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

PLAN EXECUTION INTERCHANGE LANGUAGE (PLEXIL) FOR EXECUTABLE PLANS AND COMMAND SEQUENCES

Vandi Verma[(1)], Tara Estlin[(2)], Ari Jónsson[(3)], Corina Pasareanu[(4)], Reid Simmons[(5)], Kam Tso[(6)]

[(1)]QSS Inc. at NASA Ames Research Center, MS 269-1 Moffett Field CA 94035, USA,
Email:vandi@email.arc.nasa.gov
[(2)]Jet Propulsion Laboratory, M/S 126-347, 4800 Oak Grove Drive, Pasadena CA 9110, USA ,
Email:tara.estlin@jpl.nasa.gov
[(3)] USRA-RIACS at NASA Ames Research Center, MS 269-1 Moffett Field CA 94035, USA,
Email:ajonsson@arc.nasa.gov
[(4)]QSS Inc. at NASA Ames Research Center, MS 269-1 Moffett Field CA 94035, USA,
Email:pcorina@email.arc.nasa.gov
[(5)]Robotics Institute, Carnegie Mellon University, Pittsburgh PA 15213, USA,
Email:reids@cs.cmu.edu
[(6)]IA Tech Inc., 10501Kinnard Avenue, Los Angeles, CA 9002, USA,
Email: tso@ia-tech.com

ABSTRACT

Space mission operations require flexible, efficient and reliable plan execution. In typical operations command sequences (which are a simple subset of general executable plans) are generated on the ground, either manually or with assistance from automated planning, and sent to the spacecraft. For more advanced operations more expressive executable plans may be used; the plans might also be generated automatically on board the spacecraft. In all these cases, the executable plans are received by a software system that executes the plan. This software system, often called an executive, must ensure that the execution of the commands and response of the fault protection system conforms to pre-planned behaviour. This paper presents a language, called *PLEXIL*, that is designed specifically for flexible and reliable command execution. It is designed to be portable, lightweight, predictable, and verifiable, and at the same time it is more expressive than command sequencing languages currently used on space missions.

1. INTRODUCTION

All space missions require *execution systems* that execute commands and monitor the environment. Such execution systems vary in sophistication, from those that execute linear sequences of commands at fixed times, to those that can plan and schedule in reaction to unexpected changes in the environment.

The level of autonomy and type of control that execution systems are designed for varies greatly [13]. A spacecraft controlled primarily by humans must continuously monitor critical systems and report important events. An execution system is needed for this task since it is impossible for a small team to manually keep track of all sub-systems in real time. Complex manipulators and instruments used in human missions must also execute sequences of commands to perform complex tasks. Similarly, a rover operating on Mars must autonomously execute commands sent from Earth and keep the rover safe in abnormal situations.

Execution systems realize pre-planned actions in the real world. Execution systems are particularly useful in the presence of uncertainty. Classical execution functions include selecting an action from a set of possibilities based on the current state of the robot and environment and on the outcome of previous actions. The capabilities of an execution system typically include hierarchical task decomposition, coordinating simultaneous actions, resource management, monitoring of states, relaying command status, and exception handling. One way to view an execution system is as an onboard system that takes a plan that assumes a certain level of certainty and expected outcomes and executes it in a possibly uncertain and dynamic environment.

An *execution language* is a representation of commands and plans that provides a representation for reasoning about required robot and environment state as well as the effects of executed commands. It takes into account the interdependence between actions, in terms of temporal precedence and other constraints, such as resource contention. An execution language also provides a representation for monitoring mission constraints and encoding appropriate responses if these constraints are violated.

Plan execution frameworks vary greatly, due both to different capabilities of the execution systems and due

Proc. 'ISAIRAS 2005 Conference', Munich, Germany,
5-8 September 2005 (ESA SP-603, September 2005)

to relationships with associated decision-making frameworks. The latter dependency has made the reuse of execution and planning frameworks difficult, and has all but precluded information sharing between different execution and decision-making systems.

As a step in the direction of addressing these issues, we are developing a general plan execution language, called the Plan Execution Interchange Language (PLEXIL). PLEXIL extends many execution control capabilities of other systems. It is capable of expressing concepts used by many high-level automated planners and hence provides an interface to multiple planners.

The key characteristics of PLEXIL are that it is compact, semantically clear, and deterministic given the same sequence of inputs. 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.

PLEXIL includes a domain description that specifies command types, task expansions, constraints, etc., as well as feedback to the higher-level decision-making capabilities. In addition, PLEXIL includes a graphical user interface that builds upon the Eclipse framework [10] that can be used to manually create PLEXIL plans.

This paper provides an overview of the grammar and semantics of PLEXIL. Further details may be found in [7]. It also outlines ongoing work on implementing a universal execution system, based on PLEXIL, using state-of-the-art rover functional interfaces and planners as test cases.

2. BACKGROUND

We are using several existing implementations of rover control systems to drive the design of PLEXIL. One of these is the Coupled Layer Architecture for Robotic Autonomy (CLARAty) [14], which is a two layered software architecture that was developed to enable both a plug-and-play capability and a tighter coupling of high level decision making planners and the interface to hardware. The CLARAty architecture has successfully enabled interoperability at the functional layer, which is the interface to the hardware. The development of the PLEXIL-based execution system in the CLARAty architecture will provide a level of interchangeability for the decision layer.

As test cases for the general PLEXIL execution engine, two different types of planners will be utilized for generating PLEXIL plans and re-planning based on feedback information. One is a constructive planner, called PICO [6], that generates long-term contingent plans, which are flexible.Since plans contain contingencies, PICO can be used off-board the rover. The other is an iterative repair-based planner, called CASPER [5], which generates fixed plan instances but can easily re-plan in the face of changes. CASPER is typically used as an on-board planner. Currently, each planner interacts with the hardware using different executives. The executive used with CASPER is based on TDL [11], which is an extension of the C++ programming language that includes syntax for task definition, task expansion, temporal constraints, concurrent execution, monitoring, and fault recovery. PICO uses the Contingent Rover Language (CRL) [3] executive which uses a hierarchical representation to represent simple and floating branches, nesting, flexible time, and state and resource conditions, but does not support loops and periodic tasks, or have a mechanism for providing feedback to planners.

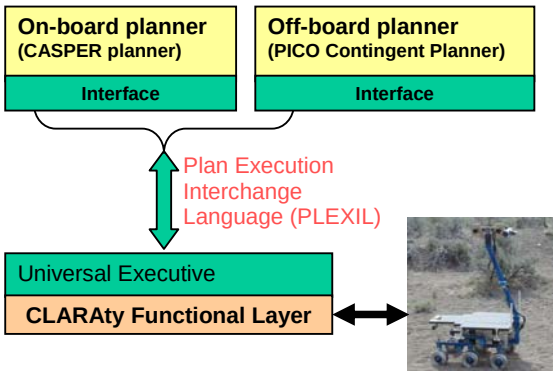

Figure 1: Architectural design where PLEXIL is shown to interface with the CASPER and PICO planners

3. TRADITIONAL SPACECRAFT OPERATIONS

In traditional spacecraft operations control commands are translated into simple sequences and uploaded to the remote spacecraft.

An example of this mode of operations is the *Mars Exploration Rover* (MER) mission [12]. In the MER mission, plans are built in a mixed-initiative fashion on Earth. In this process, tools such as the Science Activity Planner (SAP) [9] and Mixed-initiative activity planner for the Mars Exploration Rover mission (MAPGEN) [2] combine automated

capabilities and interfaces for human plan construction. The resulting plans are translated into command sequences and uploaded to the Mars rovers.

The on-board flight software on the rovers then executes the command sequences. In cases where the outcome of complex commands, such as an arm placement, is uncertain, the command sequence is generated so that the plan terminates at this point and the rover takes a picture and sends it back to Earth. Rover experts then analyze these pictures and the telemetry the rover sends back in order to determine what state the rover is in. Based on this analysis, a new sequence of commands is generated. The MER rovers have been extremely successful and have been operating on Mars for over a year. However, the mode of operation is fairly cumbersome in part because the current execution language is fairly impoverished. For example the current language does not allow naturally for contingencies, floating plan additions, loops, etc.

4. REQUIREMENTS FOR EXECUTION LANGUAGE

The case studies described above have several features in common that drive the design of an executive language. For one, the execution system that executes the language must be *efficient*. The language should be interpretable by an execution system that can fit on a flight processor and must run fast to meet certain commanding and fault handling timing requirements.

The execution language should *support the current mode of operation, as exemplified by MER*, which involves human operators generating every single sequence of commands in an offline manner. In addition, it should also be able to support general contingent plans that encode alternative sequences of actions depending on a set of possible situations Mars may dole out to the rover.

The language should be *modular* so that commands for different operations can be encoded independently. In addition it should *support multiple automated planners*. This promotes reuse and good software engineering practices.

The language needs to be *expressive* enough to represent *simple branches, floating branches, loops, time and event driven activities, concurrent activities, sequences*, and *temporal constraints*. At the same time the syntax should be simple and uniform, making plan interpretation *simple* and efficient, while enabling the application of *validation* and *testing* techniques. The language should be *semantically clear* and *deterministic given the same sequence of inputs*.

The execution language should be *amenable to mixed-initiative planning*. In other words, it should be possible to translate plans from an automated planner into plans in the execution language and in addition it should be possible to manually edit these plans if necessary.

One important aspect of the interface between planning and execution systems is the *feedback from the executive to the planner*. This is particularly important for on-board planning systems, which have a tight integration with the executive. The language should provide support for specifying what information should be returned to the planner upon successful or unsuccessful completion of tasks. For certain planners feedback should also be provided during task execution.

The language *should easily and naturally interface to other tools*, such as path planners, diagnosis tool, third-party libraries (e.g., for ephemeris data), etc.

5. PLEXIL

In our design of the PLEXIL plan execution language, we have endeavoured to incorporate all the requirements listed above. PLEXIL is based on a hierarchical representation of execution *nodes*. Execution nodes describe both initiation of real-world actions, and the control of execution. The nodes are arranged into hierarchical trees where leaf nodes are action nodes and internal nodes are control nodes. The execution of each node is governed by a set of conditions, such as when the node gets activated and when it is done. *Conditions* capture temporal relationships, as well as internal and external information, such as when the temperature gets above a certain threshold or when a rock has been seen. When action nodes are executed, commands are sent to the rover, whereas when control nodes are executed, they are expanded to the next level of nodes in the tree.

For example, consider the following PLEXIL plan:

```
Node:{
    NodeId: SafeDrive;
    Repeat-until-condition:
            Lookup{“Rover:wheelStuck”}==false;
    NodeList:{
        Node:{
            NodeId: DriveOneMeter
            Command: Rover:Drive(1);
        }
    }
}
```

The plan has one action node (with identifier `DriveOneMeter`) that drives the rover one meter by invoking the command `Rover:Drive(1)` in the

functional layer. The plan also has one control node (`SafeDrive`) that keeps repeating the action node until the rover is stuck. This is specified by a `Repeat-until-condition` that requests information from the functional layer, via a `lookup`.

5.1 Language Features

We describe now some of the main features of PLEXIL. In addition to execution nodes and the associated conditions, PLEXIL supports declared variables, assignments to these variables and explicit node interfaces.

Nodes

There are three types of nodes in PLEXIL.

- *Node-list nodes* are internal nodes that simply contain a list of children nodes.
- *Command nodes* are action nodes that contain command calls to the functional layer to initiate real world actions. A command call specifies a command name and a list of arguments, conforming to the domain description.
- *Assignment nodes* are action nodes that perform assignments to declared variables (i.e. internal actions).

In addition, each node has a set of *Node Attributes*. We discuss some of them below.

- *NodeId*: A unique symbolic name
- A set of conditions (i.e. Boolean expressions) that drive the node execution. s*tart condition, end condition* and *repeat-until condition* determine if the execution of a node should start, end or repeat, respectively. *pre-condition, post-condition, and invariant-condition* determine whether the node is executing normally; if any of these conditions fail to evaluate to true, the node execution is aborted with a failure indication.
- *Variables:* List of local variable declarations. These variables can be used in the assignments and conditions in the sub-tree of the current node (see next bullet).
- *Interface:* List of variables declared in an ancestor, "passed" to the node. A child of a node only has access to the variables declared in the parent that are explicitly passed via the interface (details in [7]).

To control execution, PLEXIL node elements can access information from external, world states, via *lookups*. Each external state is identified by a domain-specific name. For example, `Lookup{"Rover:wheelStuck"}` returns the value of state "Rover:wheelStuck" when the lookup is done. Lookups can appear in assignments or in conditions. There are three classes of lookups:

1) *Frequency based* lookups provide the requested value at a specified frequency. e.g. lookup temperature every 10 minutes.
2) *Event based* lookups provide the requested value only when it changes. e.g. return the temperature when it falls below 0ºC.
3) *One time* lookups provide the value at the time they are evaluated. e.g. lookup the current temperature.

A tolerance may also be specified with frequency and event based lookups.

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.

In addition to the external states, which are accessed via lookups, a PLEXIL plan has access to the values of a number of *internal variables*, such as the start and end times for the execution of each node, the status of completed executions (success or fail), etc.

Types

The domain of variables is extended with additional values (unknown, fail) to account for failure. As a result the conditions are interpreted using multiple-valued logic.

Syntactic Enhancements

To increase usability the language includes syntactic enhancements for constructs such as if-then-else, while-loop, macros and tail recursion. The instantiation of the syntactic enhancements is defined in terms of the core language features.

Domain Description

The domain description is an external library that contains the names of state variables, function names associated with commands, etc. The domain description defines the interface of the executive with the functional layer. In addition, it can declare any general function that may be used for example to perform complex mathematical computations.

5.2 Node Execution

As mentioned, the execution of PLEXIL nodes is governed by the various conditions on the nodes, e.g., start conditions, end conditions, and invariant conditions. Due to the nature of these conditions, the execution is driven by events, which include time passing, rover and world state changes, and changes in the values of declared and internal variables.

The semantics of node execution is defined in terms of node states and event based transitions. The states that the node can be in are waiting, executing, finishing, iteration ended, and finished.

To ensure the right execution context pre-, post-, and invariant conditions are checked at the start, duration, and end of the execution of each node. If any of these conditions fail the node fails and transitions to state finished.

Execution of each node either succeeds or fails and the outcome affects further the execution of the plan. For example, the failure of a node may trigger the execution of another node that performs some actions to recover from the failure.

The execution of a plan proceeds as follows. All top-level nodes are created immediately upon start of plan execution. Any other node gets created when its parent-node's *start condition* becomes true (and the parent's *precondition* is true at that time). A newly created node is in *waiting* state.

In the *waiting* state the *start condition* of a node is monitored. Once the *start condition* of a node becomes true it is ready for execution and it transitions to the *executing* state.

In the *executing* state, if the node is a *command* or an *assignment* then corresponding command or assignment is performed. Otherwise the node creates its children which all start in *waiting* state.

If the *end condition* is met the node transitions from state *executing* to *iteration ended*. The *repeat-until condition* is then checked. If the condition is false the node transitions back to the *waiting* state else it transitions to state *finished*.

There is a distinction between states *iteration ended* and *finished*. Some nodes must execute multiple times whenever the *start condition* is satisfied, e.g. a node that heats a component whenever the temperature falls below a certain level. State *iteration ended* represents the end of a single iteration of node execution. State *finished* represents the end of "all" iterations of the node and the *start condition* of the node iso longer monitored. Note that nodes that execute only once have a *repeat-until condition* that is set to *true*. Hence the state *iteration ended* immediately transitions to state *finished*.

5.3 XML Schema, Grammar, Conversion to Java for Verification, Plan editor

PLEXIL plans are written in XML [15] for increased portability. The XML Schema for PLEXIL plans was developed using the XMLSpy [1] Schema editor. Castor [4] was used to generate Java code and a parser for XML plans – which can be analyzed with the Java PathFinder [8] verification tool. This code will also be used for automatic test input generation (plans that conform to the schema will be generated automatically based on the PLEXIL grammar) for analysis of XML.

A PLEXIl plan editor has been developed, which allows users to create new plans or to modify existing ones, through a graphical interface. In the future the Java PathFinder tool will be used in conjunction with the plan editor – the intention is to check the newly modified plans.
For further details on the syntax, semantics, and tools for PLEXIL see [7].

6. UNIVERSAL EXECUTIVE

In order for the PLEXIL language to be useable, an execution system is required to interpret it. We are currently developing such an execution system, called the *Universal-Executive*. This is in a collaborative effort between researchers at NASA Ames Research Center, NASA's Jet Propulsion Laboratory and Carnegie Mellon University. In particular, the Universal Executive is being designed to facilitate reuse and inter-operability of execution and planning frameworks.

The input to the Universal-Executive will be a PLEXIL representation of an execution control instance and a description of relevant domain information. Having an interpreted encoding of the domain information makes the Universal Executive independent of the specific mission context in which it is being used. The Universal Executive will be capable of executing multiple nodes concurrently. When action nodes are executed, commands are sent to the rover, whereas when internal nodes are executed, they are expanded to the next level of nodes in the tree.
During plan execution, PLEXIL also enables monitoring of different state and resource information as well as command status. This information can be used to direct execution and/or can be relayed to other decision-making capabilities (i.e., planners).

The expressiveness of the language enables the Universal Executive to handle dynamic outcomes and environmental uncertainty. The executive can also provide execution information and outcomes back to higher-level systems. Consequently, it can be used both as a stand-alone execution-only system, and as an execution system coupled with an external planner [13].

7. EXAMPLE

Consider the stylized example shown in ***Table 1***.

Figure 2 and ***Figure 3*** show an example where alternate options in the same PLEXIL plan above are

- **Drive rover**
  - Until target in view, or
  - Until time-out at time 10
- **Take Navcam**
  - After drive, if drive timed out
- **Take Pancam**
  - After drive, if target in view
- **Heat up to 10 ºC**
  - Whenever temperature below 0ºC

Table 1: Example plan

executed based on different sensed states of the world. In ***Figure 2*** the drive times out and a Navcam image is taken. Execution in ***Figure 3*** follows an alternate path since in this scenario the target is reached. A Pancam image is taken instead based on the same PLEXIL plan. The domain specification given for this scenario contains the following mapping:

```
Commands: void rover_drive(int
speed);
          void rover_stop();
          void take_navcam();
          void take_pancam();
          void turn_on_heater();
StateNames: temperature,
target_in_view;
```

Here, `Commands` are function calls provided by the low level interface to the rover (functional layer) and `StateNames` are sensed or derived values that can be accessed from the functional layer.

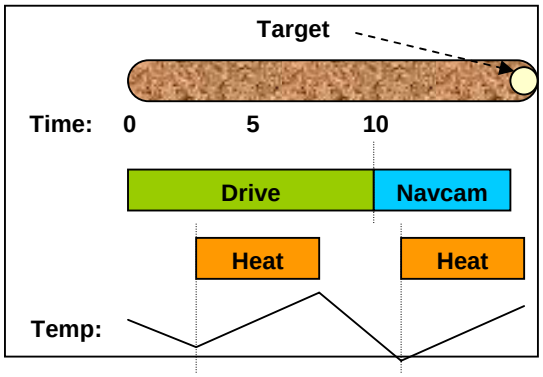

Figure 2: An example execution where the drive times out.

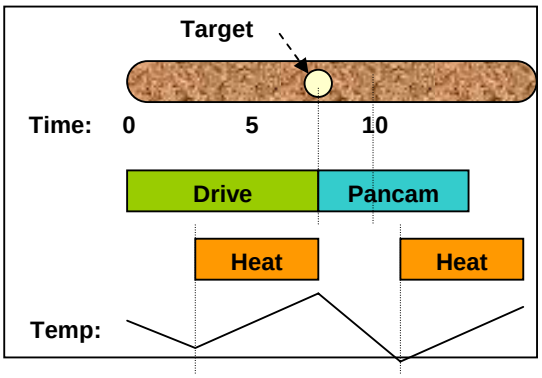

Figure 3: An example execution where the target is reached.

Note that the actual code sent to the Universal Executive will be in XML, which is a fairly standard representation for information exchange, but is not easy to read.
Example PLEXIL syntax for executing the above scenario is shown in Table 2.

8. CONCLUSION

We believe that PLEXIL could provide richer representations for executable plans on spacecraft, with minimal added effort or risk.

Using PLEXIL and a corresponding execution system would enable context-sensitive control that can take into account the current state of the rover and environment and the outcome of previous actions, while at the same time including complex monitoring of plan specific constraints. PLEXIL offers a fully general contingency plan representation. For example completely different sequences may be executed in response to different states. In addition it also has floating contingencies, which are commands that are not in the sequences, but get executed when certain conditions are met.

Each PLEXIL plan can potentially be verified individually. In addition, it is possible for human experts to evaluate the range of scenarios that may result from a PLEXIL plan.

This is ongoing work and we are reporting on part of the language that is being defined. There are related execution systems and languages such as APEX [16], *Reactive Model-based Programming Language* (RMPL) [17], and TDL [11].

9. REFERENCES

[1] Altova, *XMLSpy toolkit*, http://www.altova.com

[2] Bresina J., Jónsson A, Morris P., Rajan K., *Activity Planning for the Mars Exploration Rovers*, The International Conference on Automated Planning & Scheduling (ICAPS), 2005.

[3] Bresina J.L. and Washington, R., *Robustness via Run-time Adaptation of Contingent Plans*, In Proceedings of the AAAI-2001 Spring Syposium: Robust Autonomy. Stanford, CA

[4] Castor, The Castor Project, http://www.castor.org

[5] Chien S., Knight R., Stechert A., Sherwood R., Rabideau G., *Using Iterative Repair to Improve Responsiveness of Planning and Scheduling*, International Conference on Artificial Intelligence Planning Systems (AIPS 2000). Breckenridge, CO. April 2000

[6] Dearden R., Meuleau N., Ramakrishnan S., Smith D., and Washington R., *Incremental Contingency Planning*, ICAPS-03 Workshop on Planning under Uncertainty, Trento, Italy, June 2003.

[7] Estlin T., Jónsson A., Pasareanu C., Simmons R., Tso K., Verma V., *PLEXIL – The Language*, NASA Technical Memorandum.

[8] Java Pathfinder , http://javapathfinder.sourceforge.net

[9] Jeffrey S. Norris, Mark W. Powell, Marsette A. Vona, Paul G. Backes, Justin V., *Mars Exploration Rover Operations with the Science Activity Planner*, International Conference on Robotics and Automation 2005.

[10] Object Technology International Inc., *Eclipse Platform Technical Overview*, White Paper, July 2001.

[11] Simmons R. and Apfelbaum D., *A Task Description Language for Robot Control*, Proceedings of Conference on Intelligent Robotics and Systems, Vancouver Canada, October 1998.

[12] Squyres S., *Roving Mars: Spirit, Opportunity, and the Exploration of the Red Planet*, Hyperion Press.

[13] Verma V., Jónsson A., Simmons R., Estlin T., Levinson R., *Survey of Command Execution Systems for NASA Robots and Spacecraft*, Plan Execution: A Reality Check Workshop at The International Conference on Automated Planning & Scheduling (ICAPS), 2005

[14] Volpe R., Nesnas I. A. D., Estlin T., Mutz D., Petras R., Das H., *The CLARAty Architecture for Robotic Autonomy*. Proceedings of the 2001 IEEE Aerospace Conference, Big Sky Montana, March 10-17 2001.

[15] Yergeau F., Cowan J., Bray T., Paoli J., Sperberg-McQueen C., Maler E., *Extensible Markup Language (XML 1.1)*, W3C Recommendation, 4th February 2004.

[16] Freed M., *Managing Multiple Tasks in Complex, Dynamic Environments*. In Proceedings of the 1998 National Conference on Artificial Intelligence. Madison, WI. 1998

[17] Williams B. C., Ingham M., Chung S. H., and Elliott P. H., January 2003. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers, invited paper in Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software, vol. 9, no. 1, pp. 212-237.

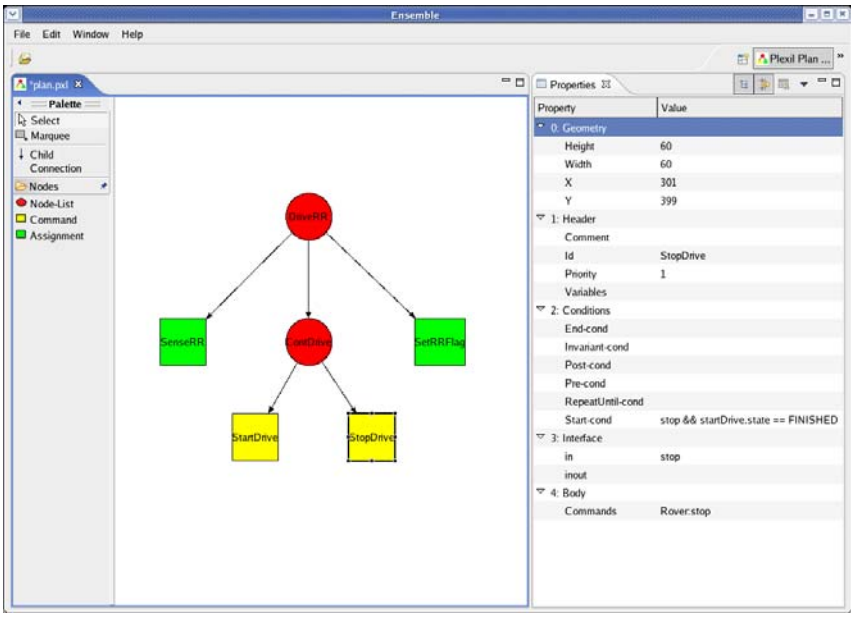

Figure 4: Plexil Plan Editor

```
              Node: {
               NodeID: DriveToTarget;
               Boolean drive_done, timeout;

               NodeList: {

                Command: rover_drive(10);

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

                When
                Lookup{"target_in_view",Frequency=10}==true;
                 Sequence:{
                   Command: rover_stop();
                   Assign: drive_done=true;
                 }

                When timeout==true
                 Command: take_navcam();

                When drive_done==true
                 Command: take_pancam();

                Node:{
                     NodeID: Heater;
                     StartCondition: Lookup{"temperature"}<0
                     EndCondition: Lookup{"temperature"}>=10
                  RepeatUntilCondition: false;
                  Command: turn_on_heater();
                 }
               }
}
```

Table 2. PLEXIL for example from Table 1