A SMALL-STEP SEMANTICS OF PLEXIL[*]

Gilles Dowek[†], César Muñoz[‡], and Corina Păsăreanu[§]

ABSTRACT

A key design principle of a high-level plan execution language is the ability to predict the output of a planning task under limited information on the external environment and on the platform where the plan is executed. This type of determinism is particularly critical in unmanned space exploration systems, such as robotic rovers, which are often deployed in a semi-autonomous mode. This paper proposes a formal framework for specifying the semantics of general plan execution languages based on NASA' Plan Execution Interchange Language (PLEXIL), a rich concurrent language that is being developed in support of NASA's Space Exploration Program. The semantic framework allows for the formal study of properties such as determinism for different assumptions on the knowledge of the external environment. The framework is organized as a stack of parametric layers defining the execution mechanism of a parallel synchronous event-driven language. This modular presentation enables the instantiation of the framework to different semantic variants of PLEXIL-like languages. The mathematical development presented in this paper has been formalized and mechanically checked in the Program Verification System (PVS).

1 INTRODUCTION

The Plan Execution Interchange Language (PLEXIL) [9] is a high-level plan execution language, recently developed at NASA to support autonomous spacecraft operations in a multi-platform environment. PLEXIL is a parallel language for specifying actions to be executed by an autonomous system. These actions can be part of normal spacecraft operations or they can respond to unexpected changes in the environment. Programs written in this language are *reactive* in the sense that they maintain a permanent interaction with their environment.

In the family of reactive languages, plan execution languages belong to the sub-family of synchronous languages such as Esterel [5], Lustre [7] and Signal [10], where the only non-determinism allowed originates from external events occurring in the environment. Plan execution languages, such as PLEXIL, differ from these general purpose reactive languages in that their processes (called *execution nodes* in PLEXIL) do not receive or emit signals. In PLEXIL, for example, each node is equipped with a set of conditions that trigger and monitor execution. These conditions enable nodes to react to changes in the environment, to communicate with other nodes via shared variables, and to enter specialized modes of operation in case of failures.

From an operation point of view, a PLEXIL node has a very simple structure as it can only do three type of things: initiate an external activity (such as start an engine), assign

[*]Authors in alphabetical order. This work was supported by the National Aeronautics and Space Administration, Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043

[†]École Polytechnique, France. Email: Gilles.Dowek@polytechnique.fr.

[‡]National Institute of Aerospace, USA. Email: munoz@nianet.org.

[§]QSS/NASA Ames, USA. Email: pcorina@email.arc.nasa.gov.

a local variable, or provide scope for children nodes. Thus, PLEXIL programs (also called *plans*) are structured into a tree, analogous to the command trees used by human operators to monitor and control spacecraft operations. Despite its relative simplicity, PLEXIL happens to be computationally complete, but this computational completeness is not as obvious as for other languages that include sequences, loops, etc., as primitive features.

This paper contributes with a presentation of the formal semantics for a family of plan execution languages based on PLEXIL. The semantics is meant as both a reference for implementations and a framework to formally prove several properties of the language such as various forms of determinism. As we shall see, these plan execution languages provide a unique combination of synchronous execution and shared variable communication that ensures deterministic execution.

The formal semantics is a small-step operational semantics and it is organized in several layers ranging from the definition of an *atomic relation* describing the evolution of a single node during a single time step to a *macro relation* describing the evolution of the plan and its environment after the occurrence of an event. This layered architecture is the most important feature of our semantic framework, as many properties of the language depend only on the definition of the upper layers and are therefore robust to the continuous evolution of the language.

The mathematical development presented in this paper has been formally specified and mechanically verified in the Program Verification System (PVS) [15], allowing us to reach the highest degree of certainty in the proof-checking.

The rest of the paper is organized as follows. Section 2 gives an informal presentation of PLEXIL. Section 3 presents the semantic framework and Section 4 discusses its properties for several semantic variants of PLEXIL-like languages. Section 5 discusses related work and Section 6 concludes the paper.

2 A HIGH LEVEL DESCRIPTION OF PLEXIL

PLEXIL is an execution language used for writing plans (control strategies) to command and control a variety of complex systems such as spacecraft, robots, instruments, and habitats. PLEXIL plans are generated by high level automated planners or human operators, and are interpreted by an *executive system* that executes the commands encoded in the plan, monitors the conditions in the environment, reacts to environment changes, and sends feedback to the higher-level decision making capabilities.

PLEXIL is akin to a concurrent reactive programming language: it can represent branches, loops, time- and event- driven activities, concurrent activities, sequences, and temporal constraints. PLEXIL has been designed to be simple (it has few syntactic constructs) but expressive enough to unify many existing execution languages and systems, and hence to be interfaced with a variety of existing automated planners.

2.1 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 consist of collections of execution 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, while *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. If any of these conditions fail, then the node execution is aborted (and the node is marked as failed).

Note that in PLEXIL there is a distinction between *gate* conditions, which are continuously monitored, e.g., start, end, 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 values (measurements) from external state variables, e.g., temperature, time, engine status, etc. We note that, in PLEXIL, time is treated like any other external variable.

There are three types of lookup operations: `LookupOnChange` returns the current value of the state variable immediately and then it monitors the state variable and repeatedly returns its value whenever it changes (according to some specified minimal change threshold). `LookupWithFrequency` is similar to `LookupOnChange` except that it repeatedly returns the value of the state variable according to a specified frequency. `LookupNow` simply returns the current value of the state variable. `LookupOnChange` and `LookupWithFrequency` are only used in gate conditions, while `LookupNow` is used in check conditions and in assignments.

Variables

Each node may 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. For example, in the PLEXIL assignment
```
temp := LookupNow(Engine.temperature)
```
the value of `temp` is set to *unknown* if the sensor for reading the temperature of the engine fails. Moreover, conditions are evaluated using a three valued logic over an extended Boolean domain $\{$*true, false, unknown*$\}$.

In addition to the explicitly declared variables and the external state variables, a PLEXIL plan has also access to some *implicit* node variables, such as the status of a node execution (*Waiting*, *Executing*, *Finished*, etc.) and the node outcome (*Skipped, Success, PreFailed,PostFailed,InvFailed,ParentFailed*).[1]

Note that standard programming constructs such as sequences, if-then-else, while-loops, etc., are not primitive in PLEXIL. However, they can be simulated using the basic constructs: concurrent nodes and conditions. To support PLEXIL's role as an interchange language between multiple planners and execution systems, the concrete syntax of the language is defined by an XML [6] schema. For simplicity, in the rest of the paper, we phrase our discussion and examples in terms of a simplified (non-XML) notation for a PLEXIL-like language, which should be sufficient for our presentation.

2.2 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 &&
               pictures < 10;
     Command: Rover.TakePicture();
    }
    Node Counter {
     Start: TakePic.status == FINISHED;
     Pre: picture < 10;
     Assignment: pictures := pictures + 1;
    }
  }
}
```

This plan consists of a list node named `SafeDrive` that contains three action nodes: `OneMeter`, which invokes a command that drives the rover one meter, `TakePic`, which invokes a command that takes a picture, and `Counter`, which counts the number of pictures that have been taken. The start condition of node `TakePic` ensures that the node starts execution only after node `OneMeter` finished and local variable `pictures` is strictly smaller than 10. A pre condition in the node `Counter` ensures that no more than 10 pictures are taken. As specified by a repeat condition, the list node keeps repeating the command nodes until the rover is stuck. The repeat condition requests information from the functional layer via a lookup.

[1]In PLEXIL, the execution status of a node is referred as its *state*. In this paper, the state of a node refers to all the implicit variables of the node including the execution status and the node outcome.

2.3 Informal Semantics

The execution in PLEXIL 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 condition, acknowledgments from the functional layer 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 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*. Each micro step corresponds to the parallel synchronous execution of the *atomic steps* of the individual plan nodes. We discuss all these notions in more detail below.

Atomic Steps

The execution semantics of an individual PLEXIL node is specified in terms of its *execution status*: *Inactive*, *Waiting*, *Executing*, *Finishing*, *IterationEnded*, *Failing*, or *Finished*. During the plan execution, nodes change their state, including their execution status, in *atomic steps*.

At the beginning of plan execution, the status of each node is initialized to *Inactive*, except the root node for which the status is set to *Waiting*. A node transitions from *Waiting* to *Executing*, when its start condition becomes *true* (and its pre condition also evaluates to *true*). When a list node changes status to *Executing*, all its children are activated, i.e., they are set to *Waiting*. When an action node has status *Executing*, it performs its action, i.e., execute an assignment or issue a command; upon completion of the action the status of the node is changed to *IterationEnded*. Furthermore, the node's status can change back to *Waiting*, if the repeat condition evaluates to *true*, or to *Finished*, if the repeat condition evaluates to *false* or *unknown*. The execution of list nodes proceeds similarly, except for the extra status *Finishing*, which allows the node to wait for its children to finish execution. Explicit end conditions can also be used to instruct node termination.

A node fails its execution if one of its pre, post, or invariant condition is violated; a node may also fail if one of its ancestors has failed. Whenever a failure occurs, an active action node aborts execution while an active list node changes status to *Failing* (and waits for all its children to abort execution). The *outcome* of the node records whether the execution of the node for the current iteration was skipped, successful or failed; the outcome is reset when a node repeats execution.

Node conditions are checked only in the states where they apply. For example, start conditions are monitored only when the node status is *Waiting*, while end conditions are monitored only when the node status is *Executing*. A detailed description of the atomic step relation is provided in terms of an ad-hoc notation for state transition diagrams in [9].

Figure 1 shows the state transition diagram for status *Waiting* (as presented in [9]). Gate conditions are shown in rectangles, i.e., `start` = *true*, `ancestor_end_true`, and `ancestor_inv_false`, while check conditions, i.e., `pre`, are depicted in diamonds. A precedence order on condition changes is used to resolve conflicts. For example, if the status of a node is *Waiting* and both the start condition and an ancestor end condition become *true*,

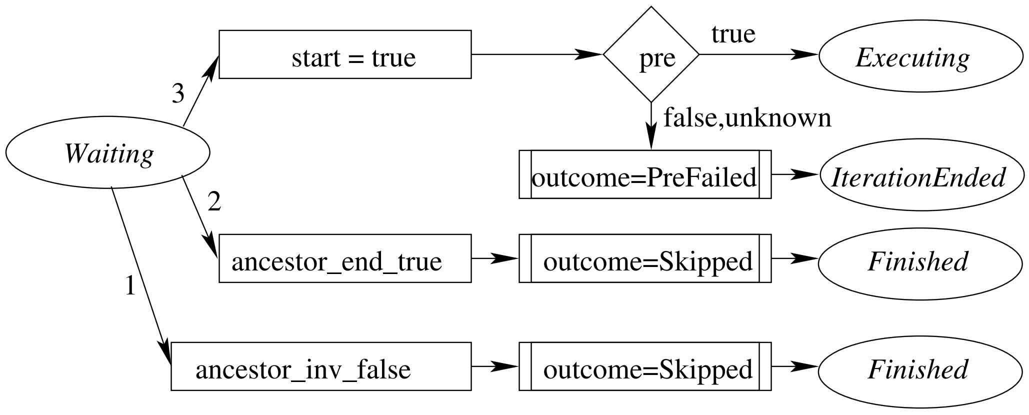

Figure 1: *Transitions from* Waiting

the ancestor end condition is given precedence: the status of the node is set to *Finished* and the outcome is set to *Skipped*.

Micro Steps

A micro step corresponds to transitions that modify only the internal, local data of a plan, i.e., the state of a node and the values of the declared variables. A micro step is defined as the synchronous parallel execution of atomic steps. Priorities are used for solving conflicting parallel assignments, e.g., simultaneous assignments to the same variables.

Macro Steps and Quiescence

A macro step is initiated by an external event. All the nodes waiting for that event are transitioned in parallel to their next states via micro steps. If these transitions trigger other condition changes, due to changes in the local data, these are executed until there are no more enabled transitions. The process of repeatedly applying micro steps until no more transitions are enabled is called *quiescence cycle*. After the execution of a macro step, a new external event is handled.

Micro Step Duration

PLEXIL does not make any assumption about the duration of execution of a micro step. An assumption that is commonly made for synchronous languages is that a step (in our case, a micro step) takes zero time. Or, alternatively, that the external world changes (and therefore the occurrence of external events happens) less frequently than the execution of steps. This assumption is not mandatory in PLEXIL. If a micro step takes more than zero time, this means that the execution of a macro step also takes more than zero time. Since the external world may keep changing in the meantime, it is possible that some micro steps within the macro step use obsolete data. A similar situation occurs if an event is processed much later than it was produced. In this case, it is possible that the current status of the external variables associated with the event might have changed. At the end of a macro step, the executive is updated with the latest information about the world and any obsolete data is discarded.

3 SEMANTIC FRAMEWORK

The very rich structure of the PLEXIL execution mechanism is specified using a *small-steps semantics* [16] via six term rewriting relations:

- The *evaluation* relation, denoted by $\rightsquigarrow$, corresponds to the instantaneous evaluation of PLEXIL expressions.
- The *atomic* relation, denoted by $\longrightarrow$, represents 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 $\overset{\star}{\longmapsto}$, describes how the external world is locally perceived by a plan.
- The *execution* relation, denoted by $\longmapsto$, describes how a plan reacts to a series of events.

These relations are defined in terms of mathematical structures that represent the internal state of the executive and the external state of the world.

3.1 Notation

Let $\to$ be in $\{\rightsquigarrow, \longrightarrow, \Longrightarrow, \Longrightarrow_{\downarrow}, \overset{\star}{\longmapsto}, \longmapsto\}$, we write $c \vdash a \to b$ to denote that, given a parameter c, the pair (a, b) belongs to $\to$. We say that $\to$ is *deterministic* if $c \vdash a \to b_1$ and $c \vdash a \to b_2$ implies that $b_1 = b_2$. Furthermore, we say that $\to$ is *terminating* if there are no infinite reductions of the form $c \vdash a_0 \to a_1, \ldots, c \vdash a_i \to a_{i+1}, \ldots$. Finally, given a parameter c, an element a is a $\to$*-normal form* if there is no b such that $c \vdash a \to b$.

3.2 Internal State

The internal state is represented by a set of processes π with two types of 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 an internal variable `x` with a value v. The set of processes π is a linear representation of the tree structure of a PLEXIL plan. In particular, process identifiers are fully qualified names where each process append its original node name to the qualified name of its parent.[2]

In our formalism, a node process is a record where each field represents an implicit variable of the node: `start` (start condition), `end` (end 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 `A.body` $=$ `x :=` e.

[2]In the examples, as the scope is clear from the context, we use simple identifiers instead of qualified names.

Except for `status` and `outcome`, all the other fields of a node state remain invariant during execution. Note, however, that conditions are expressions, which may depend on local and external variables. The expressions do not change, but their values do.

Henceforth, we write $\texttt{A with [a}_1\texttt{ = }v_1\texttt{,..., a}_n\texttt{ = }v_n\texttt{]}$ to denote the node process that is equal to `A` in all the fields except in $\texttt{a}_1, \ldots, \texttt{a}_n$, where it has the values $v_1, \ldots, v_n$, respectively.

The internal state of the plan in Example 2.2 is represented by a set

$$\begin{aligned}\pi \;=\; \{\; & \texttt{Node SafeDrive}, \texttt{Node OneMeter}, \\ & \texttt{Node TakePic}, \texttt{Node Counter}, \\ & \texttt{Node Counter}, \texttt{Var pictures:0}\;\}.\end{aligned}$$

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

For the rest of this paper, we assume that π is *well-formed*, i.e., variables are well-scoped and names are properly qualified.

3.3 External State

The set Σ denotes a snapshot of the world at a given moment in time. In consists of pairs $(\texttt{X} : v)$, where `X` is an external variable and v is its value. Variables in Σ are restricted to variables appearing in lookup operations and variables that serve as interface to the functional layer, e.g., variables that signal the abort of a command or that store a return value. We remark that time is not a special concept in PLEXIL. Indeed, if time is required by a plan, we just assume that there is variable `Time` in Σ that can be accessed via lookup operations as any other external variable.

In the case of Example 2.2, the environment Σ contains the external variable `Rover.WheelStuck` and interface variables related to the execution of commands `Rover.Drive` and `Rover.TakePicture`.

A plan does not have direct access to Σ, but to a local copy of it denoted by Γ. The variables in Σ and Γ are the same, but the values in Γ may be slightly outdated with respect to Σ. The sets Σ and Γ are called *environments* and we assume that they are *functional*, i.e., if (X, v) and (X, w) are both in the same environment, then $v = w$.

3.4 Expressions

The language includes a set of expressions formed by constant values, pre-defined functions, basic Boolean and arithmetic expressions, local variables drawn from π, and lookup on external variables drawn from Σ.

In order to accommodate different semantic variants of PLEXIL-like languages, these relations are arranged in three parametric, but independent, executions layers: (1) *kernel layer*, (2) *internal layer*, and (3) *external layer*.

3.5 Kernel Layer

The kernel layer defines the basic computation units in the language. More precisely, it provides concrete definitions of the evaluation and the atomic relations.

Evaluation Relation

The evaluation relation $(\Gamma, \pi) \vdash e \rightsquigarrow v$ represents the instantaneous evaluation of an expression e into a value v in environment Γ and local state π. This relation is defined by structural induction on e, e.g.,

$$\frac{\texttt{Var x:}v \ \in \ \pi}{(\Gamma, \pi) \ \vdash \ \texttt{x} \rightsquigarrow v} \ [\text{Var}]$$

$$\frac{(\texttt{X} : v) \ \in \ \Gamma}{(\Gamma, \pi) \ \vdash \ \texttt{LookupNow(X)} \rightsquigarrow v} \ [\text{LookupNow}]$$

$$\frac{\begin{array}{c}(\Gamma, \pi) \ \vdash \ e_1 \rightsquigarrow v_1 \\ (\Gamma, \pi) \ \vdash \ e_2 \rightsquigarrow v_2\end{array}}{(\Gamma, \pi) \ \vdash \ e_1 + e_2 \rightsquigarrow v_1 + v_2} \ [\text{Add}]$$

We assume that expressions are *well-typed* in the sense that conditions evaluate to Boolean values and expressions in assignments evaluate to the declared types of the assigned variables. A formal type system for PLEXIL is still under development.

As Γ is a functional environment and π is well-formed, the value v is unique. Therefore, we have the following proposition.

Proposition 1 *The relation $\rightsquigarrow$ is deterministic.*

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).

Atomic Relation

The atomic relation $(\Gamma, \pi) \vdash P \longrightarrow P'$, where $P \subset \pi$, corresponds to the state transition diagrams in [9]. As P is a set of processes, the atomic relation can be naturally expressed using the Chemical Reaction Model [2].

The Chemical Reaction Model is a formalism based on multi-set rewriting systems for modeling interactions between parallel processes. In this formalism, the set of processes is formed by atomic processes, in our case node processes and memory processes, and a binary associative-commutative operator (,) that maps a pair of processes P and Q into a parallel process P, Q. Interaction between processes are expressed by an associative-commutative rewrite system. For instance, the atomic rule that corresponds to the transition from *Waiting* to *Executing* in Figure 1 is written as follows. We assume that `Node A` $\in \pi$:

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

We also illustrate the rule that updates a memory cell of an assignment node whose status is *Executing*.

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

For completeness, we include all the atomic rules in Appendix A. Note that the order of the rules is relevant as it defines a precedence relation in case of conflict. By construction, the atomic relation is defined such that P is a single node process, e.g., W_{3a}, or a node process and a memory process, e.g., EA_{3a}. The resulting rewriting system is deterministic.

Proposition 2 *The relation* $\longrightarrow$ *is deterministic.*

3.6 Internal Layer

The internal layer concerns the local computations of the executive. In particular, it describes the synchronous and run-to-completion aspects of the PLEXIL execution mechanism. This layer is parameterized by a deterministic abstract atomic relation and provides concrete definitions of the micro relation and the quiescence relation. These relations have access to Γ, a local copy of the environment Σ, which remains invariant to all the internal computations.

Micro Relation

In PLEXIL, atomics are executed synchronously. For instance, the following plan effectively 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; }
  }
}
```

It may be possible that two nodes attempt to simultaneously write to the same variable. In this case, the conflict is solved by using a priority mechanism: every node has a priority and only the node with the higher priority has the right to write the memory cell.

In our semantic framework, we assume a strict partial order $\prec$ over nodes. The micro relation $\Gamma \vdash \pi \Longrightarrow \pi'$ is defined as the synchronous $\longrightarrow$-reduction of π for priority $\prec$:

$$\frac{\begin{array}{c}(\Gamma, \pi) \ \vdash \ P_1 \longrightarrow Q_1 \\ \dots \\ (\Gamma, \pi) \ \vdash \ P_n \longrightarrow Q_n\end{array}}{\Gamma \ \vdash \ \pi \Longrightarrow Q_1, \dots, Q_n},$$

where the only reducible expressions in π are $P_1, \dots, P_n$ and those that overlap with $P_1, \dots, P_n$ but have a lower $\prec$-priority.[3]

We have mechanically verified in the PVS theorem prover that for any deterministic atomic relation, the micro relation is deterministic.

Proposition 3 *The relation $\Longrightarrow$ is deterministic.*

Quiescence Relation

The run-to-completion policy in 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 [13] 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 $\Longrightarrow$. Note that since π' is a $\Longrightarrow$-normal form, π' cannot be reduced any further using the micro relation.

By simple induction on the length of the reduction $\pi \Longrightarrow^{*} \pi'$ and Proposition 3, we have verified in PVS that the quiescence relation is deterministic.

Proposition 4 *The relation $\Longrightarrow_{\downarrow}$ is deterministic.*

3.7 External Layer

The external layer concerns the event-driven aspect of the PLEXIL execution mechanism and the synchronization between the environments Σ and Γ. This layer is parameterized by an abstract deterministic quiescence relation and provides concrete definitions of the macro relation and the execution relation.

Events

An *event* is an external phenomenon that, by modifying Σ, enables one or more gate conditions in π. 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 framework, we do not model events directly but we model their effects in Σ and in gate conditions. Indeed, we assume a function $\Omega(\pi)$ that returns the set of start and

[3]Remark that $\longrightarrow$ is defined such that there is only one node process in each P_i. Therefore, we define the priority of P_i as the priority of that node.

end conditions, and negated invariant conditions of active nodes in π. We say that (Σ, π) is *enabled* if there is a guard e in $\Omega(\pi)$ that evaluates to *true*:

$$\mathit{enabled}(\Sigma, \pi) \;\;=\;\; \exists e \in \Omega(\pi) : \;(\Sigma, \pi) \;\vdash\; e \leadsto \mathit{true}\,.$$

For instance, during the execution of Example 2.2, $\Omega(\pi)$ may contain the conditions `OneMeter.status == FINISHED && pictures < 10` and `TakePic.status == FINISHED`. The pair (Σ, π) is enabled when either one of these conditions evaluates to *true*.

Macro Relation

The macro relation provides an updated Γ to the internal layer only if there is an event, i.e., if a gate condition in π is enabled.

$$\frac{\Gamma' = \begin{cases} \Sigma, & \text{if } \mathit{enabled}(\Sigma, \pi) \\ \Gamma, & \text{otherwise,} \end{cases} \qquad \Gamma' \;\vdash\; \pi \Longrightarrow_{\downarrow} \pi'}{\Sigma \;\vdash\; (\Gamma, \pi) \overset{\star}{\longmapsto} (\Gamma', \pi')}\,.$$

We have mechanically verified in PVS that for any deterministic quiescence relation, the macro relation is deterministic.

Proposition 5 *The relation* $\overset{\star}{\longmapsto}$ *is deterministic.*

Execution Relation

The execution relation defines the behavior of a plan for a series of sequential readings of the state of the world $\Sigma_0,\ldots,\Sigma_n$. Formally, the execution of π in $\Sigma_0,\ldots,\Sigma_n$ is inductively defined by the rule

$$\frac{\Sigma_i \vdash (\Gamma_i, \pi_i) \overset{\star}{\longmapsto} (\Gamma_{i+1}, \pi_{i+1})}{(\Sigma_i, \Gamma_i, \pi_i) \mapsto (\Sigma_{i+1}, \Gamma_{i+1}, \pi_{i+1})}\,,$$

where $\Gamma_0 = \Sigma_0$ and $\pi_0 = \pi$.

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 to the variables during the same macro step return the same values. In this case, Σ_i contains the value of the external variables at the time when they are first read during the i-th macro step.

In PLEXIL, the executive waits for an event to occur in order to perform a macro step. This synchronization is effectively achieved by the execution relation. A macro step is *always* performed independently of whether an event has occurred or not. However, due to the run-to-completion semantics of the quiescence cycle, if no event has occurred, the macro step is empty. Further macro steps are empty until, eventually, an event enables a non-empty macro step.

As we do not model events directly, we do not have to make explicit assumptions about their temporal nature. 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.

3.8 Example

Consider the following plan in an environment Σ that contains the external variable `Temp`.

```
Node Toy {
  int x = 0;
  int y = 0;
  List: {
    Node0 {
      Assignment: x:=10;
    }
    Node1 {
      Start: LookupOnChange(Temp) > 0;
      Assignment: x:=0;
    }
    Node2 {
     Start: x > 0,
     Assignment: y:=10;
    }
  }
}
```

Let $\pi_0 = \{\texttt{Var x}{:}0, \texttt{Var y}{:}0, \ldots\}$ and $\Sigma_0 = \{(\texttt{Temp} : 0), \ldots\}$. The following are possible executions of `Toy` for difference sequence of environments (in each case, the external state that enables a condition is underlined):

- Assuming

$$\begin{aligned}\Sigma_0 &= \Sigma_1 = \Sigma_2,\\ \Sigma_3 &= \{\texttt{Temp}{:}10, \ldots\},\\ \Sigma_4 &= \Sigma_3.\end{aligned}$$

  Then,

$$\begin{aligned}&(\Sigma_0, \Sigma_0, \pi_0) \mapsto\\ &(\Sigma_1, \Sigma_0, \{\texttt{Var x}{:}10, \texttt{Var y}{:}10, \ldots\}) \mapsto\\ &(\Sigma_2, \Sigma_0, \{\texttt{Var x}{:}10, \texttt{Var y}{:}10, \ldots\}) \mapsto\\ &(\underline{\Sigma_3}, \Sigma_0, \{\texttt{Var x}{:}10, \texttt{Var y}{:}10, \ldots\}) \mapsto\\ &(\Sigma_4, \Sigma_3, \{\texttt{Var x}{:}0, \texttt{Var y}{:}10, \ldots\}).\end{aligned}$$

- Assuming

$$\begin{aligned}\Sigma_1 &= \{\texttt{Temp}{:}10, \ldots\},\\ \Sigma_2 &= \Sigma_1.\end{aligned}$$

  Then

$$\begin{aligned}&(\Sigma_0, \Sigma_0, \pi_0) \mapsto\\ &(\underline{\Sigma_1}, \Sigma_0, \{\texttt{Var x}{:}10, \texttt{Var y}{:}10, \ldots\}) \mapsto\\ &(\Sigma_2, \Sigma_1, \{\texttt{Var x}{:}0, \texttt{Var y}{:}10, \ldots\}).\end{aligned}$$

4 PROPERTIES

This section presents the main theoretical properties of our framework for different semantic variants of PLEXIL-like languages. These properties include determinism, run-to-completion, and termination.

4.1 Determinism Under Full-Knowledge Hypothesis

Given a sequence $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$ and an initial state $(\Sigma_0, \Gamma_0, \pi_0)$, an *execution trace* is a sequence $(\Gamma_0, \pi_0), \ldots, (\Gamma_n, \pi_n)$ such that

$$\forall\ 0 \leq i < n : (\Sigma_i, \Gamma_i, \pi_i) \mapsto (\Sigma_{i+1}, \Gamma_{i+1}, \pi_{i+1}).$$

By natural induction on the length of execution traces and Proposition 5, we have verified in PVS the following property of the execution relation.

Theorem 1 (Determinism Under Full-Knowledge Hypothesis) *Given a sequence of environments $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$ and an initial state $(\Sigma_0, \Sigma_0, \pi_0)$, if the execution admits an execution trace, then it is unique.*

The property above states that 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*. Determinism under full-knowledge hypothesis guarantees that two identical plans, deployed on two different platforms, have the same outputs if their readings of the external world are the same. Notice that this does not mean that the readings of the external world have to be synchronized on time. For determinism under full-knowledge hypothesis to hold, it is sufficient that the readings of the external world after each macro step coincide.

4.2 Operational Determinism

Determinism under full-knowledge hypothesis is an interesting theoretical property. However, in practice, the full-knowledge hypothesis is seldom achieved in space applications due to uncertainties in the world. From an operational point of view, it is important to constraint potential non-deterministic behaviors to interactions with the external environment.

We say that an execution trace is *sterile* in $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$ if no event occurred during its execution, i.e.,

$$\forall\ 0 \leq i < n : \neg\texttt{enabled}(\Sigma_i, \pi_i).$$

The following property has been mechanically verified in PVS.

Theorem 2 (Operational Determinism) *For all sequence of environments $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$ and initial state $(\Sigma_0, \Sigma_0, \pi_0)$, if the execution admits a sterile execution trace, then it is unique.*

The operational determinism states that in absence of external events, the language is fully deterministic.

It is possible to instantiate our framework such that the resulting language is deterministic under full-knowledge hypothesis, but is not operationally deterministic. For 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:[4]

```
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 action nodes in `Sequence`, including the full iteration in `Loop`, are performed during the quiescence cycle in the first macro step. The plan finishes in an state where the precondition of node `C` holds.

Assume that the framework is instantiated in the following way:

1. The quiescence relation is defined as the micro step, i.e., $\Longrightarrow_{\downarrow} \equiv \Longrightarrow$.

2. The macro step relation is defined such that Γ is always updated with Σ at the beginning of the macro step, i.e.,

$$\frac{\begin{array}{c}\Gamma' = \Sigma \\ \Gamma' \ \vdash \ \pi \Longrightarrow_{\downarrow} \pi'\end{array}}{\Sigma \ \vdash \ (\Gamma, \pi) \overset{\star}{\longmapsto} (\Gamma', \pi')} \, .$$

We call this semantics *Updated Step-by-Step*. The language defined by this semantics does not have the run-to-completion and the event-driven aspects of PLEXIL. On the other hand, we can verify that it does satisfy determinism under full-knowledge hypothesis. However,

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

the updated step-by-step semantics does not satisfy operational determinism as the following executions show.[5]

- Assuming $\Sigma_i = \{\texttt{Temp}{:}0, \ldots\}$, for all i:

$$\begin{array}{l}(\Sigma_0, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}0, \ldots\}) \mapsto \\ (\Sigma_1, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}1, \ldots\}) \mapsto \\ \ldots \\ (\Sigma_9, \Sigma_8, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}9, \ldots\}) \mapsto \\ (\Sigma_{10}, \Sigma_9, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}10, \ldots\}).\end{array}$$

- Assuming $\Sigma_0 = \{\texttt{Temp}{:}0, \ldots\}$ and $\Sigma_i = \{\texttt{Temp}{:}5, \ldots\}$, for all $i > 0$:

$$\begin{array}{l}(\Sigma_0, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}0, \ldots\}) \mapsto \\ (\Sigma_1, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}1, \ldots\}) \mapsto \\ \ldots \\ (\Sigma_9, \Sigma_8, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}9, \ldots\}) \mapsto \\ (\Sigma_{10}, \Sigma_9, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}5, \texttt{Var x}{:}10, \ldots\}).\end{array}$$

Since none of the gate conditions in `Sequence` are enabled by an event, all execution traces of this plan are sterile. One may expect, as the operational determinism property states, that the output of this plan is independent of the external environment. However, in the first case, the execution ends in a state where the precondition of node `C` holds, while, in the second case, it does not.

4.3 Run-to-Completion Semantics

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 should remain invariant under the macro relation. In PVS, we have verified that our semantic framework satisfies the following theorem.

Theorem 3 (Run-to-Completion) *For all* Σ, Σ', Γ, Γ', π, *and* π', *if* $\Sigma \vdash (\Gamma, \pi) \overset{\star}{\longmapsto} (\Gamma', \pi')$ *and* $\neg\textit{\textbf{enabled}}(\Sigma', \pi')$, *then*

$$\Sigma' \vdash (\Gamma', \pi') \overset{\star}{\longmapsto} (\Gamma', \pi').$$

This property is a consequence of the definition of the quiescence relation as the normalized reduction of $\Longrightarrow$. A natural question that arises from this definition is if a quiescence cycle, or for that matter, a macro step, always terminates, i.e., if for all Σ, Γ, and π, there is a always pair (Γ', π') such that

$$\Sigma \ \vdash \ (\Gamma, \pi) \overset{\star}{\longmapsto} (\Gamma', \pi').$$

Unfortunately, this is not always the case as illustrated by the following simple example.

[5]This example is just an illustration. In the step-by-step semantics, it takes more than one step to perform an action.

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

Because the repeat condition is always true, the first-macro step of this plan does not terminate. Therefore, this plan does not admit any execution trace.

4.4 Alternative No Run-to-Completion Semantics

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*: Quiescence relation is defined such that a repeating node goes from execution status *Waiting* to *Waiting* only once (or a pre-specified number of times) during a quiescence cycle.

We can verify that these two instantiations yield PLEXIL-like languages that satisfy both operational determinism and termination of the quiescence relation. The following execution compares the PLEXIL semantics to the step-by-step semantics for the example in Section 4.2. Assume $\Sigma_0 = \{\texttt{Temp}{:}0, \ldots\}$ and $\Sigma_i = \{\texttt{Temp}{:}5, \ldots\}$, for all $i > 0$.

- PLEXIL Semantics.

$$\begin{array}{l}(\Sigma_0, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}0, \ldots\}) \mapsto \\ (\Sigma_1, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}10, \ldots\}).\end{array}$$

- Step-by-Step Semantics.

$$\begin{array}{l}(\Sigma_0, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}0, \ldots\}) \mapsto \\ (\Sigma_1, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}1, \ldots\}) \mapsto \\ \ldots \\ (\Sigma_9, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}9, \ldots\}) \mapsto \\ (\Sigma_{10}, \Sigma_0, \{\texttt{Var tempA}{:}0, \texttt{Var tempB}{:}0, \texttt{Var x}{:}10, \ldots\}).\end{array}$$

In contrast to the updated step-by-step semantics in Section 4.2, the PLEXIL semantics, step-by-step semantics, and broken-quiescence semantics do not update Γ unless there is an event in Σ. For this reason, these latter semantics satisfy operational determinism, while the former semantics does not. 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. The idea is to ignore any change in the environment before a macro step is completed and updating the environment only when some task is complete.

Semantics	**DFH**	**OD**	**TQ**	**RC**
PLEXIL	$\surd$	$\surd$		$\surd$
Updated Step-by-Step	$\surd$		$\surd$	
Step-by-Step	$\surd$	$\surd$	$\surd$	
Broken-Quiescence	$\surd$	$\surd$	$\surd$	

Figure 2: *PLEXIL-like semantic variants and their properties*

It can be easily checked that the updated step-by-step semantics, the step-by-step semantics, and the broken-quiescence semantics do not satisfy the run-to-completion property (Theorem 3). Figure 2 summarizes the semantic variants presented in this section and their properties: **D**eterminism under **F**ull-knowledge **H**ypothesis, **O**perational **D**eterminism, **T**ermination of **Q**uiescence, and **R**un to **C**ompletion.

We observe that the semantics discussed here differ in the way they are sensitive (or insensitive) to stuttering in the environment. Intuitively, a stuttering step does not change the state of the relevant external world, and therefore it should not have an observable effect on execution. As explained in [1], stuttering is a useful notion when reasoning about systems in a modular (or compositional) way. We plan to investigate this further.

5 RELATED WORK

The work presented here touches on design and implementation of plan execution languages and of synchronous programming languages. We discuss some of them below.

Plan execution languages and executive systems vary greatly in sophistication and in the degree of autonomy (i.e., the capability to operate without human intervention) that they provide. See [17] for a survey of command execution systems for NASA spacecraft and robots. 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 control strategies, which take into account unexpected changes in the environment, 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 to the high uncertainty in the environment.

A pre-requisite for the predictability and verifiability of an execution language is the availability of a formal execution semantics. However, few existing plan execution languages have well-defined semantics or are powerful enough to express complex control plans [9]. PLEXIL attempts to unify and extend many execution control capabilities of other execution languages and executive systems, while providing a clear semantics. Therefore, it is a good subject for our formal study here.

As mentioned before, PLEXIL is related to synchronous languages such as Esterel [5], Lustre [7] or Signal [10]. These languages assume that time advances in lockstep with one or more clocks. A synchronous program *reacts* to inputs such as clock ticks or more general *events*, synchronously, i.e., in *zero time*. These atomic reactions are macro steps that are formed of sequences of smaller steps (micro steps). See [3] for a survey of synchronous languages.

PLEXIL is most closely related to Esterel. Unlike Lustre and Signal, which are data-flow declarative languages, Esterel is an imperative language that focuses on describing control. An Esterel program consists of a collection of nested, concurrent threads whose execution is synchronized to a single, global clock. Threads communicate through signals, while in PLEXIL communication is through shared variables. Compile-time analysis techniques are used to rule out bad Esterel programs that have potentially infinite (or deadlocked) macro steps. Interestingly, PLEXIL and the variants studied here, do not require such an analysis to guarantee determinism and, for some variants, termination of the quiescence relation.

Related languages also include the many different variants of Statecharts [12]. The Statecharts formalism provides a hierarchical representation of state information; concurrency is modeled by simultaneously active states and by parallel transition execution. Communication is done via a broadcast mechanism. Most Statecharts variants are based on a discrete time domain (natural numbers) and transitions are executed in zero time; there is no guarantee of determinism. See [18] for a survey of Statecharts variants. In UML [8], for example, Statecharts use a run-to-completion semantics that is similar to PLEXIL's quiescence: all the locally enabled transitions of a state chart caused by the processing of an event will be executed before the next event will be taken for execution; priorities can be used to (partially) solve non-determinism.

It has been observed [3] that there is a need to combine synchrony and asynchrony [14] to take into account the intrinsic asynchrony of the interfaces of a synchronous system. Note that a similar problem is encountered in PLEXIL due to to the asynchronous interaction of plan execution with the external environment. This is an area of active research that has led to new programming paradigms, such as "Globally Asynchronous, Locally Synchronous Systems" [11] and to extensions of existing synchronous languages. For example, [4] presents a language that extends Esterel with asynchronous rendez-vous communication.

The main difference between these languages and PLEXIL is that PLEXIL is tailored for command execution in uncertain environments: it provides a set of specialized conditions that trigger and monitor the execution of nodes and it enables invocation and abortion of external activities that are specific to plan execution. When a node fails execution, it transitions to specialized failure modes which may trigger the execution of other nodes (to perform recovery actions for example). Moreover, PLEXIL provides a unique combination of synchronous execution, shared variables communication, and no zero-time transition delay that does guarantee determinism.

6 CONCLUSION

The definition of a language that satisfies both operational determinism and termination of the quiescence relation has been the object of an intensive debate during the design of the PLEXIL language. The standard semantics of PLEXIL does not guarantee termination of quiescence, but we have seen that two alternative semantics do guarantee both properties. The use of our semantic framework as a designing tool has helped to understand these issues better.

The entire PVS development for the semantic framework presented here consists of 14 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 14 theories include 107 lemmas, 1094 lines of specification, and 2864 lines of proofs. All the lemmas have been mechanically checked and there are no axioms in the theories. We are currently working on proofs of other properties of planning languages such as liveness and compositionality in the same framework. The summary of the PVS output is included in Appendix B.

PLEXIL is evolving and being able to formally check properties of variants of the language was, from the beginning, a goal of our development. This justifies the modular approach with several layers 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, thus allowing us to experiment with different language design decisions without re-doing the proofs for all the layers. Perhaps, such an approach could be recommended to other languages that have many semantic variations (such as Statecharts).

Finally, 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 Verna, for providing invaluable insights on the intended semantics of PLEXIL.

REFERENCES

[1] M. Abadi and L. Lamport. Composing Specifications. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, volume 430, pages 1–41, Berlin, Germany, 1989. Springer-Verlag.

[2] Jean-Pierre Banâtre and Daniel Le Métayer. Gamma and the chemical reaction model. In *Proceedings of the Coordination '95 Workshop*, Londres, 1995. IC Press.

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE, Special Issue on Embedded Systems, Volume 91*, 2003.

[4] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages*, pages 85–98, 1993.

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

[6] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, Franois Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). In *World Wide Web Consortium, Recommendation REC-xml11-20060816*, August 2006.

[7] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Symposium on Principles of Programming Languages (POPL)*, 1987.

[8] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Sci. Comput. Program.*, 55(1-3):81–115, 2005.

[9] Tara Estlin, Ari Jónsson, Corina Păsăreanu, Reid Simmons, Kam Tso, and Vandi Verna. Plan Execution Interchange Language (PLEXIL). NASA Technical Memorandum.

[10] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. In *Proceedings of the IEEE, Volume 79(9)*, pages 1321–1336, 1991.

[11] Nicolas Halbwachs and Siwar Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, October 2002. LNCS 2491, Springer Verlag.

[12] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[14] Robin Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

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

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

[17] Vandi Verna, Ari Jónsson, Reid Simmons, Tara Estlin, and Rich Levinson. 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)*, 2005.

[18] Michael von der Beeck. A comparison of Statecharts variants. In *Proceedings of the 3rd International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1994.

A ATOMIC-STEP RELATION

The predicates `ancestor_inv_false` and `ancestor_end_true` are inductively defined as follows.

$$\begin{array}{l} \texttt{ancestor\_inv\_false}(\Gamma,\pi,\texttt{A}) \quad = \\ \quad (\Gamma,\pi) \vdash \texttt{A.parent.inv} \rightsquigarrow \textit{false} \text{ or} \\ \quad \texttt{ancestor\_inv\_false}(\Gamma,\pi,\texttt{A.parent}). \end{array}$$

$$\begin{array}{l} \texttt{ancestor\_end\_trrue}(\Gamma,\pi,\texttt{A}) \quad = \\ \quad (\Gamma,\pi) \vdash \texttt{A.parent.end} \rightsquigarrow \textit{true} \text{ or} \\ \quad \texttt{ancestor\_end\_true}(\Gamma,\pi,\texttt{A.parent}). \end{array}$$

Inactive Nodes

$$\frac{\begin{array}{l}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A) \\ \texttt{A.status} = \textit{Inactive}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow \texttt{Node A with } [\ \texttt{status = } \textit{Finished}, \texttt{outcome = } \textit{Skipped}\ ]}\ [\mathrm{I}_1]$$

$$\frac{\begin{array}{l}\texttt{ancestor\_end\_true}(\Gamma,\pi,A) \\ \texttt{A.status} = \textit{Inactive}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow \texttt{Node A with } [\ \texttt{status = } \textit{Finished}, \texttt{outcome = } \textit{Skipped}\ ]}\ [\mathrm{I}_2]$$

$$\frac{\begin{array}{l}\texttt{A.parent.status} = \textit{Executing} \\ \texttt{A.status} = \textit{Inactive}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow \texttt{Node A with } [\ \texttt{status = } \textit{Waiting}\ ]}\ [\mathrm{I}_3]$$

Waiting Nodes

$$\frac{\begin{array}{l}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A) \\ \texttt{A.status} = \textit{Waiting}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow \texttt{Node A with } [\ \texttt{status = } \textit{Finished}, \texttt{outcome = } \textit{Skipped}\ ]}\ [\mathrm{W}_1]$$

$$\frac{\begin{array}{l}\texttt{ancestor\_end\_true}(\Gamma,\pi,A) \\ \texttt{A.status} = \textit{Waiting}\end{array}}{(\Gamma,\pi) \vdash \texttt{Node A} \longrightarrow \texttt{Node A with } [\ \texttt{status = } \textit{Finished}, \texttt{outcome = } \textit{Skipped}\ ]}\ [\mathrm{W}_2]$$

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

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.start} \rightsquigarrow \textit{true}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.pre} \not\rightsquigarrow \textit{true}\\ \texttt{A.status} = \textit{Waiting}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{IterationEnded}, \texttt{outcome = } \textit{PreFailed}\ ]\end{array}}\ [\mathrm{W}_{3b}]$$

Executing List Nodes

$$\frac{\begin{array}{l}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A)\\ \texttt{A.status} = \textit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{Failing}, \texttt{outcome = } \textit{ParentFailed}\ ]\end{array}}\ [\mathrm{EL}_{1}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.inv} \rightsquigarrow \textit{false}\\ \texttt{A.status} = \textit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{Failing}, \texttt{outcome = } \textit{InvFailed}\ ]\end{array}}\ [\mathrm{EL}_{2}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \textit{true}\\ \texttt{A.status} = \textit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\ \vdash \texttt{Node A} \longrightarrow\ \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{Finishing}\ ]\end{array}}\ [\mathrm{EL}_{3a}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \textit{true}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \not\rightsquigarrow \textit{true}\\ \texttt{A.status} = \textit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{Failing}, \texttt{outcome = } \textit{PostFailed}\ ]\end{array}}\ [\mathrm{EL}_{3b}]$$

Executing Command Nodes

$$\frac{\begin{array}{l}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A)\\ (\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \textit{true}\\ \texttt{A.status} = \textit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{Finished}, \texttt{outcome = } \textit{ParentFailed}\ ]\end{array}}\ [\mathrm{EC}_{1a}]$$

$$\frac{\begin{array}{l}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A)\\ (\Gamma,\pi)\ \vdash\ \texttt{A.end} \not\rightsquigarrow \textit{true}\\ \texttt{A.status} = \textit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{Failing}, \texttt{outcome = } \textit{ParentFailed}\ ]\end{array}}\ [\mathrm{EC}_{1b}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.inv} \rightsquigarrow \textit{false}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \textit{true}\\ \texttt{A.status} = \textit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad [\ \texttt{status = } \textit{IterationEnded}, \texttt{outcome = } \textit{InvFailed}\ ]\end{array}}\ [\mathrm{EC}_{2a}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.inv} \rightsquigarrow \mathit{false}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.end} \not\rightsquigarrow \mathit{true}\\ \texttt{A.status} = \mathit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad \texttt{[status = } \mathit{Failing}, \texttt{outcome = } \mathit{InvFailed}\ \texttt{]}\end{array}}\ [\mathrm{EC}_{2b}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \mathit{true}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \rightsquigarrow \mathit{true}\\ \texttt{A.status} = \mathit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad \texttt{[status = } \mathit{IterationEnded}, \texttt{outcome = } \mathit{Success}\ \texttt{]}\end{array}}\ [\mathrm{EC}_{3a}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \mathit{true}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \not\rightsquigarrow \mathit{true}\\ \texttt{A.status} = \mathit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad \texttt{[status = } \mathit{IterationEnded}, \texttt{outcome = } \mathit{PostFailed}\ \texttt{]}\end{array}}\ [\mathrm{EC}_{3b}]$$

Executing Assignment Nodes

$$\frac{\begin{array}{l}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A)\\ \texttt{A.status} = \mathit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad \texttt{[status = } \mathit{Finished}, \texttt{outcome = } \mathit{ParentFailed}\ \texttt{]}\end{array}}\ [\mathrm{EA}_{1}]$$

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.inv} \rightsquigarrow \mathit{false}\\ \texttt{A.status} = \mathit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad \texttt{[status = } \mathit{IterationEnded}, \texttt{outcome = } \mathit{InvFailed}\ \texttt{]}\end{array}}\ [\mathrm{EA}_{2}]$$

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

$$\frac{\begin{array}{l}(\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \mathit{true}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \not\rightsquigarrow \mathit{true}\\ \texttt{A.status} = \mathit{Executing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad \texttt{[status = } \mathit{IterationEnded}, \texttt{outcome = } \mathit{PostFailed}\ \texttt{]}\end{array}}\ [\mathrm{EA}_{3b}]$$

Executing Function Call Nodes

$$\frac{(\Gamma,\pi)\ \vdash\ \texttt{A.ancestor\_inv} \rightsquigarrow \mathit{false}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash & \texttt{Node A: } \{\mathit{Executing},\ldots\} \longrightarrow \\ & \texttt{Node A: } \{\mathit{Finished}, \mathit{ParentFailed},\ldots\}\end{array}}\ [\mathrm{EF}_1]$$

$$\frac{(\Gamma,\pi)\ \vdash\ \texttt{A.inv} \rightsquigarrow \mathit{false}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash & \texttt{Node A: } \{\mathit{Executing},\ldots\} \longrightarrow \\ & \texttt{Node A: } \{\mathit{IterationEnded}, \mathit{InvFailed},\ldots\}\end{array}}\ [\mathrm{EF}_2]$$

$$\frac{\begin{array}{c}(\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \mathit{true} \\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \rightsquigarrow \mathit{true}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash & \texttt{Node A: } \{\mathit{Executing},\ldots\} \longrightarrow \\ & \texttt{Node A: } \{\mathit{IterationEnded}, \mathit{Success},\ldots\}\end{array}}\ [\mathrm{EF}_{3a}]$$

$$\frac{\begin{array}{c}(\Gamma,\pi)\ \vdash\ \texttt{A.end} \rightsquigarrow \mathit{true} \\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \not\rightsquigarrow \mathit{true}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash & \texttt{Node A: } \{\mathit{Executing},\ldots\} \longrightarrow \\ & \texttt{Node A: } \{\mathit{IterationEnded}, \mathit{PostFailed},\ldots\}\end{array}}\ [\mathrm{EF}_{3b}]$$

Failing List Nodes

$$\frac{\begin{array}{c}\forall\ \texttt{A}_i \in \texttt{A.children} : \texttt{A}_i\texttt{.status} \in \{\mathit{Waiting}, \mathit{Finished}\} \\ \texttt{A.outcome} = \mathit{ParentFailed} \\ \texttt{A.status} = \mathit{Failing}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash \texttt{Node A} \longrightarrow & \texttt{Node A with} \\ & \quad [\ \ \texttt{status = } \mathit{Finished}\ \ ]\end{array}}\ [\mathrm{FL}_{1a}]$$

$$\frac{\begin{array}{c}\forall\ \texttt{A}_i \in \texttt{A.children} : \texttt{A}_i\texttt{.status} \in \{\mathit{Waiting}, \mathit{Finished}\} \\ \texttt{A.outcome} \neq \mathit{ParentFailed} \\ \texttt{A.status} = \mathit{Failing}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash \texttt{Node A} \longrightarrow & \texttt{Node A with} \\ & \quad [\ \ \texttt{status = } \mathit{IterationEnded}\ \ ]\end{array}}\ [\mathrm{FL}_{1b}]$$

Failing Command Nodes

$$\frac{\begin{array}{c}\texttt{abort\_complete\_true}(\Gamma,\pi,A) \\ \texttt{A.outcome} = \mathit{ParentFailed} \\ \texttt{A.status} = \mathit{Failing}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash \texttt{Node A} \longrightarrow & \texttt{Node A with} \\ & \quad [\ \ \texttt{status = } \mathit{Finished}\ \ ]\end{array}}\ [\mathrm{FC}_{1a}]$$

$$\frac{\begin{array}{c}\texttt{abort\_complete\_true}(\Gamma,\pi,A) \\ \texttt{A.outcome} \neq \mathit{ParentFailed} \\ \texttt{A.status} = \mathit{Failing}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash \texttt{Node A} \longrightarrow & \texttt{Node A with} \\ & \quad [\ \ \texttt{status = } \mathit{IterationEnded}\ \ ]\end{array}}\ [\mathrm{FC}_{1b}]$$

Finishing **Nodes**

$$\frac{\begin{array}{c}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A)\\ \texttt{A.status} = \mathit{Finishing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad\qquad [\ \texttt{status = } \mathit{Failing}, \texttt{outcome = } \mathit{ParentFailed}\ ]\end{array}}\ [\mathrm{F}_1]$$

$$\frac{\begin{array}{c}(\Gamma,\pi)\ \vdash\ \texttt{A.inv} \rightsquigarrow \mathit{false}\\ \texttt{A.status} = \mathit{Finishing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad\qquad [\ \texttt{status = } \mathit{Failing}, \texttt{outcome = } \mathit{InvFailed}\ ]\end{array}}\ [\mathrm{F}_2]$$

$$\frac{\begin{array}{c}\forall\ \texttt{A}_i \in \texttt{A.children} : \texttt{A}_i\texttt{.status} \in \{\mathit{Waiting}, \mathit{Finished}\}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \rightsquigarrow \mathit{true}\\ \texttt{A.status} = \mathit{Finishing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad\qquad [\ \texttt{status = } \mathit{IterationEnded}, \texttt{outcome = } \mathit{Success}\ ]\end{array}}\ [\mathrm{F}_{3a}]$$

$$\frac{\begin{array}{c}\forall\ \texttt{A}_i \in \texttt{A.children} : \texttt{A}_i\texttt{.status} \in \{\mathit{Waiting}, \mathit{Finished}\}\\ (\Gamma,\pi)\ \vdash\ \texttt{A.post} \not\rightsquigarrow \mathit{true}\\ \texttt{A.status} = \mathit{Finishing}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad\qquad [\ \texttt{status = } \mathit{IterationEnded}, \texttt{outcome = } \mathit{PostFailed}\ ]\end{array}}\ [\mathrm{F}_{3b}]$$

IterarationEnded **Nodes**

$$\frac{\begin{array}{c}\texttt{ancestor\_inv\_false}(\Gamma,\pi,A)\\ \texttt{A.status} = \mathit{IterationEnded}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad\qquad [\ \texttt{status = } \mathit{Finished}, \texttt{outcome = } \mathit{ParentFailed}\ ]\end{array}}\ [\mathrm{IE}_1]$$

$$\frac{\begin{array}{c}\texttt{ancestor\_end\_true}(\Gamma,\pi,A)\\ \texttt{A.status} = \mathit{IterationEnded}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash \texttt{Node A} \longrightarrow & \texttt{Node A with [}\\ & \texttt{status = } \mathit{Finished}\ \ \texttt{]}\end{array}}\ [\mathrm{IE}_2]$$

$$\frac{\begin{array}{c}(\Gamma,\pi)\ \vdash\ \texttt{A.repeat} \rightsquigarrow \mathit{true}\\ \texttt{A.status} = \mathit{IterationEnded}\end{array}}{\begin{array}{l}(\Gamma,\pi)\vdash\ \texttt{Node A} \longrightarrow \texttt{Node A with}\\ \qquad\qquad [\ \texttt{status = } \mathit{Waiting}, \texttt{outcome = } \mathit{Success}\ ]\end{array}}\ [\mathrm{IE}_3]$$

$$\frac{\begin{array}{c}(\Gamma,\pi)\ \vdash\ \texttt{A.repeat} \rightsquigarrow \mathit{false}\\ \texttt{A.status} = \mathit{IterationEnded}\end{array}}{\begin{array}{ll}(\Gamma,\pi)\ \vdash \texttt{Node A} \longrightarrow & \texttt{Node A with [}\\ & \texttt{status = } \mathit{Finished}\ \ \texttt{]}\end{array}}\ [\mathrm{IE}_4]$$

Finished **Nodes**

$$\frac{\begin{array}{c}\texttt{A.parent.status} = \mathit{Waiting} \\ \texttt{A.status} = \mathit{Finished}\end{array}}{\begin{array}{l}(\Gamma, \pi) \vdash \ \texttt{Node A} \longrightarrow \texttt{Node A with} \\ \qquad\qquad\texttt{[status = } \mathit{Inactive}, \texttt{outcome = } \mathit{Success} \texttt{]}\end{array}} \; [\mathrm{F}]$$

B PVS SUMMARY

```
Proof summary for theory reductions
   normalized_det.......................proved - complete   [shostak](0.15 s)
   normalized_nf........................proved - complete   [shostak](0.03 s)
   normalized_symm......................proved - complete   [shostak](0.03 s)
   Theory totals: 3 formulas, 3 attempted, 3 succeeded (0.21 s)

Proof summary for theory plexil_sets
   state2list_TCC1......................proved - complete   [shostak](0.14 s)
   Theory totals: 1 formulas, 1 attempted, 1 succeeded (0.14 s)

Proof summary for theory external
   macro_det............................proved - complete   [shostak](0.28 s)
   macro_weak_det.......................proved - complete   [shostak](0.06 s)
   execution_step_TCC1..................proved - complete   [shostak](0.37 s)
   execution_det........................proved - complete   [shostak](0.29 s)
   trace?_TCC1..........................proved - complete   [shostak](0.25 s)
   trace?_TCC2..........................proved - complete   [shostak](0.32 s)
   sterile?_TCC1........................proved - complete   [shostak](0.18 s)
   operational_determinism_TCC1.........proved - complete   [shostak](0.03 s)
   operational_determinism_TCC2.........proved - complete   [shostak](0.03 s)
   operational_determinism_TCC3.........proved - complete   [shostak](0.04 s)
   operational_determinism_TCC4.........proved - complete   [shostak](0.04 s)
   operational_determinism..............proved - complete   [shostak](1.20 s)
   Theory totals: 12 formulas, 12 attempted, 12 succeeded (3.09 s)

Proof summary for theory extended_reds
   extended_det.........................proved - complete   [shostak](0.07 s)
   Theory totals: 1 formulas, 1 attempted, 1 succeeded (0.07 s)

Proof summary for theory synchronous_reds
   synchronous_det......................proved - complete   [shostak](0.09 s)
   Theory totals: 1 formulas, 1 attempted, 1 succeeded (0.09 s)

Proof summary for theory internal
   max_priority_TCC1....................proved - complete   [shostak](0.59 s)
   max_redices_TCC1.....................proved - complete   [shostak](0.02 s)
   max_redices_TCC2.....................proved - complete   [shostak](0.02 s)
   micro_TCC1...........................proved - complete   [shostak](0.57 s)
   micro_det............................proved - complete   [shostak](0.05 s)
   quiescence_det.......................proved - complete   [shostak](0.03 s)
   Theory totals: 6 formulas, 6 attempted, 6 succeeded (1.28 s)

Proof summary for theory list_iterator
   iterate_TCC1.........................proved - complete   [shostak](0.03 s)
   Theory totals: 1 formulas, 1 attempted, 1 succeeded (0.03 s)

Proof summary for theory list_extra
   member_nth...........................proved - complete   [shostak](0.63 s)
   every_forall.........................proved - complete   [shostak](0.12 s)
   some_exists..........................proved - complete   [shostak](0.11 s)
   Theory totals: 3 formulas, 3 attempted, 3 succeeded (0.86 s)

Proof summary for theory plexil_defs
   TrueValue_TCC1.......................proved - complete   [shostak](0.01 s)
   FalseValue_TCC1......................proved - complete   [shostak](0.01 s)
   fun_context..........................proved - complete   [shostak](0.10 s)
   priority_TCC1........................proved - complete   [shostak](0.05 s)
   fun_procs?_TCC1......................proved - complete   [shostak](0.13 s)
   fun_procs_cons.......................proved - complete   [shostak](0.68 s)
   wf_procs?_TCC1.......................proved - complete   [shostak](0.09 s)
   wf_procs?_TCC2.......................proved - complete   [shostak](0.16 s)
   member_process.......................proved - complete   [shostak](0.05 s)
   get_node_TCC1........................proved - complete   [shostak](0.07 s)
   get_node_member......................proved - complete   [shostak](0.46 s)
   get_parent_TCC1......................proved - complete   [shostak](0.04 s)
   nth_node.............................proved - complete   [shostak](0.11 s)
   parent_TCC1..........................proved - complete   [shostak](0.10 s)
   parent...............................proved - complete   [shostak](0.56 s)
   set_body_TCC1........................proved - complete   [shostak](0.04 s)
   Theory totals: 16 formulas, 16 attempted, 16 succeeded (2.66 s)

Proof summary for theory compiler
   divide_TCC1..........................proved - complete   [shostak](0.02 s)
   equal_TCC1...........................proved - complete   [shostak](0.01 s)
   plan_order...........................proved - complete   [shostak](0.04 s)
   compile_plan_TCC1....................proved - complete   [shostak](0.04 s)
   Theory totals: 4 formulas, 4 attempted, 4 succeeded (0.15 s)

Proof summary for theory plexil_eval
   get_global_TCC1......................proved - complete   [shostak](0.01 s)
   get_global_TCC2......................proved - complete   [shostak](0.05 s)
   get_local_TCC1.......................proved - complete   [shostak](0.01 s)
   get_local_TCC2.......................proved - complete   [shostak](0.05 s)
   eval_TCC1............................proved - complete   [shostak](0.05 s)
   eval_TCC2............................proved - complete   [shostak](0.05 s)
   eval_TCC3............................proved - complete   [shostak](0.06 s)
   eval_TCC4............................proved - complete   [shostak](0.08 s)
   eval_TCC5............................proved - complete   [shostak](0.25 s)
   eval_TCC6............................proved - complete   [shostak](0.06 s)
   eval_TCC7............................proved - complete   [shostak](0.08 s)
   eval_TCC8............................proved - complete   [shostak](0.29 s)
   eval_TCC9............................proved - complete   [shostak](0.05 s)
   eval_TCC10...........................proved - complete   [shostak](0.08 s)
   eval_TCC11...........................proved - complete   [shostak](0.25 s)
   eval_TCC12...........................proved - complete   [shostak](0.06 s)
   eval_TCC13...........................proved - complete   [shostak](0.05 s)
   eval_TCC14...........................proved - complete   [shostak](0.08 s)
   eval_TCC15...........................proved - complete   [shostak](0.26 s)
   eval_TCC16...........................proved - complete   [shostak](0.05 s)
   eval_TCC17...........................proved - complete   [shostak](0.07 s)
   eval_TCC18...........................proved - complete   [shostak](0.27 s)
   eval_TCC19...........................proved - complete   [shostak](0.05 s)
   eval_TCC20...........................proved - complete   [shostak](0.07 s)
   eval_TCC21...........................proved - complete   [shostak](0.26 s)
   eval_TCC22...........................proved - complete   [shostak](0.06 s)
   eval_TCC23...........................proved - complete   [shostak](0.08 s)
   eval_TCC24...........................proved - complete   [shostak](0.25 s)
   eval_TCC25...........................proved - complete   [shostak](0.05 s)
   eval_TCC26...........................proved - complete   [shostak](0.07 s)
   eval_TCC27...........................proved - complete   [shostak](0.24 s)
   eval_TCC28...........................proved - complete   [shostak](0.06 s)
   eval_TCC29...........................proved - complete   [shostak](0.08 s)
   eval_TCC30...........................proved - complete   [shostak](0.25 s)
   eval_TCC31...........................proved - complete   [shostak](0.08 s)
   eval_TCC32...........................proved - complete   [shostak](0.06 s)
   eval_TCC33...........................proved - complete   [shostak](0.25 s)
   eval_TCC34...........................proved - complete   [shostak](0.05 s)
   eval_TCC35...........................proved - complete   [shostak](0.15 s)
   eval_TCC36...........................proved - complete   [shostak](0.07 s)
   Theory totals: 40 formulas, 40 attempted, 40 succeeded (4.44 s)

Proof summary for theory kernel
   ancestor_invariant_false_TCC1........proved - complete   [shostak](0.09 s)
   ancestor_invariant_false_TCC2........proved - complete   [shostak](0.03 s)
   ancestor_end_true_TCC1...............proved - complete   [shostak](0.08 s)
   all_children_waiting_or_finished_TCC1.proved - complete   [shostak](0.07 s)
   all_children_waiting_or_finished_TCC2.proved - complete   [shostak](0.33 s)
   inactive_TCC1........................proved - complete   [shostak](0.11 s)
   inactive_TCC2........................proved - complete   [shostak](0.05 s)
   inactive_TCC3........................proved - complete   [shostak](0.12 s)
   waiting_TCC1.........................proved - complete   [shostak](0.04 s)
   executing_node_list_TCC1.............proved - complete   [shostak](2.02 s)
   executing_node_command_TCC1..........proved - complete   [shostak](2.00 s)
   executing_node_assignment_TCC1.......proved - complete   [shostak](2.02 s)
```

```
    executing_node_funcall_TCC1...........proved - complete   [shostak](1.99 s)
    failing_node_list_TCC1................proved - complete   [shostak](0.07 s)
    failing_node_command_TCC1.............proved - complete   [shostak](0.07 s)
    finishing_TCC1........................proved - complete   [shostak](6.52 s)
    finishing_TCC2........................proved - complete   [shostak](0.50 s)
    iteration_ended_TCC1..................proved - complete   [shostak](6.57 s)
    finished_TCC1.........................proved - complete   [shostak](0.15 s)
    Theory totals: 19 formulas, 19 attempted, 19 succeeded (22.83 s)

Grand Totals: 107 proofs, 107 attempted, 107 succeeded (35.85 s)
```