

Plan Execution Interchange Language (PLEXIL) Version 1.0

**Vijay Baskaran, Michael Dalal, Tara Estlin, Chuck Fry, Robert Harris,
Michael Iatauro, Ari Jónsson, Corina Pasareanu, Reid Simmons,
Vandi Verma**

**NASA Ames Research Center, Jet Propulsion Laboratory, Carnegie Mellon
University**

Technical Content Dated: 10/19/2007

Introduction

Plan execution is a cornerstone of spacecraft operations, irrespective of whether the plans to be executed are generated on board the spacecraft or on the ground. Plan execution frameworks vary greatly, due to both different capabilities of the execution systems, and relations to associated decision-making frameworks. The latter dependency has made the reuse of execution and planning frameworks more difficult, and has all but precluded information sharing between different execution and decision-making systems.

As a step in the direction of addressing some of these issues, a general plan execution language, called the Plan Execution Interchange Language (PLEXIL), has been developed. PLEXIL is capable of expressing concepts used by many high-level automated planners and hence provides an interface to multiple planners. PLEXIL includes a domain description that specifies command types, expansions, constraints, etc., as well as feedback to the higher-level decision-making capabilities.

PLEXIL extends many execution control capabilities of other systems. The key characteristics of PLEXIL are that it is compact, semantically clear, and deterministic given the same sequence of measurements. At the same time, the language is quite expressive and can represent simple branches, floating branches, loops, time- and event-driven activities, concurrent activities, sequences, and temporal constraints. The core syntax of the language is simple and uniform, making plan interpretation simple and efficient, while enabling the application of validation and testing techniques.

Most command sequencing languages and execution engines used on spacecraft are highly simplified to ensure that spacecraft safety can be guaranteed under all execution paths. This has ruled out systems that allow complex logic. Although the PLEXIL

language is structurally simple, it is capable of expressing complex control constructs that include conditionals, branches, floating contingencies, loops, event-driven control, time-based control, etc. The language has well-defined semantics, including guarantees of unambiguous responses for any given plan and situation.

Complex systems must be operated in different ways at different times or in different contexts. For example, spacecraft operations vary according to mission phase. Furthermore, operations may fail and equipment may break or behave off-nominally. Control plans must specify what to do in all anticipated situations including how to respond to failures and problems.

There are many ways to represent control strategies for execution. The current approach for many spacecraft that repeatedly employ the same command sequence is table-driven or parameter-driven software. Here, the software is pre-defined, but certain aspects of how it operates are defined by parameters or tables. For example, a temperature controller may have a simple program whose parameters are the desired temperature and permitted deviations. Programming languages provide the opposite extreme; a program can be encoded to implement and execute a control strategy. However, this approach brings into play the full complexity of programming languages, which makes verification and validation very difficult. In addition, this approach is too brittle for spacecraft operations. Many programming languages require compilation and software installation to be done before a program can be executed on a given system.

In between those two extremes are languages that express control strategies or plans. Some such languages are akin to common programming languages, thus providing the full power of programming, but also providing some additional structure to express plans, steps, actions, etc. These languages make plan representation easier than full programming languages, but are still difficult to verify. Other languages are specialized to only express plans that can be executed by appropriate execution software. The advantage of specialized execution languages is that their expressiveness can be circumscribed and their semantics clearly defined. These properties are crucial for validation, checking, and testing of plans. However, few existing languages have well-defined semantics or are powerful enough to express complex control plans.

PLEXIL is an expressive language that can represent complex plans with loops, conditions, contingencies and other necessary control structures, while also having clear formal semantics that make the expected outcome of execution unambiguous and enable plan validation. Accompanying this language is an execution engine, the Universal Executive, designed to implement the PLEXIL language and provide interfaces to controlled systems.

Plan Execution Interchange Language (PLEXIL) is a small, well-defined language that provides a great deal of power in terms of describing complex plans, while being easy to formally validate and verify. This feature has significant implications in terms of representing and reasoning about control plans. For a long time, the specification of spacecraft operation plans like command sequences has been limited to very simple

control structures. This has largely been due to the difficulty of specifying control structures like conditions and contingencies in such plans, without making the expected semantics unclear and the validation problem unsolvable. Existing languages with expressive representations do so at the cost of formal guarantees about execution.

Another novel feature is the approach used to access information from the underlying functional layer, which avoids the requirement that there be a telemetry database, and supports different ways of accessing state information. The lookup feature and the ability to perform internal storage and calculations make it possible to replicate most any behavior of database-driven systems without requiring a specific data base. Not requiring a database simplifies the implementation of an executive for PLEXIL and can, in many cases, make an executive significantly faster. PLEXIL plans can be written in a number of ways: with syntactic enhancements, in core PLEXIL using the PLEXIL context free grammar, or in XML using the PLEXIL XML schema definition. In addition, a graphical editor for PLEXIL plans is also available.

PLEXIL Syntax

The fundamental building block of the PLEXIL language is a node. A node has two primary functional components, a set of conditions that drive the execution of the node, and the "content" of the node, which specifies what gets executed. Nodes are arranged in a hierarchical fashion, providing controllability at different levels of abstractions. Executing a high level node means enabling the evaluation of the next lower level set of nodes. In terms of content, there are two primary kinds of nodes:

1. List nodes are the internal nodes in the hierarchy, having child nodes (or children) of which they are the parent node (or parent). When a parent node commences execution, this enables the evaluation of its child nodes.
2. Action nodes are the leaf nodes of the hierarchy. They perform actions, including commanding systems, performing calculations, providing updates, and submitting information to other components.

The conditions for each node drive the execution of the node. There are nominal control conditions that specify when the node should start executing, when it should finish executing, and when it should be repeated:

1. Start conditions specify when the node should start execution.
2. End conditions specify when the node should finish its execution.
3. Repeat conditions specify when the node should be repeated, i.e., made eligible for a repeat execution.

Then there are failure conditions that identify when node execution is not successful:

1. Preconditions are checked after start conditions become true and verify that it is safe to execute the node. If the check fails, the node cannot be safely executed and will be aborted and marked as having failed.

2. Postconditions are checked after the node has completed execution and verify that the node did what it was supposed to. If the check fails, the node did not execute correctly and will be marked as having failed.

3. Invariant conditions are checked during node execution and monitor any conditions that are needed for the safe continued execution of the node. If the check fails at any point during execution, the node execution cannot be safely continued, the node will be aborted and marked as having failed.

The simple combination of these conditions and hierarchical nodes provides a great deal of power with very simple linguistic structure. Local variable declarations (of type Boolean, numeric, string, and time) and operations on these variables are also supported. Conditions, contingencies, and loops can be implemented, which provides virtually all desired control capabilities for plans, procedures and other types of operations control specifications. At the same time, the very simple language makes it possible to define formal semantics and develop methods for validation of instances.

The core language has almost no redundancy and hence it is small; but this can make it arduous to specify plans using only those simple constructs. To alleviate this and to make the language more usable PLEXIL provides syntactic short cuts. Example of short cuts include if-then-else statements, for loops, while loops, macros, etc. Any such short cut maps directly into instances of the core language, so no semantics need to be added and core validation techniques can be applied. The PLEXIL language has a great deal of flexibility in terms of interfacing with the underlying controlled system. There are two primary elements to this interface; one is the commanding interface and the other is the sensing interface.

Simple Example:

```
DriveUntilStuck:
{
  RepeatCondition: LookupOnChange("Rover:wheelStuck") ==
false;
  NodeList:
    DriveOneMeter: {
      Command: Rover:Drive(1);
    }
  }
}
```

1) Node Types

There are eight types of nodes in PLEXIL:

- *List nodes* may have one or more child nodes. They are containers that provide scope for child nodes, but do not perform any explicit action.
- *Command nodes* issue commands that drive the system being operated on.
- *Assignment nodes* perform a local computation, whose value is assigned to a variable.
- *Function Call nodes* access external functions that perform computations, but do not (directly) alter the state of the system.
- *Update nodes* provide information to the planning and decision-support interface.
- *Plan Request nodes* request a new execution plan.
- *Library Node Call nodes* will inline PLEXIL nodes stored in a library
- *Empty nodes* contain only node attributes (and they perform no action).

The type of a given node is identified by a specific element in the node's definition.

Each of the different kinds of nodes are described as follows.

A **NodeList** element identifies a List node and encloses a list of nodes.

```
NodeList:
  <node1>
  ...
  <nodeN>
```

A **Command** element has the form:

```
Command: [<variable> =] <command_name> [(<argument_list>)];
```

where `command_name` is an identifier, and `argument_list` is an optional comma-separated list of zero or more arguments, which may be variables, declared or internal states, or constants. Although numeric operations and lookups are not allowed in the list of arguments, this is not a real limitation since assignments may be used to assign values to variables that may be used as parameters to commands.

The assignment of the command's return value to a variable is optional, and if specified, `<variable>` must be a variable previously declared in the node or passed through its interface from an ancestor.

Commands are the interface to the functional layer, or library calls (e.g. functions to perform complex computation) specified in the domain description. The command name is specified in the domain description.

An **Assignment** element has the form:

```
Assignment: <variable> = <expression>;
```

where <expression> can be a declared variable (including interface variable), an internal state variable (e.g. NodeId.state.Timepoint), a LookupNow, a constant, or a numeric operation, and <variable> can only be a declared variable, including interface variable.

A **Function Call** element has the form:

```
FunctionCall: [<variable> =] <function_name>
(<argument_list>);
```

An **Update** element has the form :

```
Update: (<name> (<value> | <variable>))*;
```

A **Plan Request** element has the form :

```
Request: <NodeId> (<name> (<value> | <variable>))*;
```

A **Library Node Call** element has the form:

```
LibraryCall: <NodeId> [<RenameNodeId>] [<alias_list>];
```

where <NodeId> is the id of the node from the library; the optional <RenameNodeId> specifies how to re-name the node when it is in-lined in the current plan. The <alias_list> is an optional list of alias-es, which are pairs of the form (<node_parameter> (<value> | <variable>)). An alias allows one to rename/assign a node parameter (i.e. a variable present in the interface of the library node) with an actual value or a variable declared in the context in which the library node call is performed. It is an error to assign a value to an "inout" node parameter (see discussion about interfaces below).

2) Node Attributes

Each node has the following elements, called node attributes:

- NodeID: A unique symbolic name
- StartCondition: Boolean expression
- EndCondition: Boolean expression
- PreCondition: Boolean expression
- PostCondition: Boolean expression
- InvariantCondition: Boolean expression
- RepeatCondition: Boolean expression
- Priority: Integer. Lower Number is higher priority.
- DeclaredVariables: List of local variable declarations
- Interface: List of variables "passed" to node

The execution of a node is driven by the node conditions, which are Boolean expressions. Conditions capture internal and external information as well as temporal relationships. A

NodeID is a unique identifier for a node. NodeIDs are locally scoped. Hence node A and node B may both have children called C. The child of node A is referenced as A.C, the child of node B as B.C, etc. Two siblings in the node tree are not allowed to have the same name.

3) External and Internal Information

External States

To control execution, PLEXIL node elements may acquire information from world events and states. In PLEXIL we refer to world events and states as world state. The specific names used to look up world state are defined in a domain description.

Access to events and states is via one-time lookups, notifications of change in value, or lookups at a given frequency. The most common update is for temporal information, such as a temporal value, e.g., a comparison like:

```
AbsoluteTimeWithin("2005-10-09 14H06M12S UTC" , PLUS_INFINITY)
```

In the example above an event will be triggered when the absolute time is in the interval ["2005-10-09 14H06M12S UTC", PLUS_INFINITY] PLEXIL expects that all lookups, commands, and general function calls have no side effects. In other words, they only affect the state of execution through the value they return.

Internal Variables

In addition to the external states, a PLEXIL plan has access to the internal state of a node. There are a number of internal variables, such as the start and end times of each state of a node, the execution state of a node, etc. These are typically used to either track the state of PLEXIL execution, or to store information from external states. These variables are global and are referenced using an instance of a NodeId (and node state in some cases):

- <NodeId>.state
- <NodeId>.<node state>.Timepoint
- <NodeId>.outcome
- <NodeId>.failureType

Timepoints are integers that are bound to actual values at run time. The start and end Timepoint of each node state are stored.

A node can only reference itself, and its immediate parent, children, and siblings.

Declared Local Variables

Other variables are defined as local variables in nodes. Variable declarations are similar to corresponding declarations in programming languages. A counter, for example, could be defined in a node that leads to a looping structure:

```
Integer i=0
```

Variables passed as parameters to lookups, commands, and functions are passed by value. Hence the value of these variables is not changed. The types supported by PLEXIL are:

- Integer
- Real
- Boolean
- String
- Blob
- Time

Interfaces

Finally, declared variables can appear in the interface of nodes (these interface variables are also called node parameters). Interface variables that are read-only are specified with the keyword `in` and interface variables that are read-write are specified with the keyword `inout`. `inout` interface variables are passed by reference from a parent node to a child node.

A child of a node only has access to the variables declared in the parent that are passed via the interface.

By default, there are no variables in the interface of a node. Each declared variable from the parent that a node requires must be explicitly imported. However, syntactic sugar includes syntax for importing all variables from a parent.

It is an error to declare locally in a node a variable that has the same name as a variable that appears in a node interface.

Note that interfaces can declare the type of the parameters (optional): this is necessary for library nodes.

Example 1: the following example shows a PLEXIL plan: variable "i" is declared twice, but this is allowed since by default the interface of node "DoWork" is empty.

```
Node ErrorPlan
{ Integer i = 0;
  Integer j = 1;
  NodeList:
```

```

    DoWork: {
      Integer i = 0;
      Command: Rover:doWork(i, j);
    }
  }
}

```

Example 2: the following example shows an incorrect PLEXIL plan, since "i" is part of the interface of node "DoWork".

```

Node CorrectPlan
{ Integer i = 0;
  Integer j = 1;
  NodeList:
    DoWork: {
      InOut Integer i;
      Integer i = 0;
      Command: Rover:doWork(i, j);
    }
}
}

```

4) Sensing the external world

The syntax used for sensing world states in PLEXIL is called a Lookup. It is a read-only interface that “looks up” values of world states, or measurements. Each external sensor value or world state that might be accessed via a lookup is identified by a domain-specific measurement name, e.g., “Temperature”. Lookups can appear in Assignments or Conditions. PLEXIL provides three types of Lookups:

LookupOnChange

LookupOnChange(“Temperature”, 2) is an event-based lookup that returns an initial value immediately and then repeatedly returns the state value whenever it changes. A “minimal change” parameter may be specified to restrict the value to be returned only when it changes by more than the specified tolerance, which is 2 (°C) in the example above.

LookupWithFrequency

LookupWithFrequency(“Temperature”, 10) is a repeated lookup. The initial value is returned immediately. The second parameter specifies a frequency for checking subsequent state value. In this case it specifies that the value of state “Temperature” should be checked 10 times per second.

LookupNow

LookupNow(“Temperature”) returns the current value of state “Temperature” immediately.

In PLEXIL, the domain of declared variables and values returned by lookups and commands are extended with two additional values – Unknown and Exception. Unknown implies that the value is un-initialized and Exception implies that an exception occurred while performing the lookup or command.

5) Command interface

The commanding interface is relatively straightforward but it supports a range of different behaviors for the command execution. The simplest approach is a command that gets executed without any further feedback, except through sensed effects. The next level is a command that provides a return value; this can either be a confirmation of execution being initiated or the result of the outcome. The latter means supporting blocking commands. Finally, some commands provide an initial return (or not) and then send a result after the command execution process is completed. Each of these is supported by PLEXIL, but what is particularly notable about PLEXIL in this context is the ability to have arbitrary logical combination of sensor inputs provide indications of the outcome.

6) Conditions

Conditions drive the execution of each node. Each condition is evaluated with a Boolean expression. Boolean expressions are arbitrary logical formulas, without quantification, where each predicate is either a temporal relation or a data relation. Relations are based on Boolean expressions or standard comparisons, such as equality, inequality, “less than or equal,” etc. Relations can refer to either internal variables, external state and event information, or time. We allow getting current time through a lookup. Temporal relations `CurrentTimeWithin` and `AbsoluteTimeWithin` are also provided as syntactic sugar.

Based on the way conditions are checked, we have two types of conditions: gate conditions (monitored continuously) and check conditions (checked once):

Gate conditions

- `StartCondition` (alias `Start`)
- `EndCondition` (alias `End`)
- `InvariantCondition` (alias `Invariant`)

Check conditions

- `PreCondition` (aliases `Pre`, `Precondition`)
- `PostCondition` (aliases `Post`, `Postcondition`)
- `RepeatCondition` (alias `Repeat`)

The gate conditions are checked repeatedly until they evaluate to true, while check conditions are instantaneous - the result determines what is done at that time. A gate condition is checked whenever any of the variables in the Boolean expression representing the gate condition alter. In addition, the conditions are also classified in an alternate way. `Start`, `End` and `Repeat` conditions drive the execution of a node. `Pre`, `Post`,

and Invariant conditions monitor the execution of a node. Hence, these conditions are also called failure conditions. If any of these conditions fail to evaluate to true, the node execution is aborted with a failure indication.

Conditions may encode Data or Temporal constraints or Boolean combinations of them. Data conditions are constraints on internal or external variables, which are read via lookups (details in section 3.2). Temporal conditions specify absolute time constraints or time constraints relative to Timepoints in nodes.

Some example of boolean expressions that may appear in conditions:

```
currentTimeWithin{node1.FINISHED.START, +[20S,30S]}
LookupOnChange{"Rover:batteryCharge"} > 120.0
node3.state == FINISHED && node3.outcome == SUCCESS
```

Here, `node1.FINISHED.START` represents the Timepoint at which `node1` entered state `FINISHED`. The `RelativeTimeWithin` condition above may be understood by representing current time explicitly as `T`. Then `currentTimeWithin{node1.FINISHED.start, [20S,30S]}` is equivalent to:

```
T ∈ [node1.state.FINISHED.START+20S,
node1.FINISHED.START+30S]
```

Lookups that appear in gate conditions must be of type event based or frequency based, and lookups that appear in check conditions must be request based.

PLEXIL Semantics

Clear semantics make it easier to do verification and validation and to implement a lightweight executive that conforms to the semantics of execution. The execution of PLEXIL plans is predictable. In other words, given the same sequence of measurements, the execution is deterministic.

The execution in PLEXIL is entirely 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.

1) Atomic Steps: Node Execution

The execution semantics of an individual PLEXIL node is specified in terms of node states and transitions (atomic steps) between node states that are triggered by condition changes.

Node States

Each node must be in one and only one of the following states at any given time:

- Inactive
- Waiting
- Executing
- Finishing
- Iteration_Ended
- Failing
- Finished

Node Transitions

The set of condition changes that cause node state transitions are as follows:

- SkipCondition T : The skip condition changes from unknown or false to true.
- StartCondition T : The start condition changes from unknown or false to true.
- InvariantCondition F/U : Invariant condition changes from true to false or unknown.
- EndCondition T : End condition changes to true
- Ancestor_inv_condition F/U : The invariant condition of any ancestor changes to false or unknown.
- Ancestor_end_condition T : The end condition of any ancestor changes to true
- All_children_waiting_or_finished T : This is true when all child nodes are in either in node state waiting or finished and no other states.
- Command_abort_complete T : When the abort for a command action is completed.
- Function_abort_complete T : The abort of a function call is completed.
- Parent_waiting T : The (single) parent of the node transitions to node state waiting.
- Parent_executing T : The (single) parent of the node transitions to node state executing.
- RepeatCondition T/F : the repeat condition changes from unknown to either true or false.

PLEXIL uses three valued logic and conditions can evaluate to True, False, or Unknown. This allows interfaces to real-time systems where environment/sensor values may not be available at all times.

Conditions changes are set to unknown until the node transitions to a state where that condition change is monitored (can result in a state transition).

Nominal Execution

At the beginning of plan execution, all the nodes in a plan are initialized to state Inactive. An Inactive node does not affect the external system at all. When the parent of a node enters state Executing (for the root of a plan, which has no parent, this condition is immediately true), the node is activated, by transitioning it to state Waiting (details in Figure 13). A node remains in state Waiting until its start condition is met (details in Figure 16). The default start condition is True which implies that the node may execute immediately upon activation. In state Executing a node or its children perform the main body of execution actions. Upon completion of the action (e.g. command or assignment finished) leaf nodes transition to state Iteration_Ended from which they can either transition back to state Waiting (for looping nodes) or to state Finished, which signals the end of execution for the node, including all loops for repeating nodes (details in Figure 14, Figure 19, and Figure 18). The execution of list nodes proceeds similarly (as shown in Figure 15), except that there is an extra state, Finishing, which implies that the execution purpose of the node is complete and the node is only waiting for running child nodes to finish (Figure 21). If an ancestor node loops, the node may once again come into existence, but not otherwise.

Execution in the Presence of Failures When a failure occurs (i.e. one of the pre-, post- or maintenance conditions fail) a list node enters state Failing and causes the sub-tree to abort in a deterministic manner (details in Figure 17). Only in the case of a failure does a parent node abort a child node. State Iteration_Ended is only for looping nodes. It signals the end of a single iteration of execution (Figure 22).

Node Termination

There are three main causes for node termination: completion of execution, external events, and faults. The default completion of a node depends on the type of node. Assignment and Command nodes end when the assignment is complete or a function call returns. A List node (hierarchical node) ends when all its child nodes have finished. External events or cascading effects of external events may satisfy the explicit end condition of a node. When the end condition of a list node is satisfied and some of its child nodes are still executing the node cleanly terminates its child nodes. PLEXIL semantics for clean termination allow the running of “clean-up” nodes to ensure safe termination of processes. The semantics of clean termination of a list node with running children are :

- Only currently executing and just-activated child nodes continue to run
- Pending child nodes whose start conditions are not satisfied do not run
- Parent node waits for active child nodes to finish executing

Faults can also drive node termination. A node fails if its invariant conditions are violated or Pre- or Post-conditions are not satisfied. When a node fails it aborts its child nodes without clean-up. When a node fails no more events are processed by the sub-tree rooted at that node. All clean-up actions are handled by sibling nodes. The Outcome is a node

attribute that provides additional information about the result of node execution. During execution, the outcome of a node records the current execution status. A node may have any one of the following outcomes:

- Success
- Failure
- Skipped
- Unknown

The node outcome is initialized to Unknown. The outcome is set to Skipped if the node did not run, and to Success if the current iteration completed successfully. The outcome is set to Failure if a failure happened. Outcomes provide information only for the current iteration; they are reset for repeating nodes.

When a node fails, the failureType is assigned one of the following values:

- Precondition failed
- Postcondition failed
- Invariant condition failed
- Parent failed
- Infinite loop

Note that all conditions are checked once upon transition to a state in which they apply. Only the condition changes listed above cause node state transitions, e.g. a start condition changing to true causes node state transitions, but the start condition changing to false does not cause any node state transitions. Once a condition is enabled it stays enabled until it is explicitly reset. The conditions are only reset for repeating nodes.

The complete set of node state transitions that govern the semantics of the execution of a single node are provided in the section on node state transition diagrams. In certain states, e.g. state Waiting, all node types have the same semantics. In other cases, such as state Executing, the semantics depend on the node type (list, command, assignment...). For efficiency we represent ancestor end conditions. These are easily computed from the immediate parent and child nodes. In principle, a node only needs to know about its single immediate parent and all child nodes. PLEXIL plans are modular. PLEXIL allows safe execution of modular sub-plans (plans from a library) in the nominal case, including allowing the sub-plans authority over when execution of the sub-plan is complete. The only restriction is that child nodes may not loop if the parent is complete, but they may run to completion. One of the features of PLEXIL is that it is very expressive. It can represent command sequences where there is a pre-defined ordering, and can also represent a large number of “parallel” or “concurrent” nodes that may need to be executed at any instance, or in any combination. Additional nodes may also be added without interrupting execution. PLEXIL does not insert any latency in execution by polling nodes in some pre-defined order.

Default Values for Node Conditions

The default value for the Start Condition, Pre, Post and Invariant Condition of a node is true. The default value for Repeat Conditions is false. The default End Conditions are different depending on the type of node: The default End Condition of a List node is “All children finished”. The default End Condition of a Command node is “Command returned value” and the default End Condition of an Assignment and Function Call node is “Assignment/Function call complete”. The default end condition for an Update/Request node is "Update/Request invocation successful". The actual "end condition" of a Command, Function Call, Update, and Request node is a conjunction of the explicitly specified expression for the End Condition and the default condition. This is not the case for a List node. When an explicit End Condition is specified for a List node, it replaces the default. It does not make sense to specify an end condition for an Assignment node.

2) Micro Steps

Micro-steps correspond to transitions that modify only the local data in the executive, i.e. node states and outcomes and the values of the local variables. A micro-step is defined as the synchronous parallel execution of a transition, as defined by the state transition diagrams. No two assignment nodes can execute in parallel if they write to the same variable. Priorities are used for solving conflicting parallel assignments (e.g. assignment to the same variables). Precedence order on condition changes is used to resolve conflicts when multiple condition changes occur simultaneously.

3) 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.

4) 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.

5) Semantics of Lookups

There are a number of parameters for lookups. The subsets of parameters parsed for a condition or assignment are different. For example, if a frequency is specified with a lookup in an assignment it is ignored since an assignment is considered to be atomic. An internal PLEXIL event is generated when the value returned by a lookup (either event based or frequency based) has changed (i.e, the previous value is different from the current value by more than the Tolerance, if specified). Note that a change in value is reported based only on the information from lookups. The true state of the world may change at a higher frequency.

Lookups simply read state values. The domain declaration contains a list of state names that PLEXIL expects to look up “safely.” In other words, looking up these states should not change any states. Note that an implementation of the executive may choose to cache state values without violating this requirement.

It is good form to ensure that anything specified as a lookup in the domain description is known to have a low latency return value, else it may delay the execution of a node in which it is used (for example, if it were used in a Precondition).

Only `LookupWithFrequency` and `LookupOnChange` may be used in gate conditions (such as start conditions, invariant conditions, and end conditions). If the lookup has a frequency, the variables in these conditions are checked at the specified frequency. If a `LookupOnChange` is specified, the variable is checked asynchronously.

```
StartCondition:{ LookupOnChange{"Rover battery level"} > 10.0
&& LookupWithFrequency{"E-box temperature", 1} > 20.0
&& {powerTrackingNode.state == EXECUTING}
&& AbsoluteTimeWithin{"2005-03-21 16H20M00S UTC", PLUS-
INFINITY}}
```

In the example above, an asynchronous event is triggered whenever the rover battery level changes. The value of the E-box temperature is checked at a frequency of 1 Hz. If this value has changed since the last time it was checked, an internal event is triggered. The state of the `powerTrackingNode` is maintained internally and it triggers an event when it changes. Time too is maintained internally and events based on time are triggered internally by the executive.

The start condition specified in the example above is checked whenever an event corresponding to the a change in the rover battery level or the E-box temperature is

triggered or if the state of `powerTrackingNode` changes or if the current time enters the window [“2005-03-21 16H20M00S UTC”, PLUS-INFINITY].

Only `LookupNow` may be used in a check condition (such as Precondition, and Postcondition) or an Assignment.

Example 1

There are three types of Lookups:

- 1) Lookup with Frequency

```
StartCondition: LookupWithFrequency{x, 0.1;}
```

- 2) Lookup triggered by change

```
StartCondition: LookupOnChange{x, 0.5}
```

- 3) Lookup now. These lookups appear in check conditions (such as the Precondition below) or in Assignments. This lookup is evaluated only once at the time the condition is evaluated.

```
StartCondition: LookupOnChange{y} == 4  
PreCondition: LookupNow{x} != 1
```

In this example, the first lookup (frequency based) returns the value of `x` at time $T=10$, $T=20$ and $T=30$. The value of `x` changes at $T=30$. In the figure above a solid square represents that the value of the variable represented within the square was returned, and a dashed square represents that in addition to the value being returned an condition change was also triggered. In the third case, the value of `y` is returned in an event-based manner from the functional layer when it changes. The lookup for the value of `x` is part of a Precondition and is checked only once at the instance when the Start Condition is satisfied. The Start Condition in this example is satisfied at $T=35$ when `y=4`, hence the lookup for `x` is performed at $T=35$ and the value of `x` is returned. In this example the Precondition would evaluate to false.

Example 2

```
StartCondition:{ LookupWithFrequency{x, 0.1} == 1 ^  
LookupNow{y} == 3}
```

Example 2 is shown in the figure below. At $T=10$ a `LookupNow` is performed to initialize the value of `x` from UNKNOWN to 0. The Start Condition is checked, but the value of `y` at this point is still UNKNOWN, so the condition does not yet hold. When the value of `y`

changes to 2 another, the StartCondition is checked once again. In this case, the StartCondition evaluates to false since the value of x is 0. The StartCondition is checked again at T=20, but since the value of x is still 0 the condition still evaluates to false. The same happens when the value of y changes to 3. At time T=30, a lookup of the value of x returns 1 and the Start Condition is checked again. This time it evaluates to true.

6) Semantics of Commands

Calls to commands do NOT block execution. And command nodes do not finish until the command call completes (so they can have duration).

7) Three Valued Logic

There is an additional values for data and Boolean expressions, which is called "UNKNOWN". The domain of declared variables and values returned by lookups and commands (which may be assigned to declared variables) is extended to include the value UNKNOWN. UNKNOWN means uninitialized or that the lookup or command failed. The default value of un-initialized declared variables is also UNKNOWN. UNKNOWN has non-standard interpretation in a Boolean expression. Below is an illustration of a non-standard truth value we also call UNKNOWN.

Assume we have an invariant condition:

```
LookupWithFrequency{"Temperature", 1} > 0 &&  
LookupWithFrequency{"Battery", 0.01} > 20
```

At initialization the values are:

```
Temperature = UNKNOWN  
Battery = UNKNOWN
```

Whenever a variable that appears in a Boolean expression evaluates to UNKNOWN or FAIL the expression evaluates to UNKNOWN. We extend the definition of logical operations AND (), OR (), and NOT(!) as follows:

```
TRUE ^ UNKNOWN = UNKNOWN  
FALSE ^ UNKNOWN = FALSE  
TRUE | UNKNOWN = TRUE  
FALSE | UNKNOWN = UNKNOWN  
UNKNOWN ^ UNKNOWN = UNKNOWN  
UNKNOWN | UNKNOWN = UNKNOWN  
!UNKNOWN = UNKNOWN
```

When a condition evaluates to MAYBE or FAIL the node execution proceeds as shown in the node state transition diagrams.

In addition, UNKNOWN+3 would evaluate to UNKNOWN and Node Timepoints that have not occurred are also UNKNOWN. The PLEXIL "isKnown" operator may be used to test if the value of the variable is known.

PLEXIL Examples

Example 1 : Drive to Target in PLEXIL Core

Our first PLEXIL plan is one for controlling a rover. In this contrived example, a rover is commanded to drive until either a target is in view, or time has reached 10. If the target comes into view, an image is taken using the Pancam. If the time limit is reached, an image is taken using the Navcam. All the while temperature is monitored, and the heater is turned on whenever temperature drops below 0, and turned off when it exceeds 10. (Note that units of temperature and time are ignored here).

This plan has a top level or root node (DriveToTarget), that represents a starting point. It is a list node whose children define the actions described above. The primitive nodes in this plan are either Command or Assignment nodes.

```
DriveToTarget: {
  Boolean drive_done = false, timeout = false;

  NodeList:
  Drive: {
    Command: rover_drive(10);
  }

  StopForTimeout: {
    StartCondition: LookupOnChange(time,10);
    NodeList:
    Stop: {
      Command: rover_stop();
    }
    SetTimeoutFlag: {
      Assignment: timeout = true;
    }
  }

  StopForTarget: {
    StartCondition: LookupWithFrequency("target_in_view",
10);
    NodeList:
    Stop: {
      Command: rover_stop();
    }
    SetDriveFlag: {
      Assignment: drive_done = true;
    }
  }
}
```

```

    }
}

TakeNavcam: {
  StartCondition: timeout == true;
  Command: take_navcam();
}

TakePancam: {
  StartCondition: drive_done;
  Command: take_pancam();
}

Heater: {
  StartCondition: LookupOnChange("temperature") < 0;
  EndCondition: LookupOnChange("temperature") >= 10;
  RepeatCondition: true;
  Command: turn_on_heater();
}
}

```

Example 2 : Drive to Target in PLEXIL Syntactic Sugar

Our second example is a compact version of the plan in example 1, coded in PLEXIL's proposed syntactic sugar. They are equivalent. For more details look at the section on PLEXIL syntactic sugar

```

DriveToTarget: {
  Boolean drive_done = false, timeout = false;

  NodeList: {
    Command: rover_drive(10);

    When AbsoluteTimeWithin(10, POSITIVE_INFINITY) {
      Command: rover_stop();
      Assignment: timeout = true;
    }

    When LookupWithFrequency("target_in_view", 10) {
      Command: rover_stop();
      Assignment: drive_done = true;
    }

    When timeout Command: take_navcam();
    When drive_done Command: take_pancam();

    While true {
      When (LookupOnChange("temperature") < 0) Command:
turn_on_heater();
      When (LookupOnChange("temperature") > 10) Command:
turn_off_heater();
    }
  }
}

```

```

    }
  }
}

```

Example 3 : Ten pictures in PLEXIL Core

Our third example is another rover plan and illustrates node sequencing.

```

SafeDrive: {
  int pictures = 0;
  RepeatCondition: LookupOnChange(Rover.WheelStuck) == false;
  NodeList:
    OneMeter: {
      Command: Rover.Drive(1);
    }
    TakePic: {
      In Integer pictures;
      StartCondition: OneMeter.status == FINISHED && pictures
< 10;
      Command: Rover.TakePicture();
    }
    Counter: {
      InOut Integer pictures;
      StartCondition: TakePic.status == FINISHED;
      Precondition: 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.

Example 4 : Simple Drive in PLEXIL syntactic sugar

The root node is SimpleDrive. In this plan if TakeSample fails the outcome of the plan is "failure". This plan checks if the rover is "AtRock". If it is not at the rock then it issues a command to drive in 1m increments repeatedly until the rover is at the rock. Only after it is at the rock the command to take a sample is issued. The plan also checks to ensure that the rover is "AtRock" for the entire duration while it is taking a sample.

```

SimpleDrive: {
  PostCondition: TakeSample.outcome == SUCCESS;
  EndCondition: TakeSample.state == FINISHED;

  Sequence: {

    While (!LookupOnChange(At('Rock'))==1) {
      Command: drive(1.0);
    }

    Node TakeSample {
      Invariant: lookupOnChange(At('Rock')==0;
      Comamnd: takeSample();
    }
  }
}

```

Example 5 : Simple Drive in PLEXIL XML

This is equivalent to Example 4 above, but in PLEXIL XML.

```

<?xml version="1.0" encoding="UTF-8"?>
<PlexilPlan>
<Node NodeType="NodeList">
  <NodeId>SimpleDrive</NodeId>
  <PostCondition>
    <EQInternal>
      <NodeOutcomeVariable>
        <NodeId>TakeSample</NodeId>
      </NodeOutcomeVariable>
      <NodeOutcomeValue>SUCCESS</NodeOutcomeValue>
    </EQInternal>
  </PostCondition>
  <EndCondition>
    <EQInternal>
      <NodeStateVariable>
        <NodeId>TakeSample</NodeId>
      </NodeStateVariable>
      <NodeStateValue>FINISHED</NodeStateValue>
    </EQInternal>
  </EndCondition>
  <NodeBody>
    <NodeList>
      <Node NodeType="Command">
        <NodeId>Drive</NodeId>
        <PreCondition>
          <EQBoolean>
            <LookupNow>
              <StateName>At</StateName>
              <Arguments>
                <StringValue>Rock</StringValue>

```

```

        </Arguments>
    </LookupNow>
    <BooleanValue>0</BooleanValue>
</EQBoolean>
</PreCondition>
<RepeatUntilCondition>
    <EQBoolean>
        <LookupOnChange>
            <StateName>At</StateName>
            <Arguments>
                <StringValue>Rock</StringValue>
            </Arguments>
        </LookupOnChange>
        <BooleanValue>1</BooleanValue>
    </EQBoolean>
</RepeatUntilCondition>
<NodeBody>
    <Command>
        <CommandName>drive</CommandName>
        <Arguments>
            <RealValue>1.0</RealValue>
        </Arguments>
    </Command>
</NodeBody>
</Node>
<Node NodeType="Command">
    <NodeId>TakeSample</NodeId>
    <StartCondition>
        <EQInternal>
            <NodeStateVariable>
                <NodeId>Drive</NodeId>
            </NodeStateVariable>
            <NodeStateValue>FINISHED</NodeStateValue>
        </EQInternal>
    </StartCondition>
    <InvariantCondition>
        <EQBoolean>
            <LookupOnChange>
                <StateName>At</StateName>
                <Arguments>
                    <StringValue>Rock</StringValue>
                </Arguments>
            </LookupOnChange>
            <BooleanValue>1</BooleanValue>
        </EQBoolean>
    </InvariantCondition>
    <NodeBody>
        <Command>
            <CommandName>takeSample</CommandName>
        </Command>
    </NodeBody>
</Node>
</NodeList>

```

```

    </NodeBody>
  </Node>
</PlexilPlan>

```

Appendix A. Node State Transition Diagrams

The figure below is the legend for all the figures to follow. The yellow rectangles represent condition changes that cause a transition from a node state. Only the condition change explicitly represented causes the transition. The rectangles with bars represent the outcome of a node. The lilac diamonds represent checks. Transitions are represented by directed arrows. If there are multiple condition changes that may happen simultaneously, integers are used to represent the precedence order. A condition change with precedence order 1 gets priority over any other condition change. A condition change with precedence order 2 is processed if there is no condition change with priority 1 and so on. Some checks are a binary choice between True and False. Others are a choice between True, False, and Unknown (represented as T, F, and U respectively in the figures).

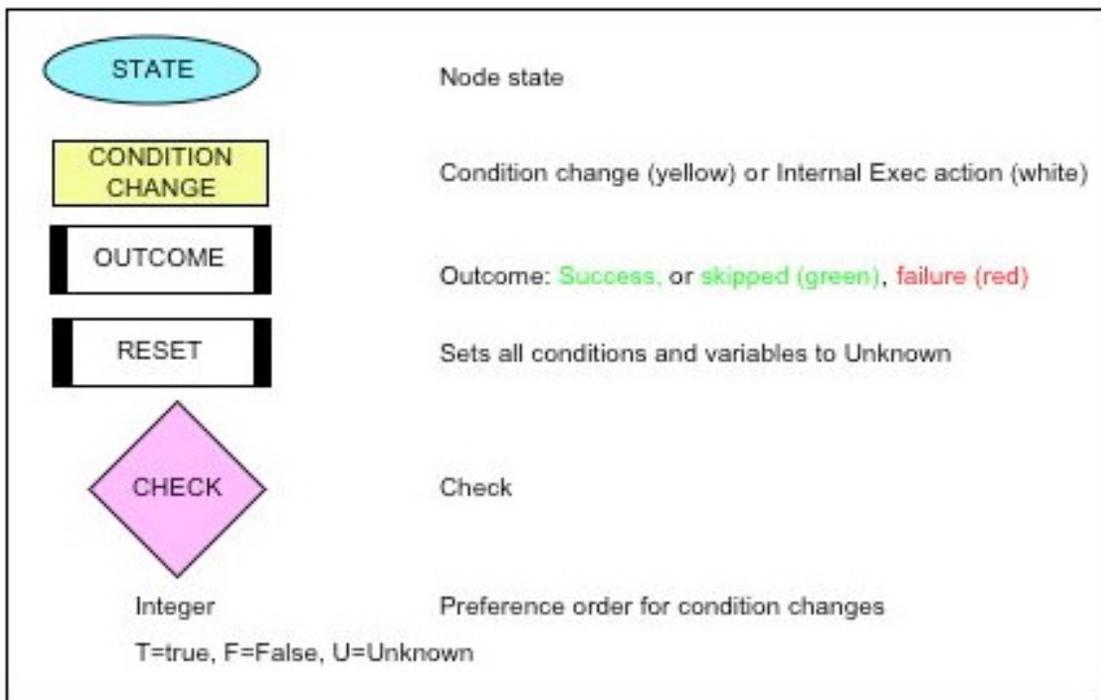


Figure 1: Legend

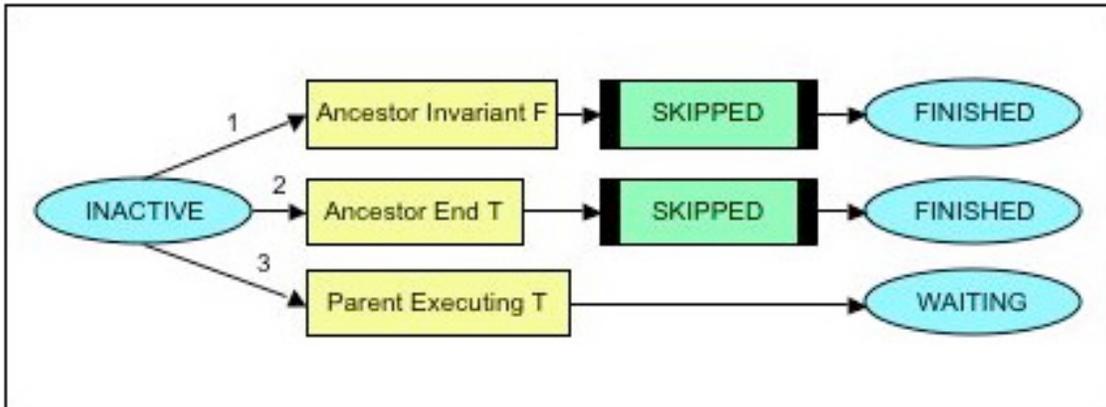


Figure 2: Transitions from state Inactive for all node types

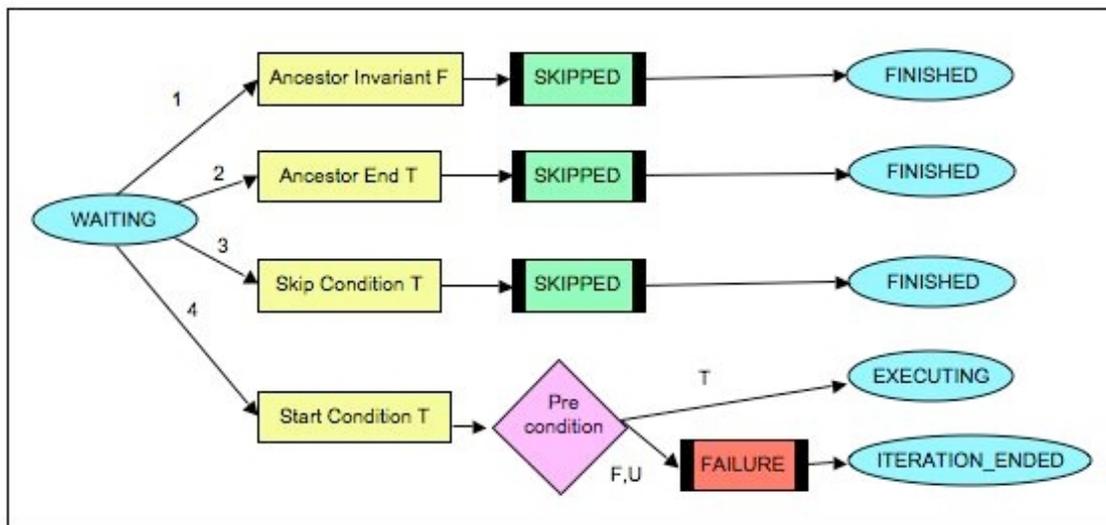


Figure 3: Transitions from state Waiting for all node types

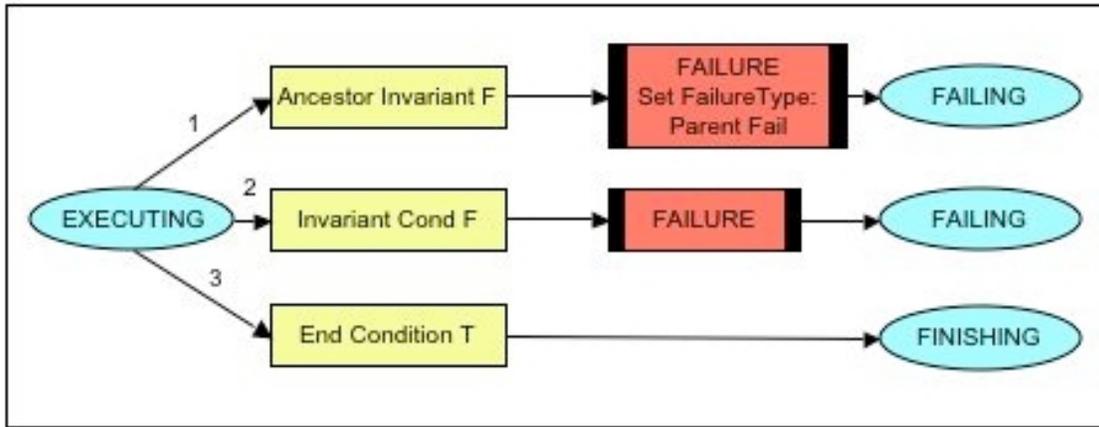


Figure 4: Transitions from state Executing for nodes of type NodeList

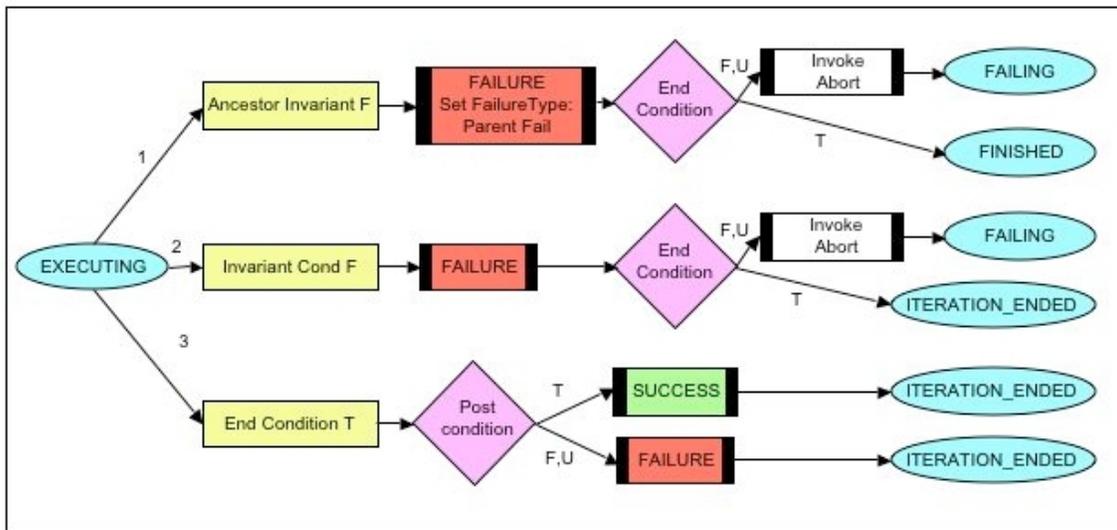


Figure 5: Transitions from state Executing for nodes of type Command, Update, or PlanRequest

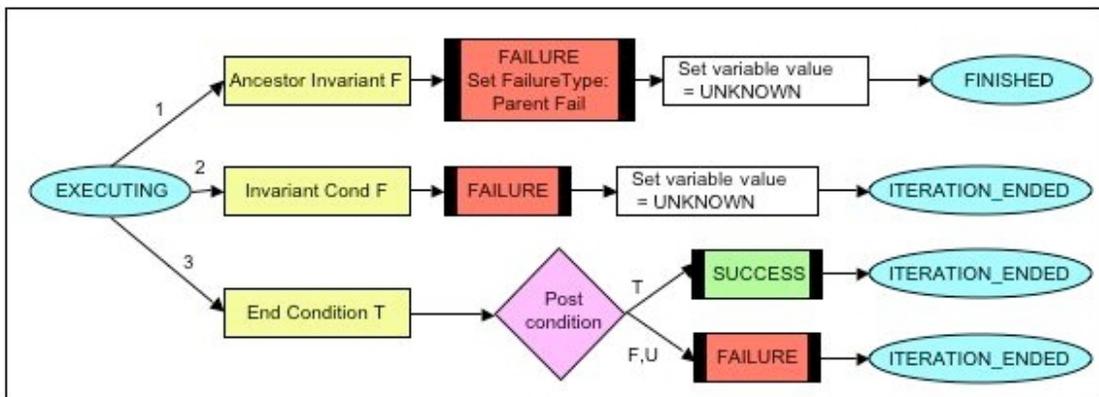


Figure 6: Transitions from state Executing for nodes of type Assignment

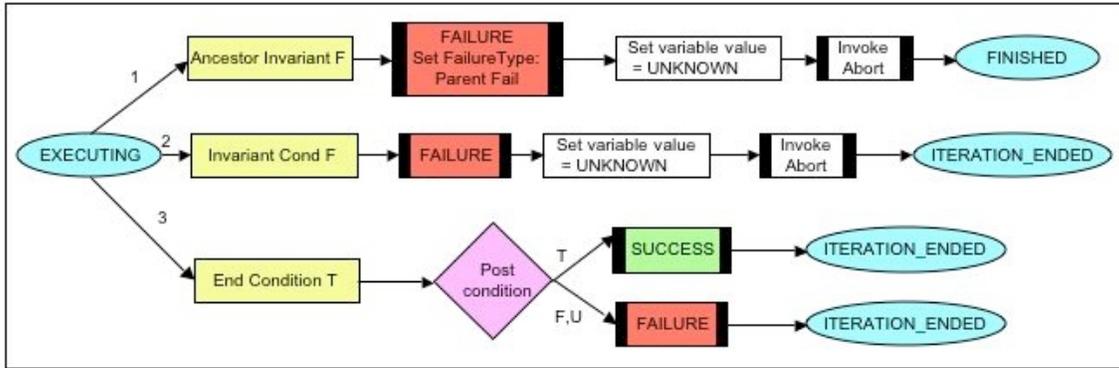


Figure 7: Transitions from state Executing for nodes of type FunctionCall

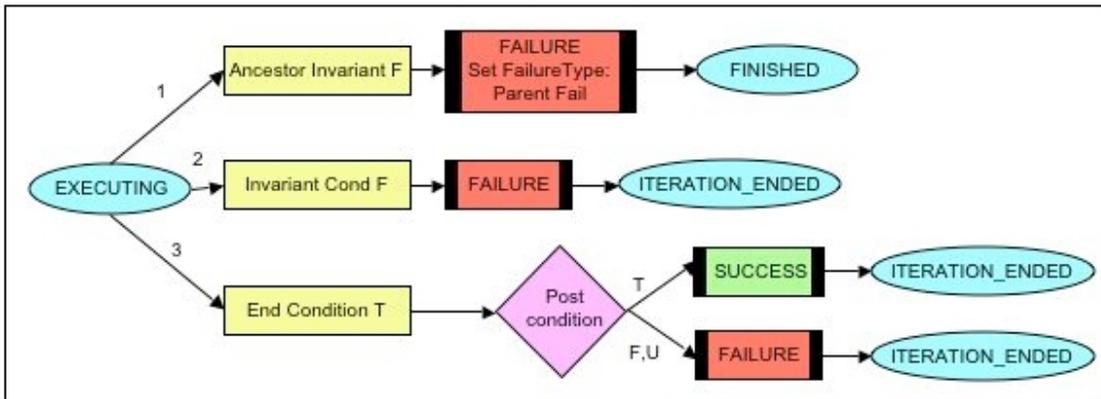


Figure 8: Transitions from state Executing for nodes of type Empty

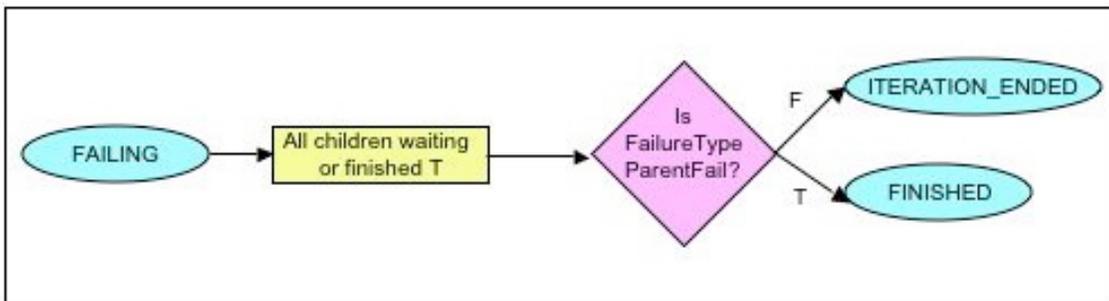


Figure 9: Transitions from state Failing for nodes of type NodeList

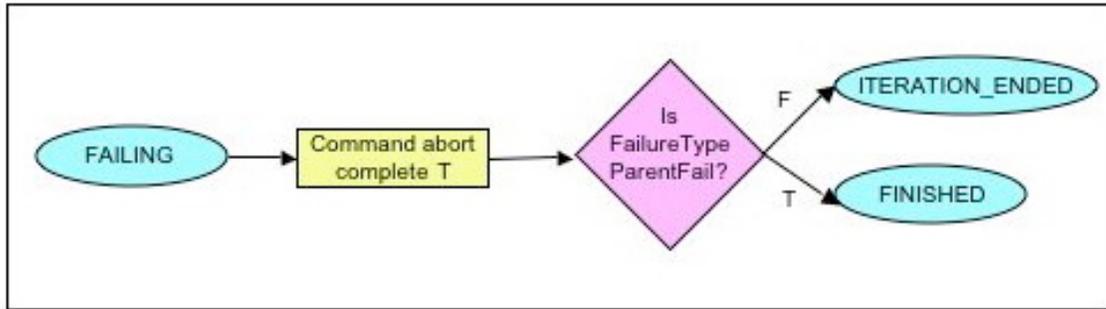


Figure 10: Transitions from state Failing for nodes of type Command, Update, or PlanRequest

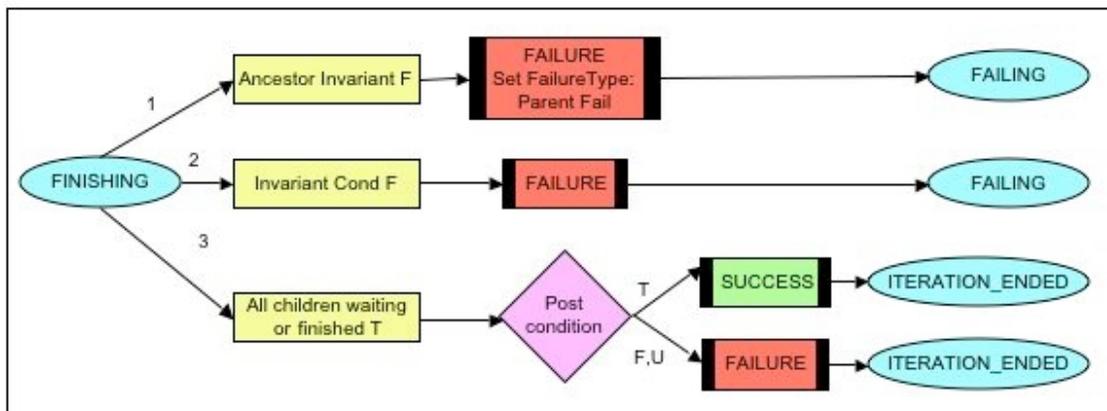


Figure 11: Transitions from state Finishing for nodes of type NodeList

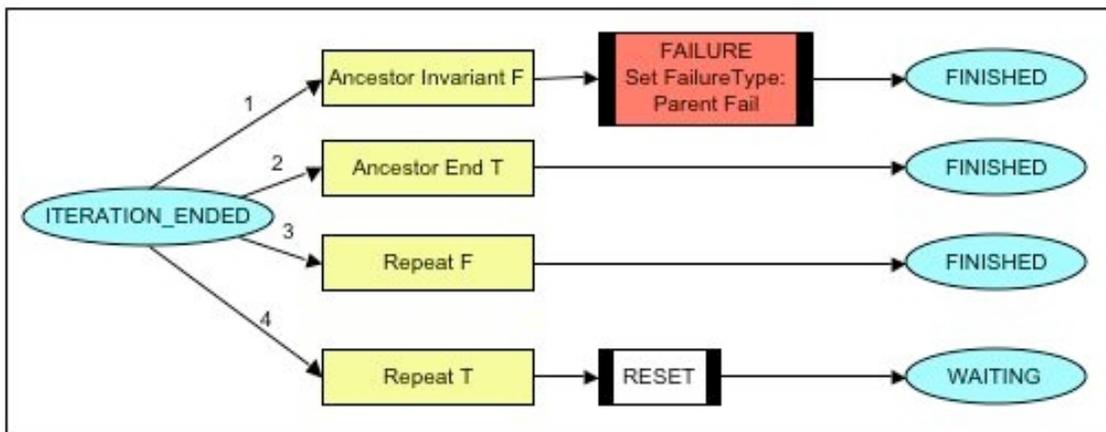


Figure 12: Transitions from state IterationEnded for all node types

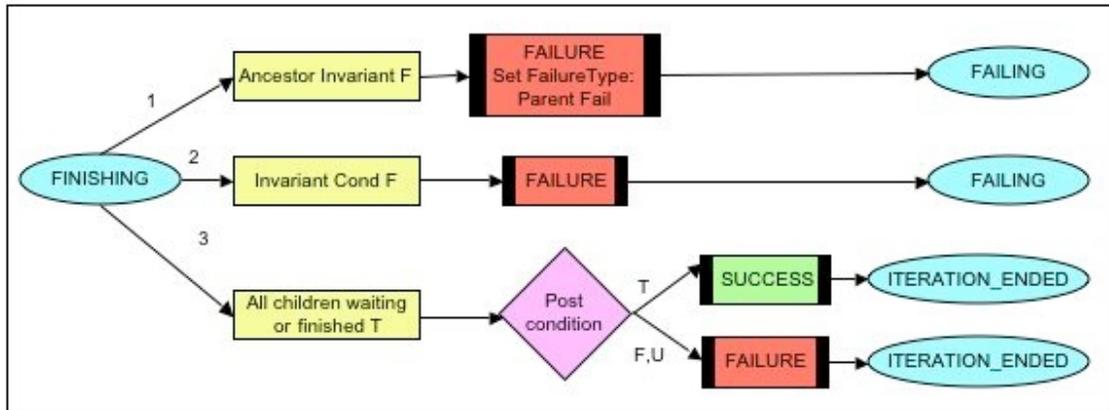


Figure 13: Transitions from state Finished for all node types

Appendix B. Relevant Publications

T. Estlin, A. Jonsson, C. Pascareanu, R. Simmons, K. Tso, V. Verma, Plan Execution Interchange Language (PLEXIL). NASA Technical Memorandum TM-2006-213483, April 2006.

V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso, “Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences”, International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS), 2005

V. Verma, A. Jónsson, C. Pasareanu, M. Iatauro, Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations, American Institute of Aeronautics and Astronautics Space 2006 Conference.

G. Dowek, C. Munoz, C. Pasareanu, Formal Semantics of a Synchronous Plan Execution Language, Workshop on Planning and Plan Execution for Real-World Systems: Principles and Practices for Planning in Execution at the International Conference on Automated Planning and Scheduling (ICAPS) 2007.

M. Bualat, L. Edwards, T. Fong, M. Broxton, L. Flueckiger,

S Lee, E. Park. V. To, H. Utz, V. Verma, C. Kunz, M. MacMahon, Autonomous Robotic Inspection for Lunar Surface Operations, Field and Service Robotics Conference (FSR) 2007.