AN ABSTRACT OF THE THESIS OF

Paul Strauss for the degree of Master of Science in Computer Science presented on June 8, 2009.

Title: Executable Semantics for PLEXIL: Simulating a Task-Scheduling Language in Haskell

Abstract approved: ____

Martin Erwig

An interdisciplinary study into the theory of design decisions has yielded a model for tracking design changes in hardware/software systems, but it still needs to be applied to a larger system to test its efficiency at tracking important data. This thesis creates an implementation of PLEXIL, a language in development at NASA for controlling various hardware systems, as a testbed for applying the model of design decisions. This PLEXIL implementation is embedded in the functional language Haskell to take advantage of its static typing and lazy evaluation, and it accurately follows the semantics defined by NASA. The external world representation in this thesis improves upon NASA's current simulator through the definition of new data types for more dynamic runtime behavior.

Executable Semantics for PLEXIL: Simulating a Task-Scheduling Language in Haskell

by

Paul Strauss

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 8, 2009
Commencement June 2009

Master of Science thesis of Paul Strauss presented on June 8, 2009.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

Paul Strauss, Author

ACKNOWLEDGEMENTS

This thesis would not have been possible without the assistance of many other people. First, I thank my advisor, Martin Erwig, for providing me with constant insights into the realm of functional programming. I also thank my committee, Irem Tumer, Tim Budd, and Harry Yen, for attending meetings with me to help guide the design of my work.

I would like to thank the other members of my project group, Eric Walkingshaw and Jon Mueller, for their assistance with realizing my design in the context of our research. I also want to thank the PLEXIL group at NASA for willingly helping me understand their work on a deeper level.

I extend my gratitude to my research group, Chris Chambers, Tim Bauer, Chris Bogart, and Weinian He, for teaching me the intricacies of Haskell programming and helping me improve my presentation style.

I appreciate my family, Robin, Ted, Marie, and Dusty, for encouraging and supporting me through my college career.

I finally thank Jason Siefken for giving me a constant motivation to finish this thesis to the best of my ability.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

Executable Semantics for PLEXIL:
Simulating a Task-Scheduling Language in Haskell

Chapter 1 – Introduction

Starting in 2007, a collaboration between members of the Mechanical Engineering and Computer Science departments at Oregon State University performed an exploratory study of hardware/software systems [26]. The goal was to see *why* errors occur during the process of system redesign and *how* to prevent such errors. The study produced theoretical results for a model that allows description of designs, manipulation of attributes, and documentation of decisions, which are summarized in Section 1.1.

While the model was in place for tracking design decisions, finding an appropriate real-world application to test this model proved to be a difficult task. The project needed to include hardware and software design components that were still in the initial design phase. Searching led to the collaborative projects of NASA's PLEXIL and Universal Executive software system and the automated K10 rovers, described in Section 1.2.

This thesis explores the design of the PLEXIL language and Universal Executive semantics. It discusses the necessary components for developing a PLEXIL program and accurately executing such a program on an external system. It also provides language support for defining the surrounding hardware system. We will use this language to design a simulator of hardware interactions with the external world, improving upon NASA's script-based simulations with more dynamic run-time behaviors. With this design, we can provide an accessible testbed with which to test the effectiveness of the theories of design decisions.

1.1 Theory of Design Decisions

During the study of hardware/software design documentation, the research team developed a cursory language for describing and recording design decisions [26]. This language provides a way to model objects, which can be a combination of physical devices or software code, and tracking decisions made while developing the object model.

Object descriptions can be created with the following definitions.

$$O ::= \{A_1 \ldots A_n\}$$

$$A ::= i : \{V_1 \ldots V_m\}$$

$$V ::= v \mid O$$

Any object O can have a set of attributes. The division into attributes is known as *object* or *structure branching*. Each attribute, A, has an identifier and a set of possible values V, referred to as *choices*. Each value V can be a final value or another subobject. This gives us a tree-like structure for defining a system as a set of subcomponents.

As an example, imagine a shovel. It consists of a blade and a handle. The blade could be made of metal or plastic. The handle can be wood or metal, and it may have grips or not. We can formulate this idea with the object model in the following way.

```
shovel_obj = {"handle" : handle_obj ; "blade" : blade_obj}
handle_obj = {"material" : Wood, Metal ; "has grips" : Yes, No}
blade_obj = {"material" : Metal, plastic_obj}
plastic_obj = {"color" : Black, Red}
```

This quick example demonstrates how we can model a design using a top-down approach. We can naturally divide the entire object in its parts and add new attributes as necessary. However, this model alone is not enough to store all information we want about our design.

Consider the case where we want to track the total weight of the shovel design, since heavy shovels are harder to use. Weight is a feature that pertains to multiple attributes of the object, and the current tree structure does not include an clear way to store these cross-cutting constraints.

The model proposes an additional *rule*-based system for formulating more complex constraints. A rule has two parts. First, a rule needs to know whether a given object tree exhibits a certain property. If so, it needs to know an action to take to correct the issue. Designers can use rules to apply a much more general set of design constraints regardless of the tree structure.

With this model, we can define a decision as a transformation that restricts the choices of an attribute. The effect of a decision is noted by the specific removal of some portion of the object tree.

In the basic case, we would use this language to create a design tree, and then apply decisions until we produce a tree with only one value per attribute. This works fine for initial design implementation, but we would like to extend this design to allow for design reuse.

We assume that “most design is done by ‘redesign’ [26],” where design objects are usually constructed by modifying similar objects. In the process of modification, designers are possibly undoing previous decisions. With the shovel, it may be simple to see what effects may occur by modifying previous decisions, but it may not be so clear in more complex examples.

For the purpose of design reuse, *annotations* can be attached to a decision. Annotations describe why a decision was made. They should persist with the object tree as long as the associated decision is applied.

By adding annotations to decisions, designers have a way to add extra notes about their decisions. Should another user later try to undo a decision, he or she can see a log of potential issues that could arise. For instance, in the shovel example, we could add the annotation “Using wood handle to prevent exceeding the weight restriction.” Later on, if a user tries to change the handle material, the system could present a warning message about breaking the weight restriction.

Forcing users to add their own annotations to decisions may lead to poor documentation, either from forcing comments on trivial changes or from lack of understanding of the situation. Finding the right level of interaction is left as an open issue.

The research of design decisions seemed to work well on toy applications, such as the shovel example. However, it is important to check the model's effectiveness on larger scale systems. PLEXIL and K10 rovers were upcoming projects in development at NASA, which makes as an ideal testing ground for tracking issues between hardware and software designs through their design and integration phases.

1.2 PLEXIL and the K10 Rover

Plan Execution Interchange Language (PLEXIL) is a task scheduling language in development at NASA. It boasts a small set of syntax rules, yet it can be expressive enough to design complicated control structures [23]. The execution of a PLEXIL plan is operationally deterministic, meaning that the state of the executing program at any point in time can be determined assuming we know how the external world is affected by the running program.

PLEXIL comes with a lightweight interpreter, the Universal Executive [24]. As an interpreted language, PLEXIL does not need to be compiled to a binary file, which allows users to alter PLEXIL plans during execution.

PLEXIL has a variety of upcoming uses, including the K10 automated rovers, the International Space Station power system, and the DAME arctic drill [21]. We will examine the K10 rovers to guide our design since they have multiple years of concurrent development with PLEXIL [25].

K10 rovers are a line of rovers, designed for cost efficiency and maintainability [3]. They are generally equipped with four wheels, GPS units, digital compass, stereo cameras, and more.

The K10 rovers are designed to be fully automated, with the purpose of assisting during manned missions. They can be used for terrain scouting, soil sampling, or equipment inspection [30, 23].

K10 uses PLEXIL for its task coordination [3]. In recent tests at the Haughton Crater, PLEXIL and the Universal Executive have been able to accurately run programs on the K10 rovers as desired [25].

1.3 Structure of This Thesis

Chapter 2 covers previous work related to PLEXIL design with a look at other languages used for interacting and simulating hardware devices. Chapter 3 presents a design for PLEXIL syntax embedded in Haskell. Chapter 4 will describe a Haskell-based implementation of internally executing PLEXIL semantics, and Chapter 5 will include function descriptions pertaining to simulating hardware devices in the external world. Chapter 6 will discuss final thoughts on this system design with respect to its contribution to the theory of design decisions and future work to further improve upon the PLEXIL language design.

This design of the PLEXIL language is embedded in Haskell [13], a purely functional language. Haskell includes the ability to define new data types using algebraic data types, which is useful when designing domain-specific embedded languages, or DSELs. Haskell functions are statically typed giving us compile-time safety. Also, it employs lazy evaluation to increase the efficiency of execution.

Chapter 2 – Related Work

2.1 Current PLEXIL Simulations

PLEXIL execution is divided into two processes. To clarify the next section, we will call these processes the *executive* and the *simulator*. The executive controls the PLEXIL program flow, while the simulator handles manipulation of the external world (see Figure 2.1). These two processes can communicate through what PLEXIL calls `Commands` and `Lookups`. They must both be running simultaneously to produce meaningful output, possibly even in the same process of a single computer. For this reason, the terms executive and simulator may be used interchangeably.

2.1.1 PLEXIL on SourceForge

The PLEXIL project at NASA has been posted on SourceForge as an open-source project since May of 2008 [21]. They have documented their syntax definitions and released their executive for public usage.

PLEXIL has three standard concrete syntaxes in development at NASA. See Figure 2.2 for examples. The two most commonly used forms are Standard PLEXIL, a standalone concrete style found in some related papers, and PlexiLisp, Lisp-like syntax designed for quick Emacs editing. Both of these syntaxes can be translated

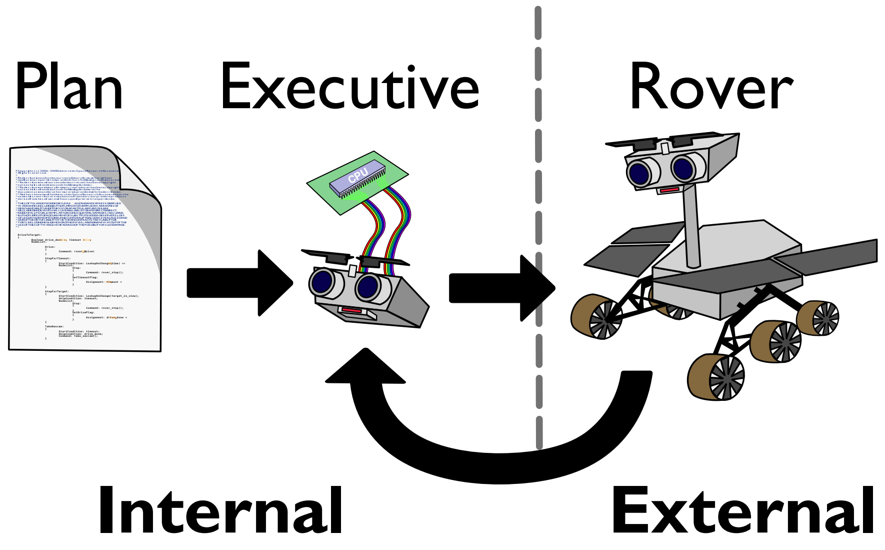


Figure 2.1: Components of the PLEXIL executive system

into an XML form, allowing for simple schema verification of programs. The executive is designed to interpret only the XML syntax, so programs must be translated to XML before running in the simulator.

NASA has a PLEXIL plan executive written in C. To run a simulation, the executive needs a PLEXIL plan in XML format and a *script*.

A PLEXIL script, shown in Figure 2.3, is a file with a sequence of predetermined events. Each event is referenced by name, and it includes the expected result for the simulator.

The problem with this script-based design is that the programmer must anticipate the list of commands sent and sensor values requested from the program during runtime. The user must then generate the corresponding return values

Standard PLEXIL:

```
Drive: {
  PreCondition: ! LookupNow(At("Rock"));
  Command: drive(1.0);
}
```

PlexiLisp:

```
(CommandNode "Drive"
  (Precondition (= false (LookupNow "At" "Rock")))
  (Command "drive" 1.0))
```

Plexil XML:

```
<Node NodeType="Command">
  <NodeId>Drive</NodeId>
  <NodeBody>
    <Command>
      <Name> <StringValue>drive</StringValue> </Name>
      <Arguments> <RealValue>1.0</RealValue> </Arguments>
    </Command>
  </NodeBody>
</Node>
```

Figure 2.2: Short examples of PLEXIL syntaxes [21]

```
initial-state {
    state At ("Rock" : string) = false : bool;
}
script {
    command-success drive (1.0 : real);
    state At ("Rock" : string) = true : bool;
    command-success takeSample ();
}
```

Figure 2.3: A sample script to simulate SafeDrive program (not shown) [21]

by hand to mimic the values returned from the external world. This process is tedious for the programmer and does not help for testing a program in dynamic environments.

Along with documentation on the PLEXIL website, a formal definition of PLEXIL semantics were presented in [5]. The details provided for these operational semantics were the most complete ones available, and were a strong basis for the executive design presented in future chapters. The semantics have since been updated and elaborated in [6].

The semantics describe how PLEXIL plans update over time, along with the interactions between the executive and simulator systems. Some papers describe changes as state diagrams, as in Figure 2.4, and others use small step semantics.

Of particular note is that the formal semantics definitions only describe a portion of the cases for execution. For instance, they do not discuss what happens when the executive sends a unknown command sent to the simulator. Some assumptions will be pointed out in later sections when the semantics do not explicitly

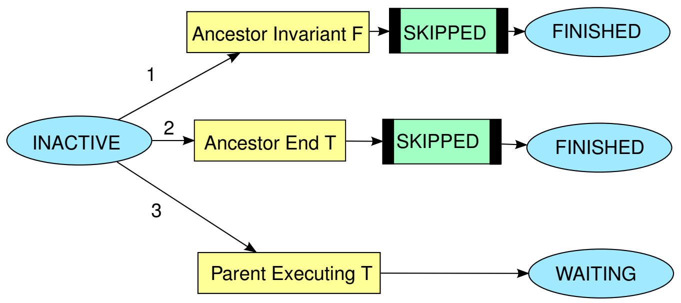


Figure 2.4: Inactive transitions at the lowest (atomic) semantic level [23]

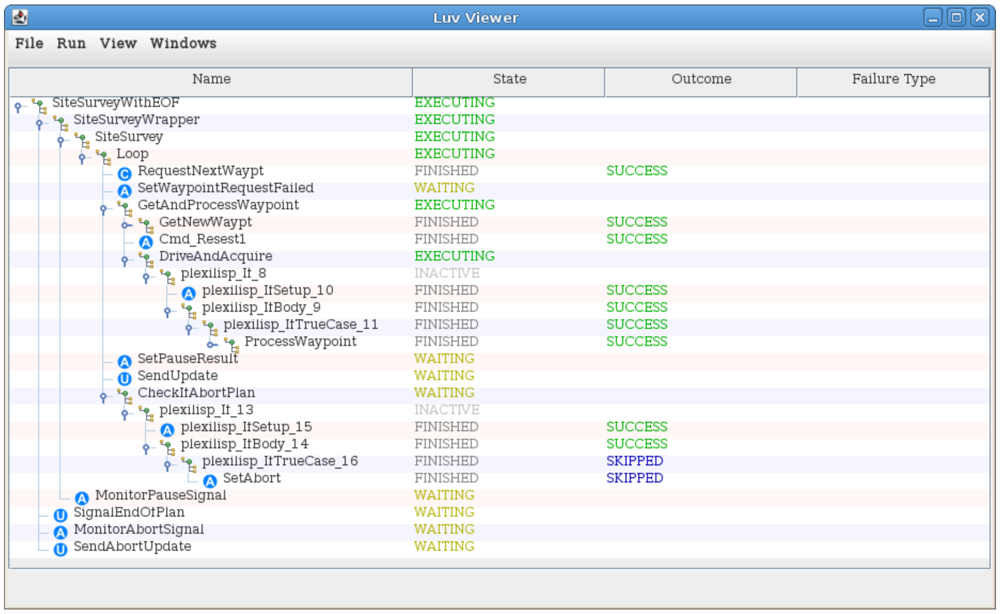
define how to handle all cases of execution.

2.1.2 Lightweight Universal Executive Viewer

The executive at NASA is extended with a graphical user interface known as Lightweight Universal Executive Viewer, or LUV. As shown in Figure 2.5, LUV provides a way to display information about PLEXIL node states during execution. This system allows a user to track the status of a program as it progresses.

With LUV, a user must supply a plan and a script to execute. It uses the executive from the SourceForge website to process the plan.

LUV is still actively updated. It includes features to read intermediate variable values during execution. Also, a user can set breakpoints on nodes or pause execution at any time to allow debugging of a PLEXIL plan in execution.



Name	State	Outcome	Failure Type
SiteSurveyWithEOF	EXECUTING		
SiteSurveyWrapper	EXECUTING		
SiteSurvey	EXECUTING		
Loop	EXECUTING		
RequestNextWaypt	FINISHED	SUCCESS	
SetWaypointRequestFailed	WAITING		
GetAndProcessWaypoint	EXECUTING		
GetNewWaypt	FINISHED	SUCCESS	
Cmd_Resest1	FINISHED	SUCCESS	
DriveAndAcquire	EXECUTING		
plexilisp_If_8	INACTIVE		
plexilisp_IfSetup_10	FINISHED	SUCCESS	
plexilisp_IfBody_9	FINISHED	SUCCESS	
plexilisp_IfTrueCase_11	FINISHED	SUCCESS	
ProcessWaypoint	FINISHED	SUCCESS	
SetPauseResult	WAITING		
SendUpdate	WAITING		
CheckIfAbortPlan	WAITING		
plexilisp_If_13	INACTIVE		
plexilisp_IfSetup_15	FINISHED	SUCCESS	
plexilisp_IfBody_14	FINISHED	SUCCESS	
plexilisp_IfTrueCase_16	FINISHED	SKIPPED	
SetAbort	FINISHED	SKIPPED	
MonitorPauseSignal	WAITING		
SignalEndOfPlan	WAITING		
MonitorAbortSignal	WAITING		
SendAbortUpdate	WAITING		

Figure 2.5: LUV Displaying an Executing PLEXIL Plan

2.2 Functional Reactive Programming

Functional reactive programming [16] is a style of simulation that is based on two key components, behaviors and events. Behaviors are simple functions that define how an object naturally reacts to a given environment, and events are time-sensitive actions that can occur within this environment. A program thus consists of behaviors that naturally occur, such as a robot driving forward, and events that control which behaviors are exhibited, like a robot running into a wall.

2.2.1 Frob/YAMPA

Frob, now combined into the YAMPA project [29], is a language designed for describing robot movements [10, 17]. Frob is a functional reactive DSEL embedded in Haskell. It extends the notion of behaviors and events with an encompassing data type `Task b e`. This should be read as, "A `Task` returns a behavior result type `b` until an event occurs with result type `e`." Tasks have been defined as a monadic type in [18], meaning that Haskell has a built-it syntax to easily compose `Tasks` together.

Tasks can be built up from behaviors and events. Frob includes the following primitives for lifting these components into a `Task`:

```
nullTask :: e -> Task b e
liftB :: Behavior b -> Task b e
```

Frob also defines a set of combinators for designing more complex tasks. A programmer can use `Task` transformations such as these to generate tasks with a wider assortment of actions:

```
withError :: Event Err -> Task a b -> Task a b
withExit :: Event a -> Task b c -> Task b a
withB_ :: Behavior a -> Task b c -> Task b a
```

The first two combinators allow a task to have additional events defining their termination, in either an error or a normal state. The third allows extra behaviors to be attached to an existing task.

```
bug goal =
  do {
    finished <- driveTo goal;
    if (not finished)
      then do {
        goAround goal;
        bug goal
      }
      else nullTask ()
  }
```

Figure 2.6: BUG Algorithm Written in Frob [10]

Frob builds upon Haskell's monadic do-notation to compose tasks together in a style similar to imperative programming. The language introduces the symbols (;) and (|||) to connect tasks either sequentially or in parallel.

In [10], we see an example of a robot maneuvering to a waypoint goal using the BUG algorithm, shown in Figure 2.6. It describes a robot driving towards a goal, traveling around objects in its path. The example shows a concise interaction of behaviors to produce a complex task.

Comparing with PLEXIL, we can see Frob as a language that is designed to take advantage of its embedded language. It makes ample use of monadic computations with the `Task` data type, as well as higher-order functions that are so prevalent in functional languages. PLEXIL was designed as a standalone, lightweight interpreter, so these features will not be as visible in a program definition.

Unlike Frob, where behaviors are an integral part of writing a program, PLEXIL is not a language for defining behaviors. PLEXIL abstracts this information away to a separate simulator of the external world, only to be referenced by command name.

As shown in the example, recursion is allowed in Frob with the Task data type, but there is no built-in iterative looping structure. PLEXIL include this ability through node repeat conditions.

YAMPA extends from Frob through the addition of arrows. Arrows provide an even more generalized technique for describing monadic computations. YAMPA has been used in many domains, reaching much farther than controlling robots in a simulated world, so it will not be discussed in depth in this thesis.

2.2.2 Timber

Timber is another functional reactive programming language embedded in O'Haskell [2]. It is designed as a general-purpose concurrent language, with a program scope ranging over "time-constrained embedded systems. . . to very high-level symbolic manipulation and modeling applications" [4].

Timber programs are designed as a group of objects, waiting for events from the external world. They can react with their defined behaviors, similar to Frob. However, as with most object-oriented languages, Timber provides a way for objects to pass messages. This allows objects to affect each other or to alter the world with some form of output.

The semantics of Timber are broken into three distinct layers, more similar to PLEXIL's approach than Frob. Tasks are broken apart into the functional layer for behavior and the reactive layer for events, which work independently of each other. Also, scheduling is separated into its own layer, unlike Frob's approach of dumping everything into a single Task description.

The representation of the environment in Timber is designed as a set of input and output ports. Any object can read or write data to a given port in the environment. This allows objects to examine the external world without needing direct access to the world's complete internal data representation (akin to information hiding).

2.3 Synchronous Reactive Languages

Synchronous reactive languages are programming languages designed to work on reactive systems. As with functional reactive languages, synchronous reactive languages continually react to their environment [7]. The difference is that synchronous languages have a system for representing time, called *clocks*, within the language. All events use this clock type to add details on when behaviors need to occur. Signal, Esterel, Lustre, and Lucid Synchrone are some languages that fit into this category that we will describe next.

2.3.1 Signal

Signal is a declarative synchronous language [14]. It is used to described interactions of processes over time through signals.

A signal is a sequence of values. Values may or may not be present at certain points in time, known as a *tick*. Any tick with no value is said to hold bottom value, ($\perp$), which represents no defined value.

The set of ticks where a signal holds a value is known as the signal's *clock*. Signals might have equal clocks, but this is not always the case. When two signals have differing clocks, their values show the relative ordering of actions.

The values of a signal can be determined based on constant input streams such as a system clock, input signals described in files, or calculated from values in other signals. An execution of a Signal program will apply some input signal definitions to produce all the signal descriptions in the program. To get a feel for what is going on, let's look at some examples.

Let's start with a simple example of how to manipulate signals, taken from [14].

```
process PLUS1 =
  ( ? integer IN :
    ! integer OUT )
(| OUT := IN + 1 |)
```

The process PLUS1 reads a signal with integer values and writes integer values to another. In this case, the signals are named IN and OUT respectively. Written between the `(|` and `|)` symbols, the body of the process describes, "Whenever the IN signal has a value, the OUT signal will be assigned the value of IN + 1". In this case, the two signals will have the same clock.

Executing PLUS1 on the sample input TEST_IN would produce the following results. Any cases where TEST_IN has a $\perp$ value can be ignored since these ticks will hold no useful data.

```
TEST_IN:  3  7  1  6  0
TEST_OUT: 4  8  2  7  1
```

It is useful to note that, in this example, the two signals show that they have the same clock. This means that a value always appears when another one has a value. There may be ticks with no values between these ticks with values, but according to Signal conventions in [14], we can ignore these ticks with no useful data. Effectively, Signal prefers to show only relevant data, which is the relative signal timing, over the absolute time of outputs.

Signal's model is extended by the concept of event types. Signals with an event type only contain outputs with values $\perp$ or *true*, to denote if the given tick has a value or not.

The Signal documentation covers more advanced examples of how to combine signals [14]. These papers discuss clock operations to combine signal clocks in interesting ways, as well as syntax for conditional signal definitions or lookup of previous data in signals.

The execution of a Signal program could be equated to an abstracted version of PLEXIL. Like PLEXIL, Signal programs read in values from an external system, process the data, and result possible results over time.

PLEXIL takes a different approach by abstracting away sense of time to an external simulator. All execution in the PLEXIL executive is considered to happen in one instant, the equivalent of one Signal click. PLEXIL programs should be able to deterministically execute as much as possible until an update from the simulator is needed. Signal is not bound by this constraint.

Also, the simulator of PLEXIL handles reading the output of the executive and dynamically building up new sensor values. This approach does not seem to exist within Signal.

2.3.2 Esterel

Esterel is a synchronous reactive language, but its syntax follows that of imperative languages [1]. By developing the appropriate abstractions, the designers argue that they can simultaneously achieve concurrency and determinism.

Here is an example of an Esterel program, described in [1].

```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```

This program reads input signals `A`, `B`, and `R`, and sends outputs to signal `O`. The command `emit O` makes the output signal *present*, meaning the signal will hold a value. If this statement is not executed, the signal will instead be *absent*, or empty.

The command `await A` tells the program to wait for the given input signal to be present. By using the (`||`) operator for parallel execution, the program in this case will wait until both `A` and `B` are present. Then, by sequencing the commands with (`;`), an output value will be produced once both inputs are received.

The program includes an abort command, in the form `loop _ each R`. The loop will be executed until the signal of `R` is present. Once this occurs, the loop is immediately exited and then started again.

[1] also gives us the following sample of execution.

```
> A;
Output:
> B;
Output: O
> R;
Output:
> A B;
Output: O
> A B R;
Output:
```

As expected, we receive an output on `O` whenever the inputs `A` and `B` exist. They can be input simultaneously or sequentially. The last case shows that the presence of `R` forces the loop to not be fully executed.

We can see from this example how signal control flow can be generated imperatively. We have access to both sequential and parallel commands, along with the equivalent of while loops not found in the declarative synchronous reactive languages.

The timing constraints of Esterel match closely with conditions in PLEXIL. Both have the power to control execution flow over time in approximately the same level of detail.

Some examples of Esterel acting as a hardware-controlling program can be found at [7]. Of particular interest, these examples include Esterel in control of Lego Mindstorm robots, showing Esterel's ability to interact with hardware.

2.3.3 Lustre and Lucid Synchrone

Lustre [11] is another declarative synchronous language. It has many similar constructs to read and write values to signals on a given clock, similar to Signal. It is considered a data-flow language, where the major focus of a program is to update the program's state based on previous and current signal values.

There are two main differences between Lustre and Signal. First, Lustre enforces an program-wide clock to which all signals must adhere, while Signal leaves this as an option. A signal is allowed to hold no value in a given cycle, but there is no mention of the ability to assign values to a signal unless it is synchronized with the system clock. In comparison, PLEXIL avoids clock-based cycles, allowing programs to change internally as much or as little as necessary at any given point in time. Having Lustre tied to a clock restricts its usability for this thesis.

Also, most examples using Lustre [12] compile output code in the form of deterministic finite automata. The result is a set of program states with transitions describing how to alter the state during execution. The programs show a deterministic execution path for a given set of inputs. However, with PLEXIL we are more interested in tracking the effects on the external world simulation, rather than changes to the internal state.

Lucid Synchrone is an extended version of Lustre [20]. It is embedded in Objective CAML, which gives additional access to a built-in type system allowing more complicated types like records and unions, and higher-order functions that are present in many functional languages. While it provides extra functionality beyond that of Lustre, Lucid Synchrone still has the same purpose of tracking the internal state of an executing program.

For these reasons, we consider Lustre and Lucid Synchrone to be in a different domain than PLEXIL. Thus they will not be considered further in this thesis.

Chapter 3 – PLEXIL Syntax

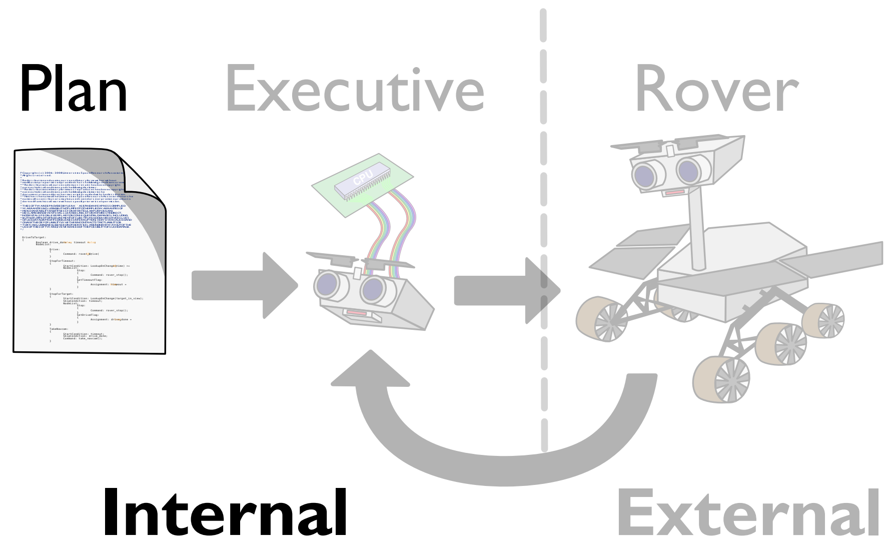


Figure 3.1: PLEXIL plans, defining the program syntax

PLEXIL is a scheduling DSL designed to interact closely with an arbitrarily complex hardware system. This chapter will discuss a PLEXIL implementation embedded in Haskell (see Figure 3.1). Later chapters will discuss how to process data created from this syntactic representation.

NASA provides PLEXIL language implementations in multiple formats, including embedded versions in Lisp and XML, as well as a standalone syntax known as Standard PLEXIL. All of these are a concrete syntaxes, which are discussed in

[21]. However, we can avoid latching onto a specific syntactic representation by using Haskell's data type system for defining syntactic constructs.

3.1 Type Parameterization

The data types discussed in the following sections may include the two type parameters, `s` and `c`. These refer to the `Sensor` and `Command` names. `Commands` are messages sent to an external simulator, and `Sensors` are bindings for sensor values received back from the simulator. By parameterizing these data types, the program will be generalized to work in different systems with their own set of commands and sensor values.

For rovers, we could consider the following data types for commands and sensors. This data type listing is shortened, just to give a feel for what definitions we might expect to see when defining the rest of the syntactic constructs. See Section 5.1 for a more in-depth look into these data types.

```
data Command =
    Drive
  | TakePic

data Sensor =
    PosX
  | PosY
  | WheelStuck
```

By creating data types for these parameter type definitions, we can use Haskell's type system to statically check that the commands and sensors used in a PLEXIL plan are valid for the simulator while allowing these definitions to reside in modules outside of the PLEXIL syntax. In comparison, NASA's current implementation of PLEXIL uses strings for names with no included collection of names for assisting with program validation.

Note that this is one place where PLEXIL widely differs from Frob. PLEXIL command names are just placeholders with no semantic information, leaving the execution of the commands entirely up to the simulator. Frob does not separate the syntax from the semantics. Instead, Frob takes advantage of its embedding in Haskell by passing around first-order functions, as shown in examples from [10].

3.2 Expressions: Use of GADTs

PLEXIL allows users to include expressions within their node definitions. However, trying to set up an efficient yet expressive way to create these expressions in Haskell poses an interesting issue.

One possible solution would involve a single data type `Expr`. Every possible expression allowed in the syntax could be a constructor for this data type, making a single consolidated type. The downside to this method is that users are allowed to interchange these constructors freely to produce meaningless expressions. For instance, we can imagine a case where two values of different types are being combined together, as in the case `Add (BoolConst True) (StrConst "Hello")`.

Another solution may be to create multiple typed data types, like `BoolExpr`, `IntExpr`, etc. Each expression data type holds its own separate combinator definitions, forcing all typed expressions to be evaluatable. The problem with this solution is a bulky and redundant syntax as a result. For instance, both `FloatExpr` and `IntExpr` need their own `Add` combinators, and `BoolExpr` would need a unique combinator for `Equals` for each variable type in the language.

This project opts to use a Generalized Algebraic Data Type [19], or GADT, as an alternative way to generating expressions. A GADT is a single parameterized data type that enforces the use of a different constructor syntax, as seen below. It allows each constructor to designate its own parameterized result type in-line. This lets the efficiency of a single data type blend with the expressiveness of multiple typed data types.

We will now look at the different types of expressions allowed by the following GADT definition.

```
data Expr s a where
  BConst       :: Bool -> Expr s Bool
  IConst       :: Int -> Expr s Int
  FConst       :: Float -> Expr s Float
  SConst       :: String -> Expr s String
  NSConst      :: Status -> Expr s Status
  ...
```

The first GADT constructors wrap constant values into the `Expr` type. Each input type needs its own constructor. The `Status` argument in the `NSConst` represents a node's execution status and will be explained in more depth later.

The following are valid `Expr` usages for constants.

```
BConst True
```

```
IConst 5
FConst 1.7
```

Just wrapping values into expressions is not very exciting. Next we will discuss accessing variables in expressions.

```
type ID = String

data Expr s a where
  ...
  VarBoolEx    :: ID -> Expr s Bool
  VarIntEx     :: ID -> Expr s Int
  VarFloatEx   :: ID -> Expr s Float
  VarStringEx  :: ID -> Expr s String
  VarStatus    :: ID -> Expr s Status
  LookupBool   :: s -> Expr s Bool
  LookupInt    :: s -> Expr s Int
  LookupFloat  :: s -> Expr s Float
  LookupString :: s -> Expr s String
  ...
```

All of the constructors above are used to get values from variables. They each take a name of some sort and produce an `Expr` of the appropriate type. The constructors starting with `Var` are used to read internal variables, while `Lookup` expressions look at sensor data from the external world.

The following are valid variable expressions.

```
VarIntEx "numPictures"
LookupFloat PosX
LookupBool WheelStuck
```

Now we need combinators to design more complex expressions.

```
data Expr s a where
  ...
    --Boolean
  Equal        :: Eq a  => Expr s a -> Expr s a -> Expr s Bool
  LsThan       :: Ord a => Expr s a -> Expr s a -> Expr s Bool
  GrThan       :: Ord a => Expr s a -> Expr s a -> Expr s Bool
  And          :: Expr s Bool -> Expr s Bool -> Expr s Bool
  Or           :: Expr s Bool -> Expr s Bool -> Expr s Bool
  Not          :: Expr s Bool -> Expr s Bool
  IfThen       :: Expr s Bool -> Expr s a -> Expr s a -> Expr s a
    --Numeric
  Add          :: Num a => Expr s a -> Expr s a -> Expr s a
  Sub          :: Num a => Expr s a -> Expr s a -> Expr s a
  Mult         :: Num a => Expr s a -> Expr s a -> Expr s a
```

The combinators can be broken into two types, with either boolean or numeric value outputs. We can look at some examples to see what is happening.

For `And`, we need two `Expr s Bool` arguments, and the result will also be `Expr s Bool` with the two inputs anded together. `IfThen` should check an argument of type `Expr s Bool`. It also takes two `Expr s a`, one if the result is `True` and the other if it is `False`.

The `Add` combinator uses a type class `Num a` to enforce that the type parameter `a` is of some numeric type. If so, the two values can be added together to produce a resulting `Expr s a`.

We define some infix operators to allow shorthand creation of expressions. The symbols should be similar to those in most programming languages.

```
(==:) :: Eq a => Expr s a -> Expr s a -> Expr s Bool
(<:),(>:),(<=:),(>=:)  :: Ord a => Expr s a -> Expr s a -> Expr s Bool
(+:),(*:),(-:) :: Num a => Expr s a -> Expr s a -> Expr s a

(==:) = Equal
(<:)  = LsThan
(>:)  = GrThan
(<=:) a b = LsThan a b ||: Equal a b
(>=:) a b = GrThan a b ||: Equal a b
(&&:) = And
(||:) = Or

(+:) = Add
(-:) = Sub
(*:) = Mult
```

With these infix operators, we can begin to generate larger expressions as follows.

```
VarIntEx "numPictures" >: IConst 10
Not (LookupBool WheelStuck) &&: LookupFloat PosX <=: 5.3
FConst pi *: VarFloatEx "r" *: VarFloatEx "r"
```

GADTs have a drawback that a list of the GADT, in this case `[Expr s a]`, will not allow varying types for parameter `a`. This becomes an issue when defining an environment, a lists of bindings for variables. For instance, the next expression will *not* type check in Haskell:

```
[IConst 0, BConst False, SConst "hello"]
```

We can fix the issue with a union type that wraps all the possible types for the parameter `a` of the type `Expr`. The following definition of `ExprType` is a valid workaround for allowing the creation of typed `Expr` lists.

```
data ExprType s =
    BoolEx   (Expr s Bool)
  | IntEx    (Expr s Int)
  | FloatEx  (Expr s Float)
  | StringEx (Expr s String)
```

The previous example will now type check by wrapping the `Expr` in an `ExprType` constructor. We provide syntactic sugar functions to ease with readability.

```
int    = IntEx    . IConst
bool   = BoolEx   . BConst
float  = FloatEx  . FConst
string = StringEx . SConst
[int 0, bool False, string "hello"]
```

The function `int` wraps an integer value into a data type `Expr s Int` by using `IConst` and subsequently into a `ExprType s` with `IntEx`. Similar functions are included for boolean values, floating-point numbers, and strings. The last line is an updated version of a variable list, which is now type correct in Haskell.

With the definition of expressions, we can add a data type for variable definitions.

```
data Binding s = Var ID (ExprType s)
type Env s = [Binding s]
```

The `Binding` data type allows us to attach an expression to a variable name. We can extend this with the `Env` data type, to store a list of variable definitions.

3.3 Actions

The purpose of PLEXIL as a task scheduling language is to produce a list of tasks, or `Actions`, over time to be executed in an external world simulator. Actions in PLEXIL can be one of the following seven types:

Command: A request to be executed on the external hardware system

Function: A complicated calculation to be calculated outside of PLEXIL

Assignment: Sets a value to an internal variable within a PLEXIL program

List: A collection of sub nodes for grouping similar nodes and allowing multiple actions

Update: Sends output data to the external system, like status information for users

Plan Request: Sends a request to design support asking for a new plan

Empty: No other defined action

The following data type describes the subset of the most commonly used actions in PLEXIL programs. This definition can be extended for a more complete simulation of programs, but this will provide most of the functionality we want for now.

```
data Action s c =
    Command c [ExprType s]
  | Assignment ID (ExprType s)
  | NodeList [Node s c]
```

A `Command` call takes a command name and a list of arguments. For example, one might use the following to tell a rover to drive forward one meter.

```
driveComm = Command Drive (float 1.0)
```

Assignments take a variable name and a new expression to be evaluated and bound to the ID. The variable should be in scope of its current node context, and the type of the expression should match its defined type. These checks will be discussed in more detail in Section 4.2. Here are a few examples of allowed assignments.

```
incPics = Assignment "numPics" (IntEx $ VarIntEx "pictures" +: IConst 1)
isWheelStuck = Assignment "wheelStuck" (BoolEx $ LookupBool WheelStuck)
```

`NodeLists` use Haskell's built in list data type to contain a list of children `Nodes`, which are described next.

3.4 Nodes

The main abstraction in PLEXIL is called a node. A node can have a name, local variables, control conditions, which are optional, and a required action.

Actions define the purpose of a node when it is executing, as described earlier. Each node is limited to containing only a single action.

Some nodes may control a list of children nodes as their action, building up a tree-like structure for the overall program. A program is defined as a single root node with children, which in turn may also be list nodes. Only the leaf nodes will contain actions that could affect either the internal or external state of the executing program.

```
data Node c s = Node ID (Env s) [Condition s] (Action s c)
```

This data type allows for creation of new nodes. The simplest `Node` we can generate would be the following.

```
Node "" [] [] (NodeList [])
```

This is effectively an empty node, with no name and no children nodes. Of course, such a node would not be of much use. We would prefer nodes to store some information.

```
Node "OneMeter" [] []
  (Command Drive [float 1.0])
```

This is a node that, when executing, will call a `Drive` command on the rover.

3.5 Conditions

Nodes will naturally try to execute as soon as possible. The programmer needs some way to limit when a node is allowed to execute. This is why the `Condition s` type was introduced in the previous section.

There are six different node conditions, which can be broken up into two categories. Gate conditions (`Start`, `End`, `RepeatWhile`) define when a node should begin or end execution. Check conditions (`Pre`, `Post`, `Invariant`) describe the settings that must be true for a node to execute properly.

```
type Condition s = (ConditionName, Expr s Bool)
data ConditionName =
     Start | End  | RepeatWhile |
     Pre   | Post | Invariant
     deriving (Show,Eq)
```

Each node can be given at most one of each condition. If a node has two defined `Start` conditions, for example, only the first will be recognized and the rest are ignored. Every condition is tied to a boolean expression that can be as complex as needed.

```
Node "DriveProgram"
  [Var "keepDriving" (bool True)]
  [(RepeatWhile,VarBoolEx "keepDriving" ==: BConst True)]
  (NodeList [ ... ])
```

In this case, the `DriveProgram` node will repeat as long as the boolean variable `keepDriving` remains `True`. If any child node causes the value to change, the node should not try to restart after it has finished executing.

3.6 Node State Information

PLEXIL needs a bit of runtime information available in the syntax. The syntax must include a way to reference the state of a node during execution. This can be broken into two parts, the `Status` and the `Outcome` of the node, which will be described in Section 4.1.1. How nodesupdate their status will be discussed in Section 4.3.1.

```
data Status =
    Inactive  | Waiting | Executing
  | Finishing | Failing | IterationEnded | Finished
  deriving (Show,Eq)

data Outcome = Success | Failure | Skipped | Unknown
  deriving (Show,Eq)
```

By including node state in the syntax, we can now write node conditions like the following:

```
afterDrive = (Start,VarStatus "DriveOneMeter" ==: NSConst Finished)
```

In this case, a node with this start condition will execute only after the node `DriveOneMeter` has finished, forcing serial execution.

Chapter 4 – PLEXIL Semantics

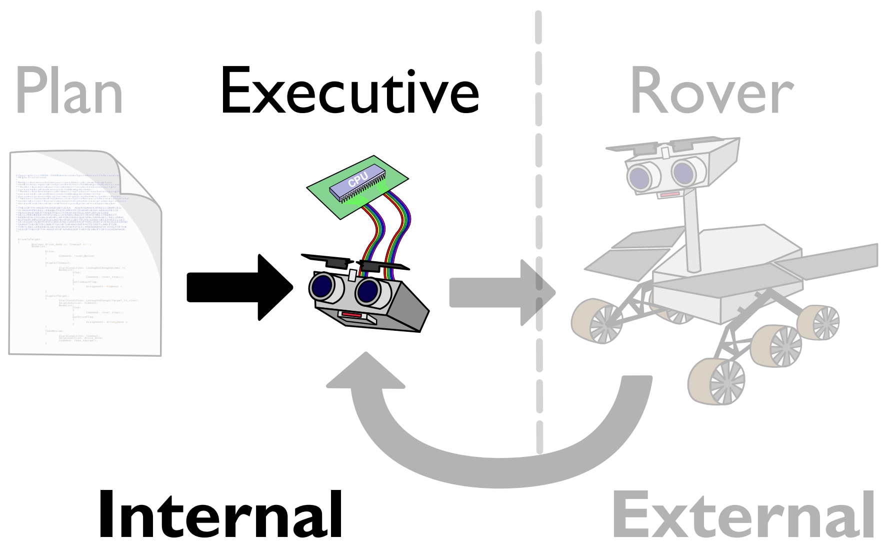


Figure 4.1: PLEXIL executive, defining the semantics for processing plans

The semantics for PLEXIL, defined in [5, 6], discuss how nodes change state over time and briefly touch on how commands are sent to an external system. In order to generate an accurate simulation of a PLEXIL plan in execution, the Haskell system follows the provided definition. This chapter will discuss in detail translating PLEXIL programs into an executable format and the Haskell implementation of PLEXIL's executive semantics (see Figure 4.1).

4.1 Building an Execution Environment

We have discussed how to design a PLEXIL plan in previous sections, but we still need a way to run them. In order to allow program execution, we need a way to store and manipulate program state during runtime. But what else is needed to execute a PLEXIL plan?

As a plan is executing, we need to store three types of data. These are node states, internal variables values, and an external world representation.

4.1.1 Node States

Nodes need to perform their actions at specific times during execution. In order to ensure this, we need a way to keep track of their status. The following data types, first mentioned in Section 3.6, will provide a way to track and control each node's execution.

```
data Status = Inactive  | Waiting  | Executing | Failing
            | Finishing | Finished | IterationEnded

data Outcome = Success | Failure | Skipped | Unknown
```

The `Status` of a node is its execution status. Each node can be waiting to start, currently executing, or completed (successfully or not) with its action. During execution, or possibly before, a node or one of its ancestors may have failed to meet certain conditions resulting in a failure state. The `Outcome` of a node refers to the result of the last attempt at execution for a node.

Section 4.3.1 will discuss the rules for updating the status and outcome of a node during execution. For now, we just need the definition of the data type for defining a semantic node state.

Now we can design a data type for storing a node's runtime state.

```
data State s c = State {nodeId::ID, status::Status, outcome::Outcome,
  start::Expr s Bool, end::Expr s Bool, pre::Expr s Bool,
  post::Expr s Bool, rep::Expr s Bool, inv::Expr s Bool,
  parent::Maybe ID, children::[ID], body::Action s c}
```

The `State` data type contains the information necessary to execute a node. This is stored in a Haskell record type, where each element of the data type is named for easier access.

From a static definition of nodes in PLEXIL, we can obtain information of each node's ID, its parent and children nodes, and the associated action. We also may have a partial listing of its controlling conditions. The undefined conditions need to be given default values, and the status and outcome fields must be initialized.

```
getNodeSem :: Maybe ID -> Node s c -> [State s c]
getNodeSem par (Node i vs cs act) = s : ss act
  where
    s = State i stat out staEx endEx preEx posEx repEx invEx par cids body
    ss (NodeList ns) = concatMap (getNodeSem $ Just i) ns
    ss _ = []
    ...
```

The function `getNodeSem` will let us convert a plan into a list of states by applying it to the root node. We can see the function constructs a new `State` to hold various data on the current node. It also recursively calls this on all children nodes, if they exist. The function's first parameter of type `Maybe ID` is used for recursively assigning the parent node field to each child node. An appropriate initial use of this function would be `getNodeSem Nothing rootNode`.

```
getNodeSem :: Maybe ID -> Node s c -> [State s c]
getNodeSem par (Node i vs cs act)
    ...
    stat  = if isNothing par then Waiting else Inactive
    out   = Unknown
    (cids,body) =
      case act of
           NodeList ns -> (map (\(Node i _ _ _) -> i) ns, NodeList [])
           a -> ([], a)
    ...
```

PLEXIL sets the root node, defined as the only node in the plan with no parent, to a status of `Waiting`, while all other nodes have an initial `Inactive` status. All outcomes start out as `Unknown` since they have yet to execute. Child nodes are stored as a list of their names, which should be unique according to PLEXIL programming guidelines [21]. Since we are converting all `Nodes` to `States`, we can throw away the `NodeList` references to children nodes to simplify the result.

```
getNodeSem :: Maybe ID -> Node s c -> [State s c]
getNodeSem par (Node i vs cs act)
    ...
    getEx def = fromMaybe def . flip lookup cs
    preEx = getEx (BConst True) Pre
    posEx = getEx (BConst True) Post
    repEx = getEx (BConst False) RepeatWhile
```

```
invEx = getEx (BConst True) Invariant
staEx = getEx (BConst True) Start
endEx =
  case act of
       NodeList ns
         -> maybe nodeListEnd (\ex -> Or ex nodeListEnd) $
              lookup End cs
       Assignment _ _
         -> getEx (BConst True) End
       _ -> getEx (BConst False) End
-- End when all children are finished
nodeListEnd = (foldr1 (&&:) .
               map (\i -> VarStatus i ==: cFinished)) cids
```

For adding conditions to a `State`, we need to scan through the node's list of defined conditions. For each condition, we will either take the first defined condition through the `flip lookup cs` function. Otherwise, we will use a predefined default. In [21], we get the following default conditions for five of the six conditions, which are used for all node types.

Condition	Default Value (all nodes)
Pre	True
Post	True
Repeat	False
Invariant	True
Start	True

Each default expression is passed to `getEx` to provide an alternative in case an explicit definition cannot be found.

The default `End` condition is a bit more complex. Each node type has a unique condition description, as seen below.

Node Type	End Condition Default
NodeList	All Children Finished
Command	Cmd Handle Received
Assignment	Call Completed

`NodeLists` use `nodeListEnd`, which creates the expression `(VarStatus` i_1 `==: cFinished) &&: ... &&: (VarStatus` i_n `==: cFinished)` for children with IDs $[i_1..i_n]$. Assignments should always occur right away, so they should attempt to finish as soon as possible. This leads to the default condition `True`. Command nodes should wait to receive a message from the simulator. We will set the command node's end condition to `False`, but provide a message handler for ending command nodes in a later section.

4.1.2 Variable Bindings

Variables are defined in nodes, but we will separate them into their own separate list for ease of use. Each variable will be bound to its current value. Recall the `Binding` data type we used for describing variables in PLEXIL syntax.

```
data Binding s = Var ID (ExprType s)
type Env s = [Binding s]
```

Given the root node for a plan, we can extract all the variables from the nodes into their own separate list. This will allow us to quickly read and update variable values.

```
getVarsSem :: Node s c -> Env s
getVarsSem (Node _ vs _ act) =  vs ++ vs'
  where vs' = case act of
                   NodeList ns -> concatMap getVarsSem ns
                   _ -> []
```

Note that by separating variables from their nodes, there is no way to check that variables are accessed by nodes with appropriate scope. It is currently left to the program designer to verify that a plan has no assignments that are out of scope, but the system could be updated in the future to statically check for this.

4.1.3 External World Representation

We will discuss designs for simulators in a later section, but for now we need a data type to store relevant pieces of an external world. We begin with some type definitions.

```
type BoundAction s c = (ID,Action s c)
type BoundReturn s = (ID,ExprType s)
type CommandFunction s =
       TimeSpan -> [ExprType s] ->
           [Sensor s] -> ([Sensor s], Maybe (ExprType s))
type TimeSpan = Float
```

`BoundActions` are used to hold commands with a reference to the ID of the node that is executing it. Similarly, a `BoundReturn` is a node ID tied to a result value from a completed command. A `CommandFunction` is a function designed in the simulator for updating sensors. With these types, we can now define a representation of the external world for the executive.

```
data ExtWorld s c = World {
  sensors::[(s,ExprType s)], comms::[BoundAction s c],
  commrets::[BoundReturn s], commlist::[(c,CommandFunction s)]}
```

Within a `World` record type, we store four items to describe the workings of a simulator. `Sensors` is a list of current sensor values. The `comms` list holds all active commands, while `commrets` is a list of return values from completed commands. We also include `commlist`, a list of available functions similar to an interface.

Note that both the `sensors` and `commlist` lists use the parameterized types `s` and `c` in their type definitions, respectively. This allows the command and sensor names to be defined outside of the executive module, allowing for an executive that is independent of any particular simulation system.

During execution, all of the external data is available to the executive, but it should only have limited write access. We will see cases later where the executive must manipulate the command returns list, but other manipulation should be left to simulator modules.

4.1.4 PlexilEnv: The PLEXIL Environment Data Type

With all of the separate components defined, we can now create an execution environment for PLEXIL. We will collect the data types defined in the previous sections into a `PlexilEnv` data type.

```
data PlexilEnv s c = PlexilEnv {
     internal::Env s,
     states  ::[State s c],
     external::ExtWorld s c,
     commands::[BoundAction s c] }
```

A `PlexilEnv` is the current state of a PLEXIL plan in execution. The first three pieces of this record type are the internal bindings for variables, the states of all nodes in the plan, and a collection of data for the external world. The last element, `commands`, is the list of commands sent from nodes, but have yet to be received by the executive. The command list could be separated from the environment for a more succinct definition, but it is easier to design the semantic relations described later when all the data is kept close together.

```
plexilEnv :: Node s c -> ExtWorld s c -> PlexilEnv s c
plexilEnv n e = PlexilEnv vars nodes e []
  where
    vars = getVarsSem n
    nodes = getNodeSem Nothing n
```

This function allows us to convert a plan into an environment for execution. It takes a plan, referenced by its root node, and a predefined external world. It uses the previously defined functions `getNodeSem` and `getVarsSem` for converting the data into a `PlexilEnv` type. With this new structure, we can begin to define semantics for updating the environment.

4.2 Evaluating Expressions

With a runtime environment, we can now begin discussing semantic rules. We will begin with an expression evaluator. PLEXIL expressions were defined earlier in section 3.2 with GADTs, and they get used often throughout execution.

```
eval :: (Show s, Eq s) => Expr s a -> PlexilEnv s c -> a
```

This function takes a expression of type `Expr s a`. Expressions need access to local variables, external sensors, and node states, which explains the need to include a `PlexilEnv` as a parameter. Once evaluated, the result should be of the resulting type `a`, which currently may be `Bool`, `Int`, `Float`, or `String`. Some expressions may also return a node's status in an intermediate step, but the `Status` value should not be returned as a final value.

Let us consider a few examples of possible expressions we will evaluate.

```
eval ex env =
  case ex of
       BConst val  -> val
       IConst val  -> val
       FConst val  -> val
       SConst val  -> val
       NSConst val -> val
       ...
```

These are constant values wrapped in an `Expr` constructor. Evaluation here is just unwrapping the value inside. For example, `FConst 3.5` should become `3.5` and `NSConst Finished` will evaluate to `Finished`.

Along with constant values, we also want to evaluate variables. This requires reading values from the `PlexilEnv`. The following helper function allows us to scan for expressions that match variable or sensor definitions by name and type.

```
findExpr :: Eq i =>
            [(i,ExprType s)] -> i -> ExprType s -> Maybe (ExprType s)
findExpr es i e' = listToMaybe [e | (i',e) <- es, i == i', e == e']
```

This function first looks for variables that match both the id and the type of expression. The call to `listToMaybe` will return the first result, or `Nothing` if none exists.

```
eval ex env =
  case ex of
       ...
       LookupBool   s -> eval (lsub s unwrapB) env
       LookupInt    s -> eval (lsub s unwrapI) env
       LookupFloat  s -> eval (lsub s unwrapF) env
       LookupString s -> eval (lsub s unwrapS) env
       ...
  where
    lsub s (un, def) = maybe (lookupError s) un $
       findExpr (sensors $ external env) s def
    lookupError i = error $ "Lookup " ++ show i ++ " does not exist"
    unwrapB = ((\(BoolEx   ex) -> ex), bool False)
    unwrapI = ((\(IntEx    ex) -> ex), int 0)
    unwrapF = ((\(FloatEx  ex) -> ex), float 0.0)
    unwrapS = ((\(StringEx ex) -> ex), string "")
```

To evaluate lookups, we can apply `findExpr` to our sensor list. The result will be in the form of an `ExprType` union type, so we need the unwrapping functions to extract the expression. If the requested sensor cannot be found in the external world definition, an error is thrown by `lookupError`. Otherwise, the result is an `Expr` type, which must be recursively evaluated.

Internal variables are evaluated similarly to lookups. The difference is that they instead scan over the list of internal variables, `states env`.

```
eval ex env =
  case ex of
       ...
       VarBoolEx    i -> eval (vsub i unwrapB) env
       VarIntEx     i -> eval (vsub i unwrapI) env
       VarFloatEx   i -> eval (vsub i unwrapF) env
       VarStringEx  i -> eval (vsub i unwrapS) env
       VarStatus    i -> stat i
       ...
  where
    vsub i (un, def) = maybe (varError i) un $
       findExpr [(i',e') | Var i' e' <- internal env] i def
    varError i = error $ "Var " ++ i ++ " does not exist"
    stat i = maybe (statError i) status (lookupByID nodeId (states env) i)
    statError i = error $ "Node " ++ i ++ " does not exist"

lookupByID :: (a -> ID) -> [a] -> ID -> Maybe a
lookupByID f as i = listToMaybe [a | a <- as, f a == i]
```

Internal variables can also include accessing node statuses. The `VarStatus` case instead needs to look for a node with a matching name and return its current status.

```
eval ex env =
  case ex of
```

```
...
Add    l r   -> eval l env + eval r env
Equal  l r   -> eval l env == eval r env
Not    l     -> not $ eval l env
IfThen i t e -> if (eval i env) then (eval t env) else (eval e env)
...
```

The rest of the cases are expression combinators, combining one or more subexpressions. A few examples are given above. They each recursively evaluate their components, then combine the results with the appropriate function.

Evaluating expressions is one of the lowest levels of semantics available in PLEXIL. We will use this ability to read variables to produce more complex effects in the PLEXIL environment.

4.3 Five Levels of Semantics

4.3.1 Level 1: Atomic Relation

The atomic relation controls one node through a single status update. Each node has the potential to update its status based on its current status, ancestors' statuses, guard conditions, and action type.

We can see three potential updates occur in any particular atomic relation. The first two are changes to a node's status or outcome. Also, command nodes may have commands ready to run, and assignment nodes may want to update internal variables. Thus, given a node state and the current `PlexilEnv`, we produce a function with the following type.

```
atomic :: (Show s, Eq s) =>
  PlexilEnv s c -> State s c -> (State s c,[Binding s],[BoundAction s c])
```

The rules defining the atomic relation transitions were originally defined in [23] as a group of state diagrams. These have since been updated to a more formalized set of small-step inference rules in [6].

In both cases, we can see that there are over 30 possible transitions a node could take and multiple conditions to track to determine which is most appropriate. We will begin by defining how to evaluate some necessary conditions, then show examples of how transition rules can be converted into semantics embedded in Haskell.

For any node, we may need to check its status or evaluate one of its conditions. We include the following subfunctions to handle these cases:

```
atomic env s = ...
  where
    is stat    = status s == stat
    evalT cond = evalnT cond s
    evalF      = not . evalT
    evalnT c x = eval (c x) env
    evalnF c   = not . evalnT c
    ...
```

The first function will compare the current node's status. Calling `is Executing`, for example, checks if the the node status is `Executing`. The other functions evaluate conditions. So, the call `evalF inv` will let us know if the invariant condition of the node evaluates to `False`. The functions `evalnT` and `evalnF` evaluate explicit node conditions in case we want to view other nodes' conditions, which we will be using next.

We may need to check if a node's ancestors are, for any reason, trying to alter execution. The following code lets us access their states to see if there is any important information.

```
getAncestors ns n =
  case maybe Nothing (lookupByID nodeId ns) (parent n) of
       Just parProc -> parProc : getAncestors ns parProc
       _ -> []

atomic env s = ...
  where
    ...
    ancestors = getAncestors (states env) s
    ancestorInvF = any id $ map (evalnF inv) ancestors
    ancestorEndT = any id $ map (evalnT end) ancestors

    parentStatus = maybe Executing status $ listToMaybe ancestors
    parentExecuting = parentStatus == Executing
    parentWaiting = parentStatus == Waiting
    ...
```

The function `getAncestors` builds a recursive list of parent nodes. In some cases we need to know if any ancestor passes an end condition or fails an invariant condition. We can view these cases by using, respectively, `ancestorInvF` and `ancestorEndT`.

The parent process's status may also trigger a change in the current node's status. We can get the parent node state by using `listToMaybe ancestors`, which safely extracts the first element of the ancestor list into a `Maybe` data type. We can then extract the parent status with the `maybe` function for comparison.

Finally, before we begin to define transitions, we add a shorthand way to update node states. Each update will set either a node status or node outcome.

```
atomic env s = ...
  where
    ...
    sInac, sWait, sExec, sFing :: State s c -> State s c
    sIter, sFail, sFind        :: State s c -> State s c
    sInac s = s {status = Inactive}
    sWait s = s {status = Waiting}
    ...
    oSucc, oFail, oSkip, oUnkn :: State s c -> State s c
    oSucc s = s {outcome = Success}
    ...
```

Each available status and outcome has its own shorthand notation. By leaving these updates as functions, we can compose them together to easily make multiple changes.

We can now begin to implement semantics for atomic relations. First, we need to determine the appropriate function for updating a node. We can use Haskell's guard syntax for defining transition conditions and their respective results.

```
atomic env s = (f s, vs, c)
  where
    f | is Waiting && (ancestorInvF || ancestorEndT) = sFind . oSkip
      | is Waiting && evalT start && evalF pre       = sIter . oFail
      | is Waiting && evalT start && evalT pre       = sExec
      ...
```

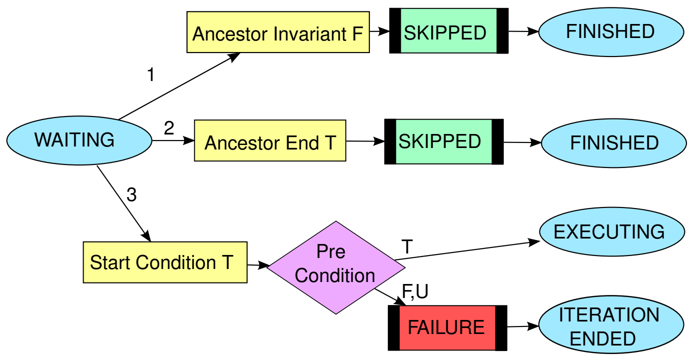


Figure 4.2: Waiting transitions at the atomic level [23]

These three cases match the transitions in 4.2, where a node is in `Waiting` status. The first definition of `f`, which executes if `ancestorInvF` or `ancestorEndT` holds true, will produce the update function that sets a node outcome to `Skipped`, then its status to `Finished`.

The transitions are ordered in the diagram. Guards in Haskell allow us to obey this ordering since the first guard it matches will be executed. So, as long as each guarded statement is written in order, a Haskell interpreter will choose the appropriate transition to follow. Also, this example shows that multiple transitions with the same result can be combined to help shorten the number of checks required.

For executing command nodes, the formal semantics do not clearly define the default end condition. The online documentation describes it as when a “command handle [is] received [21]”. The formal semantics definitions do not otherwise discuss this detail. The solution presented here is to scan for command returns sent from the simulator.

```
atomic env s = (f s, vs, c)
  where
    f ...
      | is Executing
      = case body s of
        Command _ _
          | hasCommRet && evalT post -> sIter . oSucc
          | hasCommRet && evalF post -> sIter . oFail
          | ...
    ...
    commRet = lookup (nodeId s) . commrets
    hasCommRet = isJust $ commRet (external env)
```

This solution only checks for the existence of a return value. Since it is not clearly defined how command returns should be handled, this is left as an open problem.

It should be noted also that some transitions want to know whether a failure outcome was caused by a parent or not. For simplicity, all failures are considered the same. The `Outcome` data type could be updated later to allow for a `ParentFail` constructor should test cases need this feature.

The final guard of `f` is as follows.

```
atomic env s = (f s, vs, c)
  where
    f ... | otherwise = id
```

Not every node state needs to be updated. Some nodes may be have an `Inactive` status and are waiting for their parents to begin executing. Others, like command nodes, may still be `Executing` but have yet to receive a return value. We can apply the identity function to these node states, telling them to return unchanged until some other time.

Along with updating the node state, we also need to check for variable updates and commands to return. We can use the following test to find variable updates.

```
atomic env s = (f s, vs, c)
  where
    ...
    vs = case body s of
      Assignment i exType
        | is Executing && evalT end && evalT post
                    -> [Var i $ evalExprType exType env]
        | otherwise -> []
      _ -> []
    ...

evalExprType (BoolEx   x) e = bool   (eval x e)
evalExprType (IntEx    x) e = int    (eval x e)
evalExprType (FloatEx  x) e = float  (eval x e)
evalExprType (StringEx x) e = string (eval x e)
```

Updates occur when an `Assignment` node leaves `Executing` status. The assignment expression must be evaluated outside of its `ExprType` wrapper, and then rewrapped into a new variable definition.

It is important to point out that all nodes should be executing the atomic relation in parallel. We cannot immediately update the internal list of variables now, in case the current node changes a value before another node can access it. So, the new definition must be returned up one level to execute simultaneously.

Finally, before moving on to the next relation, we need to gather new commands to execute. These can only come from command nodes.

```
atomic env s = (f s, vs, c)
  where
    ...
    c = case body s of
      c@(Command i es)
        | is Waiting && evalT start && evalT pre
                    -> [(nodeId s, c)]
        | otherwise -> []
      _ -> []
    ...
```

To find a new command, we look for a node with a `Command` action. Then if the node has the same conditions that would cause it transition to `Executing`, we can return its command. In this function, we bind the command to the ID of the node, so we can track which node sent the command request.

4.3.2 Level 2: Micro Relation

At the micro relation level, we want to collect all the updates made for each node in the atomic relation and reconstruct a new `PlexilEnv`. Updates can be in the form of nodes with new status, changed variable values, or commands to send to the external world.

It is possible that there will be no updates in a given micro relation cycle. We return a `Bool` flag that tells whether or not any updates were made. This will become important when designing the next relation in the following section.

The final consideration before defining the micro relation is the transactional nature of the relation. This means that instead of creating an implicit ordering of effects from the nodes, all effects should occur simultaneously to produce an updated state. Such a behavior can be obtained by copying the original state and updating the necessary parts, all while checking that no two nodes produce conflicting effects.

With these constraints in place, we can define the micro relation as follows:

```
micro :: (Show s, Eq s) => PlexilEnv s c -> (PlexilEnv s c, Bool)
micro env = (e', changed)
  where
    ns = states env
    (ns',vs',cs') = getAtomicUpdates env ns
    changed = nodesChanged ns' ns || nonEmpty vs' || nonEmpty cs'
    e' = PlexilEnv {internal = microVars vs' (internal env),
                    states   = ns',
                    external = (external env) {commrets = [], commmsgs = []},
                    commands = commands env ++ cs'}

getAtomicUpdates e ns = (ns', concat vs, concat cs)
  where (ns',vs,cs) = (unzip3 . map (atomic e)) ns
```

The micro relation begins with applying the atomic relation to the current states. This call produces three lists containing updated node states, variables, and commands. We insert each of these lists back into a `PlexilEnv` constructor.

To construct the new environment, we need to combine the results just obtained. The atomic relation returns an updated state for every node, so the original list of states can be replaced with `ns'`. New commands are appended to the list of commands that have yet to be sent to an external simulator. Variables are a bit more tricky and will be discussed later in this section. Notice that data from the external world is also updated, as return values and messages from externally executing commands should be consumed through node status updates in the atomic relation.

With the new environment `e'`, the result also contains the `changed` flag. It is marked as true whenever a node or variable is updated or a command is passed up from the atomic relation. We can complete this definition with the following helper function.

```
nodesChanged :: [State s c] -> [State s c] -> Bool
nodesChanged ns nsOld
  | nonEmpty nonmatches = error $ "Couldn't find State(s) "
                        ++ (show . map nodeId) nonmatches
  | otherwise = nonEmpty changed
  where
    nonmatches = [ n | n <- ns, all (diffId n) nsOld]
    changed = [ n | n <- ns, all (diffNode n) nsOld]
    diffId n n' = nodeId n /= nodeId n'
    diffNode n n' = diffId n n' || status n  /= status n'
                                || outcome n /= outcome n'
```

This definition assumes that, by some chance, new nodes with no previous definition may get added. We can assume that each node has a unique ID, which forces ID recognition to determine that no nodes are out of place. Misplaced nodes are treated as an error, as seen in the first guarded case of the `nodesChanged` function.

If copies of all nodes are present in both lists, we can determine if a node has been updated by comparing the status and outcome of nodes. The `changed` list marks all nodes in the updated list that do not match any previous node definition. If this list contains any elements, a node has been changed.

To finish up the micro relation, we need a way to handle all special cases of variable updates. We propose the following solution to generate an updated list of variable bindings.

```
microVars :: (Show s,Eq s) => [Binding s] -> [Binding s] -> [Binding s]
microVars vs vsOld
  | nonEmpty nonmatches = error $ "Attempted to assign unknown vars "
                                ++ show nonmatches
  | nonEmpty dups = error $ "Attempted the following simultaneous"
                          ++ " illegal assignments: " ++ show dups
  | otherwise  = vs ++ unchanged
  where
    nonmatches = [ v | v <- vs, notElem v vsOld]
    dups = [ v | v <- vs, length (filter (==v) vs) > 1]
    unchanged = [ v | v <- vsOld, notElem v vs]
```

The third case is the simplest. It says that, in the general case, we want to combine the updated list of bindings along with the original bindings not just updated. However, we need to take account for cases where unusual update behaviors are requested.

First, for assignment statements contained within nodes, there has been no check thus far to confirm that the assigned variable is within scope of the node. Should a program like the following one attempt to write to an unknown variable, the interpreter will produce an error.

```
Node "BadAssign"
  [Var "i" $ int 0]
  []
  (Assignment "xyz" $ bool False)
```

The `microVars` function can handle cases of simultaneous updates to a single variable. PLEXIL throws an error when two assignment actions attempt to change the same variable in a micro relation, as in the following example.

```
Node "BadXXAssigns"
  [Var "x" $ int 0]
  []
  (NodeList [
    Node "X1" [] [] (Assignment "x" $ int 1),
    Node "X2" [] [] (Assignment "x" $ int 3)
  ])
```

The `dups` list holds all new bindings that match at least one other binding. In this case, both assignments to `x` will be added to this list, resulting in an error. However, we can still make simultaneous transaction-style updates as expected in cases like this example taken from [5].

```
Node "XYSwap"
  [Var "x" $ int 1, Var "y" $ int 2]
  []
  (NodeList [
    Node "XY" [] [] (Assignment "x" $ VarIntEx "y"),
    Node "YX" [] [] (Assignment "y" $ VarIntEx "x")
  ])
```

With this example, both assignments should execute simultaneously. When they do, they values of `x` and `y` will be appropriately swapped.

4.3.3 Level 3: Quiescence Relation

PLEXIL defines the quiescence relation as the repeated application of the micro relation on the internal state until reaching a normal form. A normal form is obtained when executing the micro relation no longer produces an effect, such as nodes changing status or variables getting updated.

Executing until normal form is referred to as *running to completion* [6]. Effects in the external world cannot be determined programmatically, thus are a source of non-determinism in PLEXIL programs. In order to minimize the program variability, the executive attempts to maximize the change in internal structure for each given state of the external world.

Thus, we can define the quiescence relation in the following manner:

```
quiescence :: (Show s, Eq s) => PlexilEnv s c -> PlexilEnv s c
quiescence e = fst $ until (not . snd) (micro . fst) (e,True)
```

This says that, given an initial `PlexilEnv e`, continually update `e` using the micro relation until it produces no changes. Recall that the micro relation produces a data tuple, an updated `PlexilEnv` with a `Bool` flag noting if any changes occurred. The definition above uses the initial state `(e,True)`, executes the micro relation on `e` to produce the intermediate state `(e',b')`, and then recursively continues executing micro steps on `e'` unless `b'==False`.

It is important to note that with this definition of quiescence, we are not guaranteed termination of all PLEXIL programs. We could imagine the following program:

```
Node "Endless"
  [Var "b" $ bool False]
  [RepeatWhile $ BConst True]
  (Assign "b" $ bool True)
```

This node does not need to communicate with the external world. So, the node will continually update its status with its repeat condition always set to `True`. This is a simple case where a program will never break out of the quiescence loop.

A few alternatives have been proposed to work around the issue of infinite looping [6]. These solutions limit the number of times the micro relation is executed in a single cycle of quiescence. This could range from a single micro relation to as many as necessary such that every node changes at most from an initial status back to the same status (i.e. `Waiting` to `Waiting`). Any of these modified versions of quiescence could be easily implemented with minor adjustments to the structure of the described interpreter.

4.3.4 Level 4: Macro Relation

The macro relation is the first semantic level that is affected by the external world. It controls sending commands to the simulator and updating the resulting representation of the external world for use in the lower levels of semantics.

We can begin with a function for sending commands to the external world.

```
addCommands :: Eq c => [BoundAction s c] -> ExtWorld s c -> ExtWorld s c
addCommands as w = w {comms = comms w ++ cs'}
  where cs' = [c | c@(_,Command x _) <- as, hasKey x $ commlist w]
        hasKey k = isJust . lookup k
```

The addCommands function appends a new list of commands, stored in a BoundAction type, to the external world's list of running commands. When setting the value of cs', we can use a list comprehension with a filter to only add commands that are defined in the world's commlist, or list of known commands. This will let us simplify the following functions.

Next, we need a way to receive updated sensor values after letting the simulator run for some amount of time. We can solve this by defining the following function.

```
execComms :: Eq c => TimeSpan -> ExtWorld s c -> ExtWorld s c
execComms t w | null $ comms w = w
              | otherwise      = w' {comms = comms'}
  where
    (c@(node,Command cid ps) : cs) = comms w
    (Just f) = lookup cid $ commlist w
    (ss',ret) = f t ps $ sensors w
    w' = execComms t w {comms = cs, sensors = ss', commrets = rs'}
    rs' = maybe (commrets w) (\r -> ((node,r) : commrets w)) ret
    comms' = maybe (c:comms w') (\_ -> comms w') ret
```

Thi execComms function begins by extracting the first element from the world's list of commands. It converts this to a CommandFunction f, defined by the simulator. This function is executed to give intermediate sensor updates. The function execComms is then recursively called on the remainder of the commands until the list is empty. Each command will be added back to the list of commands unless it had a return value. In this case, the return value is bound by the node state and added to the list of return values.

Up to this point, we have a way to execute the quiescence relation for updating internal state and functions to control external simulator actions. Now we need to tie these two parts together to complete the macro relation.

To have a simulator that functions in real-time, we need access to the system clock. Haskell provides us with a `ClockTime` data type with the function `getStateTime :: IO ClockTime` to read the computer's current clock time. In order to show the progress of an executing plan over time, we will introduce the following state transformation monad.

```
type PlexilMonad = StateT ClockTime IO
```

The `StateT` monad allows us to store a value of type `ClockTime`, which we can update by reaching down to the `IO` monad. We could thus read the type `PlexilMonad PlexilEnv` as the status of the `PlexilEnv` at a given `ClockTime`.

```
getSecDiff :: ClockTime -> ClockTime -> Float
getSecDiff c d = f
  where
    t = diffClockTimes c d
    f = ((/ 10^12) . fromInteger . tdPicosec) t + (realToFrac . tdSec) t
```

The helper function `getSecDiff` compares two `ClockTime` values. We use `diffClockTimes` to get a comparison, and then extract the difference in seconds.

We will use the following function to control the application of a single entire iteration of the macro relation.

```
macro :: (Show s, Eq s, Eq c) =>
          PlexilEnv s c -> PlexilMonad (PlexilEnv s c)
```

The `macro` function is best explained when split into its parts. Monadic computations will execute commands iteratively, so we can follow the order of execution.

```
macro e = do
  e' <- return $ quiescence e
  liftIO $ threadDelay 10000 -- 1/100 second
  ...
```

First, the `quiescence` relation updates our internal state. The result is stored in `e'` for later use.

The next line is optional. `threadDelay 10000` forces the executive to pause for $\frac{1}{100}$ second, which is an external system command that must be run in the `IO` monad context. Without this line, execution traces generally grow quickly. This way, there will be at most 100 iterations of the macro relation each second.

```
macro e = do
  ...
  cthen <- get
  cnow <- liftIO getClockTime
  put cnow
  ...
```

Here, we are accessing the `ClockTime` stored within the state of the `PlexilMonad`. We update the old stored time `cthen` with the current value `cnow` obtained by calling `getClockTime`.

```
macro e = do
  ...
  ext' <- (return . execComms (getSecDiff cnow cthen) .
           addCommands (commands e')) $ external e'
  ...
```

Given the current external world, we add commands sent from the quiescence relation. Then, we simulate command execution over the span of time that passed since last execution. This gives us a new copy of the external world to use in the next iteration.

```
macro e = do
  (return . updateExt ext') e'
    where updateExt e p = p {external = e, commands = []}
```

We finish the macro relation by updating the `PlexilEnv` external world with the newly simulated version. We can also remove all commands passed up from quiescence so they will not be run again in later iterations.

4.3.5 Level 5: Execution Relation

Execution of a PLEXIL plan is defined as the repeated application of the macro relation until all nodes have finished execution. We will begin simulating this execution with the `executeN` function.

```
executeN :: (Show s, Eq s, Eq c) =>
             Int -> PlexilEnv s c -> PlexilMonad [PlexilEnv s c]
executeN i e
  | i==0 || allNodesFinished = return [e]
  | otherwise                = liftM (e:) $ macro e >>= executeN (i-1)
  where
    allNodesFinished = all (\x -> x == Inactive || x == Finished)
                       (map status $ states e)
```

Each iteration of `executeN` will execute one step of the macro relation and add the resulting state to a recursive list of states. This allows us to gather an entire trace of an executing plan.

There are two base cases given in the first guarded statement. First we include an `Int` parameter to the function. This controls the maximum number of iterations allowed to ensure we only run a finite number of macro steps in the simulation. Second we can stop execution when all nodes are have either `Inactive` or `Finished` status. In this case, no conditions can force any node to reactivate, so we can use a form of short-circuit evaluation to exit early.

The next function `trace` handles wrapping and unwrapping `executeN` in the `PlexilMonad`.

```
trace :: (Show s, Eq s, Eq c) =>
          PlexilEnv s c -> Int -> IO [PlexilEnv s c]
trace e i = getClockTime >>= evalStateT (executeN (max i 0) e)
```

To finish getting a trace of a plan execution, we need to attach our initial state to the computer's current time. The function `trace` will get the initial `ClockTime` from the system, execute a given number of iterations, and extract the result from the `PlexilMonad` state through `evalStateT`.

We can then opt to read portions of the trace with functions like the following.

```
execute  :: (Show s, Eq s, Eq c) =>
             PlexilEnv s c -> Int -> IO (PlexilEnv s c)
execute e = liftM last . trace e

traceExt :: (Show s, Eq s, Eq c) =>
```

```
             PlexilEnv s c -> Int -> IO [ExtWorld s c]
traceExt e = liftM (map external) . trace e
```

The first function, `execute`, will return only the final element of the trace, instead of the list of intermediate states. The other function, `traceExt`, filters the trace environments to only their external world data.

Chapter 5 – The External World

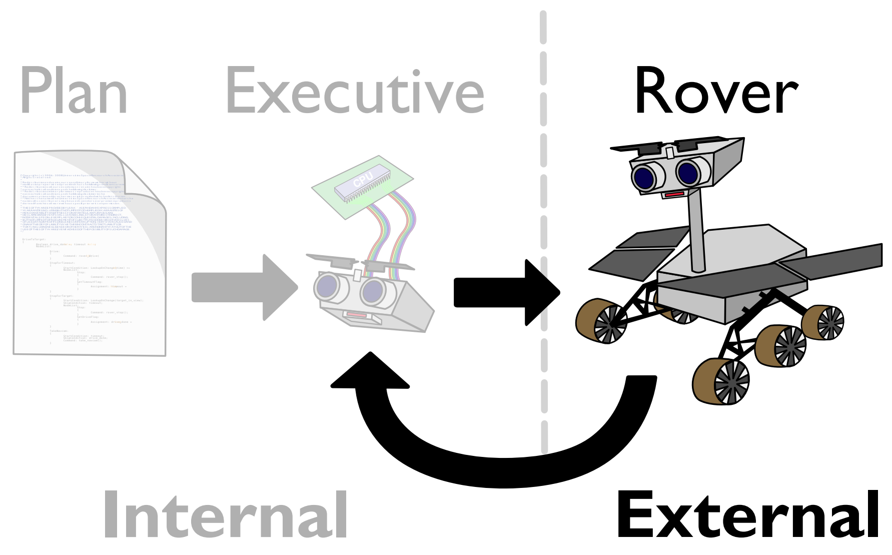


Figure 5.1: PLEXIL simulator, controlling effects in the external world

So far, we have discussed the necessary components to design a PLEXIL plan and update its state internally. We have included a representation for holding data of the external world, but we have yet to define a simulator for an external system.

This chapter discusses the components necessary to describe an external world representation (see Figure 5.1). We will set up the language to represent the hardware, then show some potential functions that affect the status of the world.

5.1 Naming Sensors and Commands

In order to define an external world, we need a way to describe the connections with the PLEXIL interpreter. That is, we only need to know the parts of the external world that a PLEXIL plan can see.

Recall that in previous chapters, our data types have included type parameters. For instance, we had the type `Node s c`. In this case, the `s` and `c` are the data type for sensor names and command names, respectively.

There are a few different ways to define names for sensors and commands. PLEXIL assigns the `String` type to these, which has low overhead in predetermining command name definitions and allows for dynamic command execution. For instance, calling a command with `"Check" + systemName` could be used to run multiple system verification commands, only by changing the value of `systemName`. However, this becomes a hassle for static typing of PLEXIL plans. We will opt to define our own data types, as shown below.

Let us use a simple rover as our example. The rover can be equipped with some wheels, a few cameras, and an internal heater.

```
data RoverSensor =
    PosX | PosY | WheelStuck | MoveDist
  | DirX | DirY | LocalTemp  | NumPics
  deriving (Eq,Show)
```

In order to move, we allow a program to track the rover's location. `PosX` and `PosY` store the current position, while `DirX` and `DirY` give relative, component-wise direction vectors in X and Y directions, respectively. We can also include sensors `WheelStuck` to check if the wheels can spin and `MoveDist` to record the total distance moved since the start of the last movement command.

Along with sensors for movement, we add on two more sensors to the list. The first, `LocalTemp`, stores a temperature reading from an on-board thermometer. The other is `NumPics`, which is a counter for number of pictures taken. This is useful to store in case, say, there is limited storage space for images.

With these sensor names defined, we can use expressions like `LookupFloat PosX` in a program to access sensor values.

For commands, we give the following data type to define potential command names.

```
data RoverCommand =
  Drive | Turn | TakePic | TakePancam | RunHeater
  deriving (Show,Eq,Enum)
```

These commands allow us to activate the hardware components on the rover. `Drive` is meant to move the rover straight forward, and `Turn` will represent a change in direction. `TakePic` and `TakePancam` each activate their own specific cameras, while `RunHeater` turns on a heater.

The command name constructors only store names with no list of command parameters. In this design of PLEXIL, recall the constructor definition `Command c [ExprType s]` for the `Action` data type. We can see the `ExprType` parameters are kept separate from the command name.

We could instead add parameters of the `RoverCommand` constructors. For instance, we could define the constructor `TakePic (Expr s Int)`, where the additional parameter defines the number of pictures to take. However, this ties the command name data type to a specific implementation, which we want to leave as general as possible for now.

5.2 Defining Behaviors

A behavior is an action that transforms sensor values over time.

```
type CommandFunction s = TimeSpan -> [ExprType s] -> [Sensor s] ->
                             ([Sensor s], Maybe (ExprType s))
```

With this type definition, we can see that a command takes, among other values, a list of sensor values and produces a new list of updated sensor values. The result is a tuple, with the second element being a potential return value. The return value is used to signal the end of a command execution on the simulator.

There are two other parameters to consider in the definition. The first is a `TimeSpan`, which is a type synonym for `Float` that represents the number of seconds the behavior should be applied. The second parameter, `[ExprType s]`, is the list of command arguments defined in the PLEXIL plan. As described earlier, this parameter list could be added to the command name data type, which would affect this part of the type definition.

To help design some `CommandFunction` definitions, we first include some helper functions.

```
remove k = filter (\s -> fst s /= k)
insert s@(k,_) = (s:) . remove k
```

The function `remove` uses the name of a key to throw out all pairs of a dictionary, a list of key-value tuples, that have a matching key. The `insert` function uses this definition to add a tuple to a dictionary, replacing any old occurrences of the key with the current value.

```
getConst :: Expr s a -> a
getConst (BConst x) = x
getConst (IConst x) = x
getConst (FConst x) = x
getConst (SConst x) = x
getConst _ = error "Rover has a non-resolved value!"
```

Values stored in sensors will be wrapped in the `ExprType s` union type, which itself will contain an element of type `Expr s a`. The `getConst` function just helps us unwrap the innermost layer of the value, exposing the raw value. Note that we do not have access to the internal `PlexilEnv` in the simulator, so we cannot use the `eval` function. We will expect that all sensor values will always be constants, not unevaluated expressions, and this function will throw an error if this is ever not the case.

```
sensToVal unwrap ss = getConst . unwrap .
                      fromMaybe undefined . flip lookup ss
```

Given a dictionary of sensors and a sensor name, the function `sensToVal` finds the corresponding sensor value and extracts the value out of the `ExprType` wrapper. In case no valid value was found, the function will throw an `undefined` error. The user must supply the unwrapper for getting from `ExprType` to `Expr` values, as seen below.

```
bval :: Eq s => [Sensor s] -> s -> Bool
  ...
bval = sensToVal (\(BoolEx x)   -> x)
ival = sensToVal (\(IntEx x)    -> x)
fval = sensToVal (\(FloatEx x)  -> x)
sval = sensToVal (\(StringEx x) -> x)
```

These four functions give an unwrapper to `sensToVal`. To satisfy Haskell's type system, each possible value type needs its own function. Therefore, we include `bval` for `Bool` values, `ival` for `Int` values, and so on.

We need a default dictionary of sensors at the beginning of execution. We will use the following list.

```
roverSensors = [
  (PosX, float 0),
  (PosY, float 0),
  (DirX, float 1),
  (DirY, float 1),
  (MoveDist, float 0),
  (NumPics, int 0),
  (LocalTemp, float 0),
  (WheelStuck, bool False)
  ]
```

We will use this list to read current sensor values. Then, we can change the values to simulate effects occurring in the external world.

With this set of helper functions, we can begin defining behaviors in a `CommandFunction` context. Let's look at some possible effects we may like to include in our program.

```
takePic :: CommandFunction RoverSensor
```

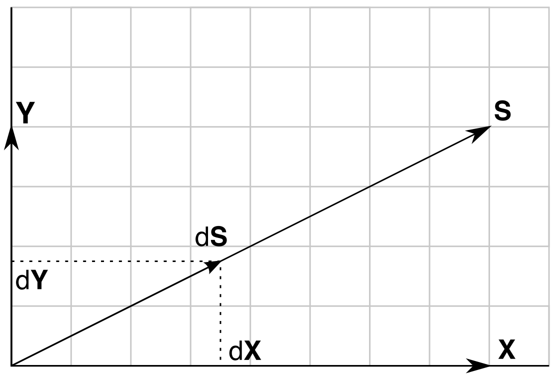


Figure 5.2: Distance calculations represented with Euclidean vectors

```
takePic _ _ ss = (ss',Just $ int 0)
  where ss' = insert (NumPics, int $ 1 + ival ss NumPics) ss
```

In the `takePic` function, we will immediately add a new photo to memory. For our given list of sensors, this means updating the number of pictures currently taken and stored. We can use `ival ss NumPics` to read the current number of pictures. With this, we can increment the value of `NumPics` by adding one to its current value and using `insert` to insert the updated tuple back into the sensor list. The result, `(ss',Just $ int 0)`, will return the updated sensors with a value of 0, indicating no errors during execution.

The next function we define will control movement of the rover. We are storing component-wise direction vectors `DirX` and `DirY` in our sensor list. To calculate the distance traveled, as shown in Figure 5.2, we will need the following formulas.

Given $\mathbf{X} = \langle x, 0 \rangle$, $\mathbf{Y} = \langle 0, y \rangle$, time t, speed s:

$$d\mathbf{S} = st$$

$$d\mathbf{X} = st\frac{x}{\sqrt{x^2 + y^2}}$$

$$d\mathbf{Y} = st\frac{y}{\sqrt{x^2 + y^2}}$$

With these formulas, we can begin calculating updated values for sensors.

```
move :: CommandFunction RoverSensor
move t args ss = (ss', ret)
  where
    [dirX,dirY,posX,posY,dist] = map (fval ss)
                                 [DirX,DirY,PosX,PosY,MoveDist]
    ...
```

The `move` function starts by reading the necessary values from the current sensor values.

```
move t args ss = (ss', ret)
  where
    ...
    spd = 1 -- For now, assume speed 1 unit/second
    len = sqrt $ dirX*dirX + dirY*dirY
    dS = spd * t
    dX = spd * t * dirX / len
    dY = spd * t * dirY / len
    ...
```

Here we use the formulas defined earlier to calculate the total distance moved `dS` and the component-wise distances `dX` and `dY`. For now, we are assuming the rover speed is 1 meter per second, though this can be changed within this function later as necessary.

```
move t args ss = (ss', ret)
  where
    ...
    ss' = foldr insert ss [(PosX,float $ posX + dX),
                           (PosY,float $ posY + dY),
                           (MoveDist,mv)]
    endDist = case args of
      FloatEx x:_ -> getConst x
      _           -> 1
    (mv,ret) = if dist + dS <= endDist
               then (float $ dist + dS, Nothing)
               else (float 0, Just $ int 0)
```

The last part of `move` prepares the result values. The updated position values get inserted into the sensor list to produce the output `ss'` list. The value `endDist` is how far the rover should move, which will either be the only parameter value or a default value of 1 (meter).

Based on the value of `endDist`, we can determine whether or not the rover has finished executing the `Drive` command. The values assigned to `mv` and `ret` show that, should the total distance driven since starting the latest call to `Drive` be greater than `endDist`, we can return a success value of 0 and reset the value of `MoveDist` for the next call to `Drive`. Otherwise, we produce no return value and continue accumulating the distance travelled thus far.

Once we get every `CommandFunction` defined for our simulator, we can put them into a dictionary, as seen below.

```
roverCommands = [
  (Drive, move),
  (TakePic, takePic),
  (TakePancam, takePic),
  ...
  ]
```

Recall that we included the parameter `commlist` in the `ExtWorld` data type. This is a list of tuples with command name and `CommandFunction`, which is used as a dictionary to access simulator behaviors from the PLEXIL executive. We can add new tuples to the `commlist` as new functionality is generated, and the `addCommands` function defined in Section 4.3.4 uses this list to communicate the executive's output commands to the simulator.

Suppose that we want to simulate the `move` function in a less forgiving terrain. We could make a new description of the function to insert issues into the simulator.

```
move2 :: CommandFunction RoverSensor
move2 t args ss
  | posX > 2.5 = (insert (WheelStuck, bool True) ss, Just $ -1)
  | otherwise  = move t args ss
  where posX = fval ss PosX
```

If the rover ends up traveling into an area where it should not go, we could force the wheels to get stuck. In the above case, this will occur anywhere the rover's position in the X direction is beyond 2.5. Note that we do not care what is causing the issue, be it rough terrain or a rock, as long as the effect is the same.

We can arbitrarily update these functions to design different effects of the simulator. By editing the command list in the representation of the external world, we can change the description of the external environment.

5.3 Sample Programs

This section will cover some definitions of PLEXIL nodes and plans that represent the execution of a K10 rover. We will begin with some node definitions.

```
driveOneMeter =
  Node "OneMeter"
    []
    []
    (Command Drive [float 1])
```

The node `driveOneMeter` immediately calls the `Drive` command. It should end when the simulator finishes execution.

```
getPicture =
  Node "TakePic"
    []
    [(Start, (VarStatus "OneMeter" ==: cFinished)
         &&: (VarIntEx "pictures" <: IConst 10))]
    (Command TakePic [])
```

The `getPicture` node will tell the simulator to take a picture. We can see it contains an additional `Start` condition that forces it to only execute when the previous node is done executing and as long as a local variable `pictures` doesn't get too large.

```
countPics =
  Node "Counter"
    []
    [(Start, VarStatus "TakePic" ==: cFinished),
     (Pre, VarIntEx "pictures" <: IConst 10)]
    (Assignment "pictures" (IntEx $ VarIntEx "pictures" +: IConst 1))
```

The `countPics` node increments the local `pictures`. This will only occur after we have taken a picture in `getPicture` node.

Now, we can combine these nodes to make a full program.

```
safeDriveProgram =
  Node "SafeDrive"
  [Var "pictures" (int 0), Var "distance" (float 0.0)]
  [(Invariant, LookupBool WheelStuck ==: cFalse),
   (RepeatWhile, LookupBool WheelStuck ==: cFalse)]
  (NodeList [driveOneMeter, getPicture, countPics])
```

This node, `safeDriveProgram`, is the root node for this example, based off an example from [5]. It creates all the internal variables necessary for this program and puts all the previous nodes into a `NodeList`. It loops the program as long as the rover can drive (the wheels are not stuck). We also include an `Invariant` condition, which causes the program to quit all subnodes immediately if it reaches the error state of being stuck.

To use this plan, we first need to define the external world. We use the following world definition for rovers.

```
roverExtWorld = World roverSensors [] [] roverCommands
```

We put the default lists of sensors and commands available to a `World` constructor, along with empty lists for the initial command calls and command returns. With an external world, we can define an entire `PlexilEnv`.

```
safeDriveEnv = plexilEnv safeDriveProgram roverExtWorld
```

Recall back to Section 4.1.4, where we defined the `plexilEnv` function. This builds the framework for executing PLEXIL plans in the executive. We can run the program now with any of the execution functions defined in 4.3.5. For instance, we could try running the following `traceExt` function.

```
> traceExt safeDriveEnv 150
[external =
[(PosX,0.0),(PosY,0.0),(DirX,1.0),(DirY,0.0),(MoveDist,0.0),...]
[]
[]
,external =
[(PosX,1.0296e-2),(PosY,0.0),(MoveDist,1.0296e-2),...]
[("OneMeter",Command: Drive(1.0))]
[]
,external =
[(PosX,2.5093e-2),(PosY,0.0),(MoveDist,2.5093e-2),...]
[("OneMeter",Command: Drive(1.0))]
[]
,
...
,external =
[(PosX,1.781234),(PosY,0.0),(MoveDist,0.7732531),(NumPics,2),...]
[("OneMeter",Command: Drive(1.0))]
[]
]
```

As the program executes, we can build up a stack trace of the effects of program execution. By using the `traceExt` function, we can focus on the external effects of the PLEXIL plan. In this case, we can see the values of `PosX` and `MoveDist` changing as the `Drive` command is executed.

We could also try running with program on a different external world. Recall the definition of the `move2` function, where the wheels got stuck if the X position of the rover exceeded 2.5. We can design a different `PlexilEnv` that uses this external world definition.

```
roverExtWorld2 = roverExtWorld {commlist = list}
    where list = [(Drive, move2),(TakePic, takePic)]
safeDriveEnv2 = plexilEnv safeDriveProgram roverExtWorld2
```

Once again, we can execute the program with a different external world to see the changes in program behavior.

```
> execute safeDriveEnv2 300
PlexilEnv {
internal =
Var pictures = 2
Var distance = 0.0

states =
State:
  ID: "SafeDrive"
  Status: Finished
  Outcome: Failure

State:
  ID: "OneMeter"
  Status: Finished
  Outcome: Failure

State:
  ID: "TakePic"
  Status: Finished
  Outcome: Skipped

State:
  ID: "Counter"
  Status: Finished
  Outcome: Skipped

external =
[(WheelStuck,True),(PosX,2.5042782),(PosY,0.0),(MoveDist,0.50297904),
 (NumPics,2),(DirX,1.0),(DirY,0.0),(LocalTemp,0.0)]
[]
[]
}
```

By using the `execute` function, we can examine the final program state in full. From this result, we can see that the `WheelStuck` sensor did indeed get set to `True` once the rover's `PosX` went over 2.5 units. The nodes shifted to a `Finished` status, with either `Failure` or `Skipped` outcomes. This shows us that something went wrong, and it provides details to allow us to examine what might have caused the issue.

Chapter 6 – Closing Remarks

This thesis has presented a new implementation of the language PLEXIL with the semantic layers of the Universal Executive embedded in Haskell. The implementation followed the formal semantics defined by NASA to produce accurate results from execution.

The language description was extended by a simulator definition. The simulator included the names of commands and sensors available on a K10 rover, as well as functions to simulate the effects of commands in the external world. The simulator definition was abstracted from executive to allow future users to develop new simulated external environments for further testing separate from the PLEXIL executive.

This design can be used to test the theories on design decisions, discussed in Section 1.1. The project team wants a way to see if altering design decisions in a complicated system can produce meaningful messages about potential issues in a hardware/software design. Both the design of PLEXIL plans and the external world interfaces can be added into the object model representation. Whenever a change is made in either the hardware and software design, the team can use this PLEXIL simulator to check for errors that occur during program execution. Should any errors occur, they can see if their object model would catch these errors in order to prevent them from occurring in real tests. This simulation allows them to verify the effectiveness of their model.

6.1 Project Issues

One of the design goals was to allow for quick translation from PLEXIL plans in Haskell to Standard PLEXIL formatting. This would allow code to better integrate into NASA's open source system as an additional testing tool. The simplest way to translate plans is to add *pretty printing* to all the Haskell data types. Pretty printers are functions that convert data type values to nicely formatted Strings.

This design goal ended up proving to be a major roadblock for refactoring the PLEXIL data type definitions. This PLEXIL definition is notably full of overhead compared with similar languages embedded in functional languages, such as those in Section 2.2. Haskell, along with other functional languages, can use *higher-order functions* as a powerful abstraction tool [13]. Higher-order functions are functions that take other functions as arguments [28]. Higher-order functions can often lead to elegant solutions, but they cannot always be used.

However, functions in Haskell cannot be natively printed. The Haskell compiler GHC [9] will display a message like `No instance for (Show (t -> t1))` when a user tries to print a function by name. By enforcing that PLEXIL plans embedded in Haskell are printable, the available options for data types becomes severely restricted.

This project also ran into an issue with finding technical data on the NASA's integration of PLEXIL on the K10 rovers. However, PLEXIL avoided using a database of command names during execution to improve execution speed [23]. As such, no database of the available command names was ever generated and publicized. The commands listed in Section 5.1 were based on examples either found online or in earlier PLEXIL papers. These examples gave a peek into commands available for a K10, but it was difficult to design an accurate system without a more comprehensive list of commands.

6.2 Future Work

There are a couple of ways this work could be expanded. First, it would be useful to include graphical output of plans in execution.

Currently, the output of this PLEXIL simulator is a textual execution trace. These traces could be used as keyframes for generating animated graphical versions of program execution.

NASA recently added a RoboSim project to their open source release. It is based on the OpenGL graphics library. The program displays rovers in a 2D environment, and it allows interactive input of simple commands such as driving in four cardinal directions. However, it does not currently support the input of complex PLEXIL plans into its simulations.

The SVG image format includes the option for designing animated images. This would be a potential lightweight graphical system for displaying the results of PLEXIL execution. SVG animation is not currently supported by many programs, but should this change it would be a viable option.

Also, this design for PLEXIL execution focused around implementing a simulator for the K10 rover. As NASA begins to implement more hardware systems with PLEXIL, it is recommended to produce a new simulator interface for describing these devices to verify the accuracy of the executive.

The PLEXIL executive could also be extended to use multi-threading in the future. The macro semantics of the executive are still closely tied to the design of the simulator. Using multi-threading in Haskell could allow for better separation of these two devices, which can lead to a wider variety of simulator designs.

Bibliography

[1] Gérard Berry. *The Esterel v5 Language Primer - Version v5_91.* Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, June 2000.

[2] Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical report, Oregon Graduate Institute School of Science && Engineering, 2002.

[3] Maria Bualat, Laurence Edwards, Terrence W. Fong, Michael Broxton, Lorenzo Flueckiger, Susan Y. Lee, Eric Park, Vinh To, Hans Utz, Vandi Verma, Clayton Kunz, , and Matt MacMahon. Autonomous robotic inspection for lunar surface operations. In *6th International Conference on Field and Service Robotics*, July 2007.

[4] Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. The semantic layers of Timber. In *Programming Languages and Systems, First Asian Symposium, Proceedings : APLAS 2003*, volume 2895, pages 339–356, January 2003.

[5] G. Dowek, C. Muñoz, and C. Păsăreanu. A formal analysis framework for PLEXIL. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*, September 2007.

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

[7] Software safety home page. Esterel synchronous language. `http://www.softwaresafety.net/Esterel.org/esterel.html`.

[8] Terrence W Fong, Maria Bualat, Laurence Edwards, Lorenzo Flueckiger, Clayton Kunz, Susan Y. Lee, Eric Park, Vinh To, Hans Utz, Nir Ackner, Nicholas Armstrong-Crews, and Joseph Gannon. Human-robot site survey and sampling for space exploration. In *AIAA Space 2006*, September 2006.

[9] The Glasgow Haskell compiler. `http://haskell.org/ghc/`.

[10] Gregory D. Hager and John Peterson. Frob: A transformational approach to the design of robot software. In *Robotics Research: The Ninth International Symposium*, pages 257–264. Springer Verlag, 1999.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[12] Nicolas Halbwachs and Pascal Raymond. A tutorial of LUSTRE, January 2002.

[13] Haskell. `http://www.haskell.org/`.

[14] Bernard Houssais. The synchronous programming language SIGNAL, a tutorial. IRISA, April 2002.

[15] Robert Morris, Jennifer Dungan, Petr Votava, and Lina Khatib. Coordinated data acquisition on sensor webs. In *Aerospace Conference, 2008 IEEE*, pages 1–7, March 2008.

[16] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[17] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, pages 1144–1151, 1999.

[18] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 91–105, London, UK, 1998. Springer-Verlag.

[19] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM.

[20] Marc Pouzet. *Lucid Synchrone - version 3.0: Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006.

[21] SourceForge. PLEXIL and the Universal Executive. `http://plexil.wiki.sourceforge.net`.

[22] V. Verma, T. Estlin, A. Jónsson, C. Păsăreanu, R. Simmons, and K. Tso. Plan execution interchange language (PLEXIL) for executable plan and command sequence. In *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, volume 603 of *ESA Special Publication*, August 2005.

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

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

[25] Vandi Verma, Vijay Baskaran, Hans Utz, Robert Harris, and Charles Fry. Demonstration of robust execution on a NASA lunar rover testbed. In *'i-SAIRAS 2008' - The 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, February 2008.

[26] Eric Walkingshaw, Paul Strauss, Martin Erwig, Jonathan Mueller, and Irem Tumer. A formal representation of software-hardware system design. In *Proceedings of the 2009 Design Engineering Technical Conference & Computers and Information in Engineering Conference*, 2009. To appear.

[27] Wikibooks. Haskell/GADTs. `http://en.wikibooks.org/wiki/Haskell/GADT`.

[28] Wikibooks. Haskell/higher-order functions and currying. `http://en.wikibooks.org/wiki/Haskell/Higher-order_functions_and_Currying`.

[29] Yampa: Functional reactive programming with arrows. `http://www.haskell.org/yampa/`.

[30] Kris Zacny, Jack Wilson, Arthur Ashley, Chris Santoro, Michael Sudano, Susan Lee, Linda Kobayashi, Terrence W. Fong, , and Matthew Deans. Geotechnical property tool on NASA Ames K-10 rover. In *Joint Annual Meeting of LEAG-ICEUM-SRR*, October 2008.