. PLEXIL execution is *synchronous*, i.e., when conditions enable the execution of several nodes all of them are executed simultaneously, and *reactive*, i.e., nodes respond to unexpected changes in the environment. Indeed, PLEXIL belongs to the family of synchronous reactive languages such as Esterel (Berry 2000) and Signal (Guernic *et al.* 1991), where the only non-determinism allowed originates from the environment.

Typically, the PLEXIL executive is deployed on platforms with limited computational resources. For that reason, the language has been designed to be compact and semantically clear, and at the same time to be deterministic and compositional. Despite its relative simplicity, PLEXIL happens to be computationally complete, but this computational completeness is not as obvious as for other languages that include complex control structures as primitive features.

This paper presents a formal framework for specifying the small-step operational semantics of plan execution languages based on PLEXIL. This framework has been used as a designing tool for the developers of PLEXIL and as a reference for the implementation of the executive system. The framework is organized as a stack of five abstract relations, which range from an *atomic relation* describing the evolution of a single node at a moment in time to an *execution relation* describing the evolution of a plan and its local state after the occurrence of a series of events. In this layered architecture properties on the upper layers depend on abstract properties on the lower layers. Thus, the framework is robust to the evolution of the language.

We have worked closely with the PLEXIL development team to formulate and check key properties of the language, such as determinism, compositionality, run to completion, termination, and stuttering, for different assumptions on the language. Our formal framework helped in proving these properties and in pointing out to the developers some deficiencies, e.g., compositionality holds only under certain assumptions. The mathematical development presented in this paper has been formally specified and verified in PVS (Owre, Rushby, & Shankar 1992).[1]

[1]The PVS development is electronically available at http://research.nianet.org/~munoz/plexil.tgz.

A High Level Description of PLEXIL

PLEXIL plans are generated by automated planners or human operators, and are interpreted by an executive system that executes the commands encoded in the plan. The plan also encodes conditions that monitor the environment, react to changes in the environment, and feedback communication to higher-level decision making capabilities. PLEXIL and its executive system are still under development; their basic capabilities have been demonstrated on mid-size plans of up to 2000 lines of code.

The language has been designed to be syntactically simple but expressive enough to unify many execution languages and executive systems, and to be interfaced with a variety of automated planners. To support PLEXIL's role as an interchange language between multiple systems, the concrete syntax of the language is defined by an XMLschema. For simplicity, we phrase our discussion and examples in terms of a simplified non-XML notation.

Language Features

A PLEXIL plan consists of a hierarchical set of *nodes*, which execute concurrently and communicate through shared variables. The execution of each node is governed by a set of *conditions* (or *guards*) that encode logical and temporal relationships between nodes and the external environment.

Nodes. There are two kinds of nodes: *action nodes* that perform actions such as commanding systems, assignments to local variables, calls to pre-defined library functions, calls back to the planner, etc.; and *list nodes* that provide scope to a collections of nodes. Thus, a PLEXIL plan has a tree structure, where the leaves are action nodes, and the root and the internal nodes are list nodes. A list node is the *parent* of the nodes stored in its list, which form its *children*.

Conditions. The execution of each node is driven and monitored by a set of *conditions*. *Start* and *end* conditions specify when a node should start and end execution, respectively, *skip* conditions specify when the execution of a node can be skipped, and *repeat* conditions specify when the execution of a node should be iterated. *Pre* and *post* conditions are checked before and after each node execution, while *invariant* conditions are checked during node execution. When a pre, post, or invariant condition fails, the execution of the node is aborted and the node is marked as failed. In PLEXIL, there is a distinction between *gate* conditions, which are continuously monitored, e.g., start, end, skip, and invariant conditions; and *check* conditions, which are checked upon request during a node execution, e.g., pre, post, and repeat conditions.

Lookups. The execution senses the external world via *lookup* operations that read measurements from external variables, e.g., temperature, time, rover speed, etc. There are three types of lookups: `LookupNow` returns the latest value of an external variable, `LookupOnChange` returns a value only when it has changed according to a specified threshold, and `LookupWithFrequency` which returns a new value according to a specified frequency.

Variables. Each node contain a set of local variable declarations of type integer, boolean, float, string, or time. The scope of local variables is the sub-tree where they are declared. The domain of variables is extended with an additional value `Unknown` to account for undefined values. In particular, conditions are evaluated using a three valued logic over an extended Boolean domain `True`, `False`, and `Unknown`. In addition to the explicitly declared variables, each node has *implicit* variables, such as the *status* of the node execution, i.e., `Inactive`, `Waiting`, `Executing`, `Finishing`, `IterationEnded`, `Failing`, or `Finished`, and the node outcome, i.e., `Skipped`, `Success`, `Failure`.[2]

Control Structures. The only control structures provided by PLEXIL are the node conditions, e.g., start, end, skip, and repeat; and the tree structure of the plan, e.g., children nodes are activated in parallel by their parents. Standard programming control structures such as sequences, if-then-else, while-loops, etc., are not primitive in PLEXIL but they can be simulated using those basic features.

Example. Consider the following PLEXIL plan.

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

The plan consists of a list node `SafeDrive` that contains three action nodes: `OneMeter` invokes a command that drives the rover one meter, `TakePic` invokes a command that takes a picture, and `Counter` counts the number of pictures taken. The start condition of `TakePic` ensures that the node starts execution only after `OneMeter` finished and variable `pictures` is smaller than 10. The pre condition in `Counter` checks that no more than 10 pictures are taken. According to the repeat condition in `SafeDrive`, the action nodes are repeated until the rover is stuck. This information is requested from the environment via a lookup.

Informal Semantics

PLEXIL execution is driven by external events. The set of events includes events related to lookups in conditions, e.g., changes in the value of an external state that affects a gate

[2] In PLEXIL, the state of a node refers to its execution status. In this paper, the state of a node refers to all declared and implicit variables of the node.

condition, acknowledgments that a command has been initiated, 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 the next event is processed; this behavior is known as *run-to-completion* semantics. A macro step of execution consists of a number of *micro steps*. A micro step is the parallel synchronous execution of the *atomic steps* of the individual nodes. We discuss all these notions in more detail in the next section.

Semantic Framework

The semantic framework is defined in terms of mathematical structures that represent the external state of the world, the local state of the nodes, and the evolution of the external and local states during the execution of a PLEXIL program.

External State

The state of the world at a moment in time is represented by a set Σ of associations $\mathtt{X} = v$, where X is an external variable and v is its value. A plan does not have direct access to Σ, but to a local copy Γ, which is frequently updated from Σ. Variables in Γ are the same as in Σ, but the values may be different. Sets Σ and Γ are called *environments*. We say that an environment is *functional* if each variable appears only once in the environment, i.e, if $\mathtt{X} = v$ and $\mathtt{X} = w$ are both in the environment, then $v = w$. We assume that Σ and Γ are both functional environments.

We note that time is not a special concept in PLEXIL. Indeed, the current implementation of the PLEXIL executive time-stamps a node when the node is executed. In this case, we assume that there is variable `Time` in Σ that is accessed by the executive via lookup operations as any other external variable.

Example. The external state of the `SafeDrive` plan contains the external variable `Rover.WheelStuck` and interface variables related to the execution of commands `Rover.Drive` and `Rover.TakePicture`, e.g., `Rover.Drive.Completed`, when the command has been completed, and `Rover.Drive.Aborted`, if the command has been aborted.

Internal State

The internal state of a program is represented by a set of processes π with two types of processes: *node processes*, e.g., `Node A`, which contains the state information of a plan node A, and *memory processes*, e.g., `Var x=`v, which represents a local variable x with a value v.

A node process is a record where each field represents an implicit variable of the node: `identifier` (node's unique identifier), `priority` (node's priority), `start` (start condition), `end` (end condition), `skip` (skip condition), `repeat` (repeat condition), `pre` (precondition), `post` (postcondition), `inv` (invariant), `parent` (node's parent), `children` (node's children), `status` (execution status), `outcome` (outcome value), and `body` (node's body). For instance, in the case of an assignment node A, we have $\mathtt{A.body} = \mathtt{x} := e$. Except for `status` and `outcome`, all the other fields of a node state remain invariant during execution. Henceforth, we write `A with [`$\mathtt{a}_1 = v_1$`, ..., `$\mathtt{a}_n = v_n$`]` to denote the node process that is equal to A in all the fields except in $\mathtt{a}_1, \ldots, \mathtt{a}_n$, where it has the values $v_1, \ldots, v_n$, respectively.

We assume that π is *well-formed* in the following sense:

- Node identifiers are unique, i.e., `Node A` $\in \pi$, `Node B` $\in \pi$, and $\mathtt{A.identifier} = \mathtt{B.identifier}$, implies $\mathtt{A} = \mathtt{B}$.
- Local variables are uniquely defined, i.e., `Var x=`$v \in \pi$, `Var y=`$w \in \pi$, and $\mathtt{x} = \mathtt{y}$, implies $v = w$.
- Expressions are well-scoped, i.e., local variables appearing in conditions and assignments are declared in the node or in one of its ancestors.

Example. The internal state of the `SafeDrive` plan is represented by the set

$$\pi = \{ \; \mathtt{Node\ SafeDrive}, \; \mathtt{Node\ OneMeter}, \; \mathtt{Node\ TakePic}, \; \mathtt{Node\ Counter}, \; \mathtt{Var\ pictures} = 0 \; \}$$

In the initial state, it holds that $\mathtt{SafeDrive.status} = \mathtt{Waiting}$, $\mathtt{OneMeter.status} = \mathtt{TakePic.status} = \mathtt{Counter.status} = \mathtt{Inactive}$.

Expressions

The set of PLEXIL expressions is formed by constant values, pre-defined functions, basic Boolean and arithmetic expressions, local variables from π, and lookups on external variables from Γ. As we are interested in the high-level execution semantics of PLEXIL, we model the set of expression by an inductive abstract data type `Expression` with constructors for values, variables, lookups, and functional closures (i.e., pairs formed by a function and its arguments). The type of values is abstract, but we distinguish the values `True`, `False`, and `Unknown`.

Given environment Γ and program state π, the evaluation of expression e into value v in Γ and π is denoted $(\Gamma, \pi) \vdash e \rightsquigarrow v$. This relation is inductively defined on terms of the data type `Expression`:

$$\frac{}{(\Gamma,\pi) \;\vdash\; v \rightsquigarrow v}\ [\text{Val}] \qquad \frac{\mathtt{X} = v \;\in\; \Gamma}{(\Gamma,\pi) \;\vdash\; \mathtt{X} \rightsquigarrow v}\ [\text{Lookup}]$$

$$\frac{\mathtt{Var\ x}{=}v \;\in\; \pi}{(\Gamma,\pi) \;\vdash\; \mathtt{x} \rightsquigarrow v}\ [\text{Var}]$$

$$\frac{\begin{array}{c}(\Gamma,\pi) \;\vdash\; e_i \rightsquigarrow v_i, \text{ for all } 1 \le i \le n \\ (\Gamma,\pi) \;\vdash\; f(v_1,\ldots,v_n) = v\end{array}}{(\Gamma,\pi) \;\vdash\; (f, [e_1 \ldots e_n]) \rightsquigarrow v}\ [\text{Fun}]$$

By abuse of notation, we will write $(\Gamma, \pi) \vdash e \not\rightsquigarrow v$ to denote that expression e does not evaluate to v in (Γ, π).

Note that the set of expressions includes only one lookup rule. In the semantic framework presented in this paper we

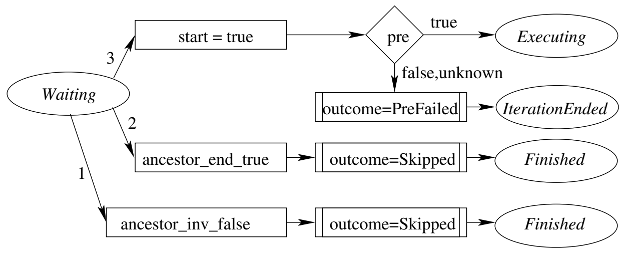

Figure 1: Transitions from `Waiting`

only consider `LookupNow` and `LookupOnChange` without a minimum threshold. These two operators are evaluated using the same Lookup rule. The modeling in our framework of the other lookup operators is still subject of research.

Execution

The very rich structure of the PLEXIL execution mechanism is specified using a *small-steps semantics* (Plotkin 1981) via five relations:

- The *atomic* relation, denoted by $\longrightarrow$, represents node state transitions and memory updates.
- The *micro* relation, denoted by $\Longrightarrow$, is defined as the synchronous execution of the atomic relation.
- The *quiescence* relation, denoted by $\Longrightarrow_{\downarrow}$, is defined as the run until completion of the micro relation.
- The *macro* relation, denoted by $\leftarrowtail\!\!\!\rightarrow$, describes how PLEXIL reacts to events in the external world.
- The *execution* relation, denoted by $\leftarrowtail\!\!\!\rightarrow^{n}$, is defined as the n-time step iteration of the macro relation.

We note that atomic and macro relations are the only relations that are specific to PLEXIL. The other three relations are based on well-known abstract relations. A technical detail is that the first four relations are *contextual*, i.e., they are defined on a global context. For the atomic and micro relations the context consists of the external environment Γ and the internal state π, for the quiescence relation the context consists of the external environment Γ, and for the macro relation the context consists of a family of external environments $(\Sigma_j)_{j\geq 0}$. If $\to$ is a contextual relation, we write $C \vdash a \to b$ to denote that the pair (a, b) belongs to the relation $\to$ under the context C.

Atomic Relation. The atomic relation defines the individual evolution of node and memory processes at a given time. Those evolutions were originally described by the designers of PLEXIL using an ad-hoc graphical notation (Estlin *et al.* 2005). For example, Figure 1 shows the state transition diagram for a node in status `Waiting`.

In our formal framework, we use a notation inspired on the Chemical Reaction Model (Banâtre & Métayer 1995), which is a formalism for modeling interactions between parallel processes. The atomic relation is written $(\Gamma, \pi) \vdash P \longrightarrow P'$, where $P \subseteq \pi$. For instance, the atomic rule that corresponds to the transition from `Waiting` to `Executing` in Figure 1 is written as follows.

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.start} \rightsquigarrow \texttt{True}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.pre} \rightsquigarrow \texttt{True}\\ \texttt{A.status} = \texttt{Waiting}\end{array}}{(\Gamma,\pi)\ \vdash\ \texttt{Node A} \longrightarrow \begin{array}{l}\texttt{Node A with}\\ \texttt{[status = Executing]}\end{array}}\ [\mathrm{W}_{3a}]$$

The rule that updates a memory cell of an assignment node whose status is *Executing* is written:

$$\frac{\begin{array}{l}\texttt{A.status} = \texttt{Executing}\\ \texttt{A.body} = \texttt{x := e}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \texttt{True}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \rightsquigarrow \texttt{True}\\ (\Gamma,\pi)\ \vdash\ e \rightsquigarrow w\end{array}}{\begin{array}{l}(\Gamma,\pi) \vdash\ \texttt{Node A}, \texttt{Var x:}v \longrightarrow \texttt{Node A with}\\ \quad\texttt{[status = IterationEnded},\\ \quad\ \texttt{outcome = Success]}, \texttt{Var x:}w\end{array}}\ [\mathrm{EA}_{3a}]$$

In total, 38 rules define the atomic relation. The rules are defined such that P is a singleton containing a process node, except for rules that update local variables such as EA_{3a}. In this case, P is a set of that contains the node process that writes the variables and the memory processes of the variables to be updated.

Micro Relation. The micro relation $\Gamma \vdash \pi \Longrightarrow \pi'$ is defined as:

$$\frac{\begin{array}{c}(\Gamma,\pi)\ \vdash\ P_1 \longrightarrow Q_1\\ \cdots\\ (\Gamma,\pi)\ \vdash\ P_n \longrightarrow Q_n\end{array}}{\Gamma\ \vdash\ \pi \Longrightarrow \pi \setminus \bigcup_{1\leq i\leq n} P_i \cup \bigcup_{1\leq i\leq n} Q_i}\ ,$$

where $\mathcal{M}_\pi = \{P_1, \ldots, P_n\}$ is the set of node and memory processes in π that are affected by the atomic relation. If two different processes in π, say A and B, write to the same variable, only the update of the process of higher priority is considered, e.g., A if `A.priority` > `B.priority`, B if `B.priority` > `A.priority`, and none otherwise.

Example. The following PLEXIL plan exchanges the values of the two variables `x` and `y`.

```
Node Exchange {
  int x = 0; int y = 1;
  List: { Node XY { Assignment: x:= y; }
          Node YX { Assignment: y:= x; }
        }
}
```

This behaviour is a consequence of the synchronous execution of atomic rule EA_{3a} during a micro step. In the case that two different nodes attempt to simultaneously write to the same variable, the conflict is solved by using the nodes' priorities.

Quiescence Relation. The run-to-completion execution semantics of PLEXIL states that micro steps are reduced until a stable state is reached. Formally, the quiescence relation $\Gamma \vdash \pi \Longrightarrow_{\downarrow} \pi'$ is defined as the $\Longrightarrow$-normalized reduction (Marché 1998) of π:

$$\frac{\begin{array}{c}\Gamma\ \vdash\ \pi \Longrightarrow^{*} \pi'\\ \pi' \text{ is a } \Longrightarrow\text{-normal form}\end{array}}{\Gamma\ \vdash\ \pi \Longrightarrow_{\downarrow} \pi'}\ ,$$

where $\Longrightarrow^{*}$ denotes the reflexive and transitive closure of $\Longrightarrow$. That π' is a $\Longrightarrow$-*normal form* means that π' cannot be reduced any further using the micro relation.

Macro Relation. The macro relation first updates Γ when an external event has occurred, and then starts a cycle of quiescence. In PLEXIL, an event occurs when one or more gate conditions are enabled. In the case of start and end conditions, the guard is enabled if it evaluates to `True`. In the case of invariant conditions, the guard is enabled if it evaluates to `False`.

In our formalism, we do not model events directly, but we assume an abstract predicate $\texttt{event?}(\Sigma, \Gamma, \pi)$ that triggers the update of values in Γ with those in Σ. Therefore, the macro relation is defined as follows:

$$\frac{\Gamma' = \begin{cases} \Sigma_i & \text{if } \texttt{event?}(\Sigma_i, \Gamma, \pi) \\ \Gamma, & \text{otherwise,} \end{cases} \qquad \Gamma' \vdash \pi \Longrightarrow_{\downarrow} \pi'}{(\Sigma_j)_{j\geq 0} \vdash (i, \Gamma, \pi) \looparrowright (i+1, \Gamma', \pi')},$$

where i is a time step and $(\Sigma_j)_{j\geq 0}$ is a sequence of environments.

Intuitively, the environment Σ_i can be understood as the state of the world before the i-th macro step. The PLEXIL executive optimizes the access to the external world by reading the external variables the first time they are needed during a macro step. These values are cached such that further lookups during the same macro step return the same values. In this case, Σ_i contains the values of the external variables at the time when they are first read during the i-th macro step.

We assume that changes to the external world are visible to the executive only at the beginning and end of macro steps. In practice, the executive enforces this assumption by queuing events and processing them one at a time at the end of each macro step. As we will see, except for compositionality, none of the properties we have verified requires a concrete definition of `event?`. Hence, those properties still hold for variants of PLEXIL where an event always occurs after a macro step or when Γ is updated only when Σ changes.

Execution Relation. The execution relation $\looparrowright^n$ is the n-iteration of the macro relation from the initial state $(0, \Sigma_0, \pi)$.

Properties

This section presents the main theoretical properties of our semantic framework for PLEXIL and other variants of PLEXIL. All the properties presented in this section were formally checked in PVS.

Determinism Under Full-Knowledge Hypothesis

If we assume that the sequence of readings of the external world is known *in advance*, then the behavior of a plan can be predicted. We call that assumption the *Full-Knowledge Hypothesis*.

Theorem 1 (Determinism) *Let $(\Sigma_j)_{j\geq 0}$ be a sequence of environments. For all n, if*

1. $(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \pi) \looparrowright^n (n, \Gamma', \pi')$, *and*
2. $(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \pi) \looparrowright^n (n, \Gamma'', \pi'')$,

then $\Gamma' = \Gamma''$ and $\pi' = \pi''$.

In practice, the full-knowledge hypothesis is seldom achieved in space applications due to uncertainties in the external world. However, determinism under full-knowledge hypothesis is the property that guarantees that the behavior of a PLEXIL plan is reproducible in a simulation environment. Indeed, if the readings of the external world are recorded during deployment, determinism states that the behavior of the deployed plan is the same as the behavior of the plan executed in a simulation environment that replays the recorded readings.

Operational Determinism

From an operational point of view, it is important to constrain the non-determinism to interactions with the external environment. Operational determinism states that in absence of external events, the behavior of a PLEXIL plan can be predicted.

We say that a sequence of environments $(\Sigma_j)_{j\geq 0}$ is n-*sterile* if for all $k \leq n$, $(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \pi) \looparrowright^k (k, \Gamma', \pi')$ implies $\neg\texttt{event?}(\Sigma_k, \Gamma', \pi')$.

Theorem 2 (Operational Determinism) *Given two n-sterile sequences of environments $(\Sigma_j)_{j\geq 0}$ and $(\Omega_j)_{j\geq 0}$ such that $\Gamma = \Sigma_0 = \Omega_0$, if*

1. $(\Sigma_j)_{j\geq 0} \vdash (0, \Gamma, \pi) \looparrowright^n (n, \Gamma', \pi')$, *and*
2. $(\Omega_j)_{j\geq 0} \vdash (0, \Gamma, \pi) \looparrowright^n (n, \Gamma'', \pi'')$,

then $\Gamma' = \Gamma''$ and $\pi' = \pi''$.

Note that operational determinism does not require that the readings of the external world are known in advance as in the case of determinism under full-knowledge hypothesis. For operational determinism it is enough that the external world does not generate events. Furthermore, operational determinism is neither weaker nor stronger than determinism. Indeed, it is possible to instantiate our framework such that the resulting language is deterministic under full-knowledge hypothesis, but is not operationally deterministic.

Example. Consider the following plan that performs in sequence a lookup on an external variable, a finite iteration, and a lookup on the same variable:[3]

```
Node Sequence {
  int tempA = 0;
  int tempB = 0;
  List: {
    Node A {
      Assignment: tempA := LookupNow(Temp);
    }
    Node Loop {
      int x = 0;
      Start: A.status == Finished;
      Repeat-while: x < 10;
      Assignment: x := x + 1;
    }
    Node B {
      Start: Loop.status == Finished;
      Assignment: tempB := LookupNow(Temp);
    }
    Node C {
      Start: B.status == Finished;
      Pre: tempA == tempB;
    }
  }
}
```

In PLEXIL, all the actions in `Sequence`, including the full iteration in `Loop`, are performed during the quiescence cycle in the first macro step. The plan finishes in a state where the precondition of node `C` holds.

Now, assume that the macro relation is defined as follows:

$$\frac{\Gamma' = \Sigma_i \qquad \Gamma' \vdash \pi \Longrightarrow \pi'}{(\Sigma_j)_{j\geq 0} \vdash (i, \Gamma, \pi) \looparrowright (i+1, \Gamma', \pi')},$$

[3]This example has been extensively discussed by the PLEXIL team. The version presented here was provided by Michael Iatauro.

i.e., Γ is always updated and the quiescence cycle is avoided. We call this semantics *updated step-by-step*. Under this semantics, every assignment in node `Loop` takes at least one macro step. Therefore, when the node `B` is finally executed, the value of the external variable `Temp` is likely to be different from the saved value `tempB`. Hence, the precondition of the node `C` does not necessarily hold.

We can verify that the language defined by the updated step-by-step semantics *does* satisfy determinism under full-knowledge hypothesis, but *does not* satisfy operational determinism .

Run-to-Completion

The *run-to completion* semantics of PLEXIL states that when a quiescence cycle terminates, it reaches a stable state. In particular, if no event occurs, this state remains invariant under the macro relation.

Theorem 3 (Run-to-Completion) *Let* $(\Sigma_j)_{j\geq 0}$ *be a sequence of environments. If* $(\Sigma_j)_{j\geq 0} \vdash (i, \Gamma, \pi) \star\!\!\longrightarrow (i+1, \Gamma', \pi')$ *and* $\neg$`event?`$(\Sigma_{i+1}, \Gamma', \pi')$*, then*

$$(\Sigma_j)_{j\geq 0} \vdash (i+1, \Gamma', \pi') \star\!\!\longrightarrow (i+2, \Gamma', \pi').$$

A natural question that arises from this definition is if a macro step, or for that matter, a quiescence cycle, always terminates, i.e., if for all Γ and π, there is a π' such that

$$\Gamma \vdash \pi \Longrightarrow_{\downarrow} \pi'.$$

This is not always the case as illustrated by the following trivial example. Since the repeat condition is always true, the first quiescence cycle does not terminate.

```
Node InfiniteLoop {
  int x = 0;
  Repeat-while: x >= 0;
  Assignment: x := x + 1;
}
```

Termination

In order to gain termination of the quiescence relation, it may be necessary to constrain the run-to-completion semantics. Consider the following instantiations of our semantic framework.

- *Step-by-Step Semantics*: The quiescence relation is defined as the micro step, i.e., $\Longrightarrow_{\downarrow} \equiv \Longrightarrow$.
- *Broken-Quiescence Semantics*: The quiescence relation is defined such that a repeating node goes from execution status *Waiting* to *Waiting* a pre-specified number of times during the quiescence cycle.

We can verify that these two semantic variants of PLEXIL yield languages that satisfy determinism, operational determinism, and termination of macro steps. The step-by-step semantics differs from the updated step-by-step semantics in the way Γ is updated. By ignoring changes to Σ during a macro step, we manage to remain deterministic under full-knowledge hypothesis without having a zero-time assumption. By keeping a local copy Γ of the external environment Σ, we gain operational determinism. On the other hand, it can be easily checked that the step-by-step semantics and the broken-quiescence semantics do not satisfy the run-to-completion property

Stuttering

The stuttering property states that if the environment does not change the state of a program does not evolve. Due to the run-to-completion semantics of PLEXIL, if Σ does not change after a macro step, the next macro step is empty. Further macro steps are empty until, eventually, Σ changes and enables a non-empty macro step. We say that a sequence of environments $(\Sigma_j)_{j\geq 0}$ is n*-constant* if for all $k \leq n$, $\Sigma_k = \Sigma_0$.

Theorem 4 (Stuttering) *Let* $(\Sigma_j)_{j\geq 0}$ *be a sequence of environments that is* $n+1$*-constant. If*

1. $(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \pi) \star\!\!\longrightarrow (1, \Gamma', \pi')$*, and*
2. $(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \pi) \star\!\!\longrightarrow^{n+1} (n+1, \Gamma'', \pi'')$,

then $\Gamma' = \Gamma''$ *and* $\pi' = \pi''$.

Compositionality

The compositionality property states that the execution of parallel processes can be inferred from the independent execution of each one of them. We have verified that PLEXIL is compositional under particular assumptions on the concrete definition of `event?`.

Two sets of processes P and Q are said to be *non-overlapping* if their corresponding sets of identifiers, e.g., node identifiers and name of local variables, are disjoint. A predicate `event?` is *process independent* if `event?`(Σ, Γ, P) = `event?`(Σ, Γ, Q) for any Σ, Γ, and non-overlapping P, Q.

Theorem 5 (Compositionality) *Let* $(\Sigma_j)_{j\geq 0}$ *be a sequence of environments, and* π *and* χ *be two different PLEXIL programs. For any concrete definition of* `event?` *that is process independent, if*

1. $(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \pi) \star\!\!\longrightarrow^{n} (n, \Gamma, \pi')$*, and*
2. $(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \chi) \star\!\!\longrightarrow^{n} (n, \Delta, \chi')$,

then

$$(\Sigma_j)_{j\geq 0} \vdash (0, \Sigma_0, \pi \cup \chi) \star\!\!\longrightarrow^{n} (n, \Gamma \cup \Delta, \pi' \cup \chi')$$

We have identified at least two concrete definitions of `event?` that satisfy the property of being process independent.

1. `event?`$(\Sigma, \Gamma, \pi) =$ `True`.
2. `event?`$(\Sigma, \Gamma, \pi) = (\Sigma \neq \Gamma)$.

The first definition states that an event always occurs at the end of a macro step. The second definition states that an event occurs if Σ has evolved. Actually, the PLEXIL executive implements an optimization of the second definition that first checks if the external variables that have changed occur in the program or not. A priori, this optimized implementation of `event?` does not satisfy the process independent property. We are currently working on a weaker assumption on `event?` that guarantees compositionality for a larger set of non-overlapping processes.

Conclusion

We have defined a modular semantic framework where several variants of PLEXIL can be formally analyzed. In practice, this framework has contributed to the design of the PLEXIL language. Indeed, we worked with the PLEXIL team to guarantee that the language has a clear semantics and that the intended and operational semantics coincide. For instance, the definition of a language that satisfies both operational determinism and termination of macro steps has been the object of an intensive debate during the development of PLEXIL. Our formal analysis has shed light on this issue and, consequently, we propose in this paper a few semantic variants of PLEXIL where both properties hold.

PLEXIL is evolving and being able to formally check properties of variants of the language was a goal of our development. This justifies the modular approach with several relations separating the description of computations, parallel composition, and interaction with the environment. We can prove properties of one layer based on assumptions on the other layers. The entire framework was specified and mechanically verified in PVS, allowing us to reach the highest degree of certainty in the proof-checking. It consist of 11 theories. Most of these theories are specific to PLEXIL, but we have also developed general theories on abstract relations such as normalized reductions and synchronous reductions with priorities. In total, these 11 theories include 172 lemmas and 1523 lines of specification.

From a verification point of view, the contribution of this work has several dimensions. First, the semantic framework is fundamental to the theoretical study of PLEXIL and, thus, to the understanding of the the language and its features. Second, it enables the verification that a particular PLEXIL executive correctly implements the language. Another application of this semantics is the ability to predict the behavior of a plan, or a family of plans, under particular scenarios. As a matter of fact, based on this semantics, we are working on the developement of a verification tool that automatically checks for properties that encode requirements such as "no node is repeated after the plan is paused or aborted". Finally, a pre-requisite for a definition of *correctness* in a plan execution language is the availability of a formal execution semantics. It is in this area that we will focus our future research. In particular, the design of an appropriate specification logic for PLEXIL where safety and liveness properties can be naturally written and mechanically checked.

Although the work presented in this paper concerns the PLEXIL language, we believe that the general concepts of our framework, such as the multi-layered semantics, is applicable to other synchronous languages that have many semantic variations, such as Statecharts. This work, among others, suggests that formal methods are now mature enough to be used for programming languages design and in engineering domains, such as aerospace, where safety is a major issue.

Acknowledgments

We are thankful to the PLEXIL group and, in particular, to Vandi Verma and Mike Iatauro, for providing invaluable insights on the intended semantics of PLEXIL.

References

Banâtre, J.-P., and Métayer, D. L. 1995. Gamma and the chemical reaction model. In *Proceedings of the Coordination '95 Workshop*. Londres: IC Press.

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

Estlin, T.; Jónsson, A.; Păsăreanu, C. S.; Simmons, R.; Tso, K.; and Verma, V. 2005. Plan Execution Interchange Language (PLEXIL). NASA Technical Memorandum.

Guernic, P. L.; Gautier, T.; Borgne, M. L.; and Maire, C. L. 1991. Programming real-time applications with SIGNAL. In *Proceedings of the IEEE, Volume 79(9)*, 1321–1336.

Marché, C. 1998. Normalized Rewriting: an unified view of Knuth-Bendix completion and Gröbner bases computation. *Progress in Computer Science and Applied Logic* 15:193–208.

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

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

Verma, V.; Jónsson, A.; Simmons, R.; Estlin, T.; and Levinson, R. 2005. Survey of command execution systems for NASA spacecraft and robots. In *Plan Execution: A Reality Check Workshop at the International Conference on Automated Planning and Scheduling (ICAPS)*.

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