UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Regression Test-Driven Extension of PLEXIL5: Support for Update Nodes

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Moreno Latorre, Jorge

Tutor: Escobar Román, Santiago

ACADEMIC YEAR: 2022/2023

Resum

Aquest projecte descriu l'esforç d'extensió de PLEXIL5, un intèrpret formal de PLEXIL especificat en el motor de lògica de reescriptura Maude, per aconseguir majors graus de correcció i completitud en comparació amb el PLEXIL Executive, l'intèrpret oficial de PLEXIL. PLEXIL5 està basat en una versió anterior de PLEXIL i ha quedat obsoleta. Aquest projecte pretén introduir suport per als nodes Update de PLEXIL en PLEXIL5. L'estratègia seguida consistix en aprofitar les proves de regressió oficials del PLEXIL Executive. La comparació automàtica entre les execucions de les proves oficials en el PLEXIL Executive i en PLEXIL5 és una mesura de la correcció i completitud de l'intèrpret formal respecte a la implementació de referència.

Paraules clau: PLEXIL, PLEXIL5, Maude, intèrpret, verificació, lògica de reescriptura, proves de regressió, semàntica executable, nodes update

Resumen

Este proyecto describe el esfuerzo de extensión de PLEXIL5, un intérprete formal de PLEXIL especificado en el motor de lógica de reescritura Maude, para lograr mayores grados de corrección y completitud en comparación con el PLEXIL Executive, el intérprete oficial de PLEXIL. PLEXIL5 se basa en una versión anterior de PLEXIL y ha quedado obsoleta. Este proyecto pretende introducir soporte para los nodos Update de PLEXIL en PLEXIL5. La estrategia seguida consiste en aprovechar las pruebas de regresión oficiales del PLEXIL Executive. La comparación automática entre las ejecuciones de las pruebas oficiales en el PLEXIL Executive y en PLEXIL5 es una medida de la corrección y completitud del intérprete formal respecto a la implementación de referencia.

Palabras clave: PLEXIL, PLEXIL5, Maude, intérprete, verificación, lógica de reescritura, pruebas de regresión, semántica ejecutable, nodos update

Abstract

This project describes the extension effort of PLEXIL5, a PLEXIL formal interpreter specified in the rewriting logic engine Maude, to achieve higher degrees of correctness and completeness with respect to the PLEXIL Executive, the official PLEXIL interpreter. PLEXIL5 is based on a former version of PLEXIL and has become deprecated. This project aims to introduce support for PLEXIL Update Nodes into PLEXIL5. The strategy followed consists of leveraging the official regression tests of the PLEXIL Executive. The automatic comparison between the executions of the official tests running on the PLEXIL Executive and in PLEXIL5 is a measure of the formal interpreter correctness and completeness with respect to the reference implementation.

Key words: PLEXIL, PLEXIL5, Maude, interpreter, verification, rewriting logic, regression tests, executable semantics, update nodes

Contents

Contents v
List of Figures vii
List of Tables viii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objectives . . . 3
1.3 Structure of the report . . . 3
2 Background 5
2.1 Rewriting logic . . . 5
2.2 Maude . . . 6
2.3 PLan EXecution Interchange Language (PLEXIL) . . . 7
2.4 PLEXIL Formal Interactive Verification Environment (PLEXIL5) . . . 12
3 Design of the solution 15
3.1 Design of the PLEXIL5 semantics . . . 15
3.1.1 Syntax of plans . . . 15
3.1.2 Internal representation . . . 15
3.1.3 State of the system . . . 17
3.1.4 Node state transitions - The atomic relation . . . 18
3.1.5 The micro relation . . . 20
3.1.6 Launching a macrostep . . . 20
3.2 Architecture of the *plexil2maude* tool . . . 21
3.2.1 Architecture of *plx2maude* . . . 21
3.2.2 Architecture of *psx2maude* . . . 22
4 Methodology 23
5 Workflow 27
5.1 A test-driven approach . . . 27
5.1.1 TDD in plexil2maude . . . 28
5.1.2 TDD in the PLEXIL5 semantics . . . 30
5.2 Use of PLEXIL regression tests as PLEXIL5 acceptance tests . . . 31
5.3 Development workflow . . . 32
5.3.1 The runRegression script . . . 32
5.3.2 Parsing errors . . . 33
5.3.3 Debugging a plan . . . 33
6 Extending the semantics of PLEXIL5 35
6.1 Adding Update Nodes to PLEXIL5 . . . 35
6.2 Fixing the semantics of Command Nodes in PLEXIL5 . . . 37
6.3 Short-circuiting the macro relation . . . 39
6.4 Extending the generation of inputs . . . 41
6.5 Other contributions . . . 44
6.5.1 String concatenation . . . 46
6.5.2 Node *self* references . . . 47

6.5.3 LookupOnChange 48
6.6 Results 49

7 Conclusions and future work **51**

8 Acknowledgements **53**

Appendices

A Sustainable Development Goals **57**

B Code **59**
B.0.1 Code for execassignment 59

List of Figures

2.1 Module that declares commutativity and associativity as axioms 7
2.2 Execution diagram of PLEXIL plans [15] 12
2.3 Execution diagram of PLEXIL plans adapted to PLEXIL5 13

3.1 A compiled PLEXIL plan and its translated counterpart 16
3.2 Node state diagram of assignment nodes in state *executing* 20

4.1 Life cycle of a feature under development in the PLEXIL5 project 25

5.1 Testing in *plexil2maude* 29
5.2 boolean1.ple PLEXIL plan 31
5.3 Empty/trivial script to use with plans that do not require one 32

6.1 Test UpdateTest fails to execute because update nodes are not implemented 37
6.2 Test UpdateTest executes correctly 37
6.3 A command node *myNode* enters execution and creates a command-on-execution node 38
6.4 Simple plan and script that test for command abortion 39
6.5 Test commandabort1 does not execute correctly 39
6.6 Test commandabort1 executes correctly 40
6.7 The macrostep is not short-circuited on update node execution 41
6.8 The macrostep is short-circuited on update node execution, which starts a new macrostep 42
6.9 PLEXIL plan and script that demonstrate the appending of extra empty inputs to the script 43
6.10 Log of the transitions of the example plan under the example script that demonstrates the appending of extra empty inputs to the script 43
6.11 Log of the transitions of the example plan under the example script translated to Maude, executed with the PLEXIL5 semantics before implementing the appending of extra empty external inputs 43
6.12 The nodes *cell* when the plan terminates without appending an extra empty external input, leaving a node executing and suspended 44
6.13 Log of the transitions of the example plan under the example script translated to Maude, executed with the PLEXIL5 semantics after implementing the appending of extra empty external inputs 44
6.14 The nodes *cell* when the plan terminates after appending an extra empty external input, correctly terminating the node that previously did not . . . 44
6.15 Tests concat1 and concat2 lead to errors because the feature is not supported 47
6.16 Tests concat1 and concat2 execute correctly 47
6.17 Test skip1 does not execute correctly 47
6.18 Test skip1 executes correctly 48
6.19 Test array1 executes correctly 49
6.20 Bar chart showing the number of correct tests before and after. 49

List of Tables

2.1 Common default values for node conditions 9
2.2 Default values for node end condition 9
6.1 Table analyzing what features are left to implement into PLEXIL5 45
A.1 Degree to which the work impacts the United Nations' Sustainable Development Goals (SDG). 57

CHAPTER 1
Introduction

This work was developed at the National Institute of Aerospace (NIA) under NASA sponsorship, in Hampton, Virginia, USA. It stems from a stay in March 2023 through a connection that Santiago Escobar, the tutor of this project, shares with NASA researchers Marco A. Feliu and Laura Titolo. The stay was intended to be longer and happen in Q4 2022, but bureaucratic and budgetary constraints resulted in a one month stay in March 2023.

This report addresses contributions to the PLEXIL5 project under Marco A. Feliu's mentorship. PLEXIL5 is an implementation of the semantics of PLEXIL in a rewriting logic engine called Maude. PLEXIL is a language developed by NASA to create software that executes in spacecraft operations and other autonomous systems. Maude is a language that implements rewriting logic, and allows to specify other programming languages and to formally verify properties of systems in said languages.

The PLEXIL5 project is an open-source project that was initially developed by Camilo Rocha et al. as an interpreter for PLEXIL accompanied by a set of graphical tools to aid with creation of programs (called *plans*) and verification. However, PLEXIL has been greatly updated since then and PLEXIL5 has become mostly deprecated. NASA researcher Marco A. Feliu now leads a project that aims to modernize PLEXIL5 and adapt it to the latest official version of PLEXIL as of March 2023, which is PLEXIL 4.6. The version of PLEXIL5 currently under development has been stripped of all graphical tools to focus on the specification of the rewriting logic semantics of PLEXIL. As it is, PLEXIL5 is composed by:

- The executable semantics of PLEXIL in Maude
- A tool (plexil2maude) that translates PLEXIL plans into their analogous in the executable semantics, built in Haskell.
- A tool (plexilog) that compares the execution of the PLEXIL Test Executive with that of the executable semantics.

The development of PLEXIL5 has a special focus on Test Driven Development to guide the implementation of new features. It has an extensive suite of unit tests for both the semantics and plexil2maude.

1.1 Motivation

Every PLEXIL distribution offers a set of tests used by the PLEXIL development team to ensure that PLEXIL operates according to its requirements. They are called *regression tests* [1], and are used whenever new enhancements or new requirements are introduced

into an already developed system. When a new version of a system no longer provides functionality that should be preserved, it has *regressed*. After these modifications are introduced, test cases are run to re-establish confidence that the system will perform according to its requirements. When the changes are enhancements, the test cases need not be modified. Moreover, when changes respond to a modification in the requirements, test-cases may need to be modified or be added to the test base. This ensures that future modifications will be able to test that the behaviour of the system has not been altered in unwanted ways.

Taking the regression test suite provided by PLEXIL as a reference, one may argue that any PLEXIL interpreter would be semantically complete with respect to the original PLEXIL Executive if they behave in the same manner under the same conditions. That is, if the state of a system under test changes in the exact same manner for the PLEXIL interpreter under test as it does for the official PLEXIL Executive, it would be reasonable to assert that they are somewhat equivalent — or, that the PLEXIL interpreter under test is somewhat complete with respect to the official PLEXIL Executive—, to the extent that the PLEXIL development team considers that the PLEXIL Executive meets the requirements if it passes the test cases in the regression test suite.

PLEXIL evolution has been driven by its application on a variety of NASA projects. Some of its most significant uses have come in early versions of said projects. Here are some of its most relevant uses:

- PLEXIL is used for lander autonomy in the open source Ocean Worlds Autonomy Testbed for Exploration Research and Simulation (OceanWATERS) [2], a simulator on a space lander on Jupiter's Europa moon. It is a project intended to help jump-start development of on-board software for potential missions to ocean worlds.
- NASA's Deep Space Habitat and Habitat Demonstration Unit (DSH/HDU) [3] is a functional station to provide living and working conditions for astronauts in deep space missions in adverse environments, such as ones on other planets like Mars, the moon, asteroids or planets in stationary orbit. PLEXIL is part of the DSH/HDU software, and was successfully tested for two years to provide control of several subsystems.
- The Drilling Automation for Mars Exploration (DAME) application is a system that uses PLEXIL to control a fully automatic drill rig intended to use in Mars, and has been successfully tested at a lunar/martian impact crater analog for fault diagnostics, recovery and control of drilling [4].
- PLEXIL is the base of an Autonomy Operating System (AOS) for Unmanned Aerial Vehicles (UAVs) named Pilot-in-a-box [5], an Artificial Intelligence driven open flight software platform. It aims to create a collection of apps that allow for autonomous pilot-less air vehicles to provide mobility-on-demand transportation.
- PLEXIL has been used to demonstrate automation for International Space Station (ISS) operations.

The aforementioned applications showcase the capacities of PLEXIL and the potential that systems built with PLEXIL have. Providing a framework that helps with formally verifying that these systems have the properties that are required of them may help in spreading its use. In a world with prospects of furthering space exploration, providing more ubiquitous and efficient flight transportation, automating housing and factory systems, etc., PLEXIL may help build the systems that tackle whatever problems society faces in the future.

1.2 Objectives

The first project objective is to give support to all types of PLEXIL nodes. This involves introducing support for Update and LibraryNodeCall node types and completing the support for Command nodes. These modifications entail three challenges: a comprehension of the current state and limitations of the PLEXIL5 semantics, a careful analysis of the execution of test plans to identify any unsupported features, and adding any other essential features necessary to incorporate these node types.

The second objective is to revise the current execution semantics of PLEXIL5 via a thorough comprehension of how PLEXIL plans execute. This revision is necessary to handle unforeseen execution scenarios that may arise during development.

Additionally, the project aims to incorporate any essential features required for conducting new, previously unsupported individual tests within the regression test suite. This will involve carefully analyzing the execution of test plans to identify any unsupported features.

Consequently, all advancements in terms of new features of the PLEXIL5 semantics require including accurate translations of PLEXIL elements into the syntax defined in the PLEXIL5 semantics for those elements.

On another note, the project aims to promote interest in formal system verification by providing a tool for verifying PLEXIL systems, recognizing the crucial role autonomous systems will play in the future of human development. With the understanding of the future widespread utilization of these types of systems, verification can act as a cross-cutting axis for secure development in critical environments where human lives may be at stake.

1.3 Structure of the report

This document presents the contributions made to the PLEXIL5 project, extending the current functionality to obtain a greater degree of completeness with regards to the PLEXIL Executive. It starts by outlining the context of the project in Chapter 2, the framework and language that have been used, the structure and capabilities of the PLEXIL language and a description of PLEXIL5. Then, the design of the PLEXIL5 project is explained in Chapter 3. Moreover, the methodology followed and the workflow are detailed in Chapters 4 and 5, respectively. In Chapter 6, the main contributions to the project are presented in depth, emphasizing on their success in replicating the behaviour of the PLEXIL Executive and the results of the extension efforts. Finally, in Chapter 7, the future work and conclusions are discussed.

CHAPTER 2
Background

2.1 Rewriting logic

Rewriting logic is a framework that describes the behaviour of a system in the form of rewriting rules, which are functions that specify how terms are rewritten into other terms. Precisely, rewriting logic reasons about change in a system. It provides information about how a system can develop — how it can change given its current state. At each point in time, rewriting rules represent the basic actions that are possible in a system. They are made up of a left-hand side pattern that matches a term, and a right-hand side that specifies how the matched term is rewritten.

Intuitively, when a system is in a given state made up of terms, different terms may be able to be rewritten in different, non-overlapping ways. This implies that systems modeled by rewriting logic are concurrent by nature [6]. Thus, many of the simplifications that rewriting rules can perform in a system can be performed in parallel. That is, rules with disjoint left-hand sides can be applied concurrently.

Rewriting logic uses a non-deterministic execution model, where the order of application of rules is not predetermined and is chosen during runtime. Fine-grained control over the the execution of rules is given by side conditions, which specify constraints that must be fulfilled in order for a rule to be applied.

Moreover, if all rules in a rewriting system are deterministic and terminating, the outcome of the system is not influenced by the order of application of the rules. This is known as confluence, and it assures that the rewriting system will have a consistent and predictable behaviour.

Formally, a rewriting logic specification, denoted as $\mathcal{R} = (\Sigma, E, R)$, consists of three components[7]:

- Σ: A signature that defines the set of symbols (e.g., functions, constants) and their types used in the specification.
- E: An equational theory that consists of a set of equations involving the symbols in Σ, which define the equalities between terms built from these symbols. The equational theory induces a congruence relation denoted as $=_E$ on the set of ground terms T_Σ, where $(\Sigma, E) \vdash t = u$ means that t and u are equivalent under E.
- R: A set of rewrite rules that specify the transformations or transitions between terms in T_Σ. They consist of a left-hand side (LHS) and a right-hand side (RHS), and specify that any occurrence of the LHS in a term can be replaced by the RHS. These rewrite rules define a rewrite relation denoted as $\rightarrow_R$ on the set of E-equivalence classes of ground Σ-terms, denoted as $\mathcal{T}_{\Sigma/E}$. The rewrite relation $\rightarrow_R$ captures the concurrent transitions between terms in $\mathcal{T}_{\Sigma/E}$ that can be generated with R.

The initial reachability model of R is a tuple $\mathcal{T}_R = (\mathcal{T}_{\Sigma/E}, \rightarrow_R)$ that represents a concurrent system where the states are the set of E-equivalence classes of ground Σ-terms, and the transitions between states are specified by the rewrite rules in R. The reachability model captures the possible state transitions of the concurrent system and can be used to analyze properties such as reachability, termination, or confluence.

In rewriting logic, a normal form is a term that cannot be further reduced by the rewrite rules R specified in the system. In other words, a normal form is an irreducible term with respect to the rewrite relation $\rightarrow_R$. Normal forms allow to reason about the properties of a term without having to consider all its possible reductions.

2.2 Maude

Maude is a high-performance language and a versatile system that supports rewriting logic as its computational paradigm [8]. Developed by researchers from different universities and companies, Maude is an ideal tool for the specification, analysis, and implementation of concurrent systems due to its formal foundations in rewriting logic. Maude provides a syntax for specifying the three components of a rewriting logic specification: the signature Σ, the equational theory E, and the rewrite rules R. Additionally, Maude comes with a powerful and efficient rewriting engine that can execute the rewrite rules specified in the system, allowing for the simulation and analysis of the concurrent system's behaviour. Moreover, Maude provides a rich set of built-in features for formal reasoning, including support for model checking [9], theorem proving [10], and static analysis. These tools can be used to verify properties of the concurrent system, such as reachability, termination, or confluence, which are essential for ensuring the correctness and reliability of the system.

In Maude, a rewriting logic specification splits the equational theory E into oriented equations E and commonly occurring axioms Ax such as associativity, commutativity and identity, so that $\mathcal{R} = (\Sigma, E \cup Ax, R)$[7]. These axioms are used to perform matchings modulo Ax, to produce a finite number of Ax-matching substitutions, which determine when two terms can be rewritten using $\rightarrow_R$.

Figure 2.1 shows a file that defines a simple module called "SIMPLE-SUM" that deals with natural numbers, represented by the sort[1] `Nat`. It declares three operators: `0` represents the number zero, `s_` represents the successor function, and `_+_` represents integer addition. It also declares three variables `X`, `Y`, and `Z`, all of type `Nat`.

The attribute declaration `[assoc comm]` after the `_+_` operation holds a significant level of interest. These attributes stand for "associativity" and "commutativity." They specify that the addition is both associative and commutative, meaning that changing the grouping or the order of addition does not change the result of the operation. With these attributes, the representation of addition expressions can be simplified. For example, using the commutative axiom `X + 0 = X` can evaluate `2 + 0` and also `0 + 2` to get 2. Likewise, the axiom `X + s(Y) = s(X + Y)` can evaluate `2 + s(3)` to get `s(2 + 3)` or `s(5)`.

In contrast, without the `[assoc comm]` attribute, the behaviour dictated by the aforementioned axioms would require explicit definition to cover all possible combinations of the addition. As a result, additional mathematical properties should be introduced like `X + Y + Z = (X + Y) + Z` and `X + Y = Y + X` to ensure the desired properties of associativity and commutativity.

[1] In Maude, data types are known as *sorts*. Its subtypes are consequently known as *subsorts*.

```
fmod SIMPLE-SUM is
    sort Nat .
    ops 0 : -> Nat
        s_ : Nat -> Nat
        _+_ : Nat Nat -> Nat [assoc comm] .
    vars X Y Z : Nat .

    eq X + 0 = X .
    eq X + s(Y) = s(X + Y) .
endm
```

Figure 2.1: Module that declares commutativity and associativity as axioms

By using attribute declarations in Maude, the specification of operations can be simplified by providing a concise and powerful way to express their desired properties, resulting in more elegant and compact code.

2.3 PLan EXecution Interchange Language (PLEXIL)

PLEXIL (PLan EXecution Interchange Language) is a synchronous language developed by NASA that provides support to autonomous spacecraft operations. Traditionally, command sequences to be executed by spacecraft were crafted on the ground and then transmitted to the spacecraft. These command sequences were received and executed in the real world by a software system in the spacecraft called an executive. Execution systems are useful when spacecraft must behave somewhat autonomously, as it must be able to take into account the state of the environment and of the spacecraft itself. In some missions, such as some human-controlled like the Mars Exploration Rover (MER), command sequences were written from Earth. When the spacecraft found uncertainty in the execution of complex commands, such as the placement of one of its arms, it was instructed to take a picture of the environment to be analysed on the ground. With this analysis, a new command sequence could be crafted and sent to the spacecraft to continue the mission. Evidently, this methodology was clumsy, not in small part aided by the fact that execution languages were not too powerful, lacking functionalities such as loops or floating point operations.

PLEXIL presents itself as a solution to this problem, being lightweight, simple, efficient, semantically clear and expressive [11]. Using PLEXIL, command sequences can be built such that a spacecraft is able to react to an ever-changing environment. This reactions can be expressed in PLEXIL programs. In this way, spacecrafts would be able to execute command sequences crafted on the ground while still reacting to changes in the environment to protect their safety safety.

PLEXIL programs contain detailed sequences of actions known as plans that autonomous systems must follow during spacecraft operations and as response to environment changes. PLEXIL also provides an execution engine for plans called the PLEXIL Executive, which allows for execution of plans in real scenarios [12]. The PLEXIL Executive runs aboard the spacecraft and uses some external interfaces in order to communicate with the external system. The Executive can sense the environment using lookups, which only read values of the state of the external system. Lookups can either be immediate via LookupNow or continuous via the event-based LookupOnChange.

PLEXIL nodes

PLEXIL plans are essentially a tree of execution nodes that govern the control of the spacecraft, specifying some kind of behaviour. There are different types of nodes, which are differentiated depending on the functionality they provide. Nodes are denoted between curly braces and may have a name (`MyNode: {...}`). Here are the different types of nodes:

- List Nodes: the interior (branch) nodes in a plan. They contain a set of children nodes of any type. Every List node is the parent node of its children nodes.
- Leaf Nodes
  - Empty node: a node that does not perform any action.
  - Assignment node: a node that performs a local computation and assigns the resulting value to a variable
  - Command node: a node that emits a command to an external system via the external interface. Commands are executed asynchronously.
  - Update node: a node that outputs information through the external interface of the system.
  - Library call node: a node that invokes nodes that belong to external libraries.

PLEXIL node states

An individual PLEXIL node is executed by performing state transitions that advance them through their life cycle. A node can be in one of the following states: *inactive*, *waiting*, *executing*, *finishing*, *iteration ended*, *failing* or *finished*.

PLEXIL node attributes

PLEXIL nodes have attributes that hold information about themselves. Some attributes are related to the execution of the nodes and will be discussed along this section. They may declare local variables that are stored as node attributes, as well as input and output variables that are usually used for nodes to be called from other nodes (these are Library Node Calls referenced in Section 2.3).

However, some types of nodes have special attributes:

- Command nodes
  - Arguments: information passed to the command
  - Command handles: the status of the execution of a command. They are received from the environment via acknowledgements. They indicate whether a command has:
    - Been sent to the system.
    - Been received by the system.
    - Been accepted.
    - Succeeded.
    - Failed.
    - Been Denied.
    - Produced an error in the external interface.

  A command node may fail or be exited. In that case, the external system that is executing the command signals this occurrence to the plan via the external interface in the form of a *Command Abort* to stop the execution of said command node. It is accompanied by a boolean value that indicates the completion of

the abort. The official PLEXIL implementation in C++ currently only considers this value to possibly be true. In Section 6.2, where *Command Aborts* are introduced into the PLEXIL5 semantics, this behaviour is reproduced.
  - Command result: a value that indicates the result of the command, which is usually assigned to a variable in the plan.
  - Destination variable: in-scope variable that is the target of the command.

- Update nodes
  - Pairs: a list of associations of variables of the interface with their corresponding values to be updated.
  - Acknowledgement: A boolean value that indicates whether an update of the interface has completed

PLEXIL node conditions

The execution of nodes is governed by a set of conditions evaluated at different points during the execution of plans. Fulfillment of these conditions may cause a node or its children nodes to change its state [13]. Node conditions are a type of node attributes. While in this section node conditions are referred to as simple true or false values, they may also be unknown values or complex boolean expressions that depend on other variables and results of lookups, commands and library calls. They are evaluated on an as-needed basis.

The following conditions are known as *gate conditions*, are user-defined and commence a transition.

- *Skip Condition*: controls whether node execution is skipped.
- *Start Condition*: controls whether node execution is allowed to start.
- *Invariant Condition*: controls whether node execution is allowed to complete, by checking that the condition is not broken during execution. For this reason, it may be considered a *check condition*, explained later.
- *Exit Condition*: controls whether node execution is exited, cancelling it.
- *End Condition*: controls whether node execution is ended,
- *Repeat Condition*: controls whether node execution is repeated, simulating a loop.

Condition	Default Value
Start	True
Skip	False
Pre	True
Invariant	True
Exit	False
Repeat	False
Post	True

Table 2.1: Common default values for node conditions

Table 2.1 shows the default values for conditions that are common between nodes. Table 2.2 shows the default values for *End Condition*.

The following conditions are known as *check conditions*, are also user-defined and

Node Type	End Condition
Empty	True
List, Library Call	All children finished
Command	Command failed or denied
Assignment	True
Update	Update acknowledged

Table 2.2: Default values for node end condition

can modify whether a transition is performed or not. If these conditions are not evaluated to true, the transition they govern is not executed.

- A *Precondition* checked after the *Start Condition* becomes true.
- A *Postcondition* checked after the *End Condition* becomes true.

Internally, PLEXIL keeps a set of conditions for every node that are not user-defined, but are checked during execution:

- The *Invariant Condition* of any of the node's ancestors becoming false.
- The *Exit Condition* of any of the node's ancestors becoming true.
- The *End Condition* of any of the node's ancestors becoming true.
- Whether all children of a node are in state *Waiting* or *Finished* (`All_children_waiting_or_finished`).
- The *abort* for a command completing.
- Whether the parent of a node transitions into state *Waiting*
- Whether the parent of a node transitions into state *Finished*
- Whether the parent of a node transitions into state *Executing*

While these conditions are common for all nodes in a plan, some may not be applicable to some types of nodes — e.g. a leaf node does not have children nodes, so `All_children_waiting_or_finished` does not apply to its execution.

Node execution starts with all nodes initialized to state *Inactive* except for the root node, which is initialized to *Waiting*. The exact execution semantics of each node type can also differ, and can be found in [14]. The case of *Update nodes*, an emphasis of this work, will be explored in detail in Section 6.1

PLEXIL nodes also hold information about the outcome of their execution in an *Outcome* attribute, which is initially *Unknown*, but can become:

- *Success* when the node terminates normally
- *Failure* when a node in states *Executing* or *Iteration Ended* has its *Invariant Condition* become false
- *Interrupted* when a node in states *Executing* or *Iteration Ended* has its *Exit Condition* become false
- *Skipped* when a node in states *Inactive* or *Waiting* has its *Skip Condition* or *Invariant Condition* become true, its parent's *End Condition* or *Exit Condition* become false, or its parent transitions to *Finished*.

PLEXIL node termination

Nodes terminate normally under different conditions depending on its type:

- Empty and assignment nodes: immediately when it transitions to *Executing*
- Command node: when an acknowledgment of the command, known as *command handle*, is received.
- Update node: when an acknowledgment of the update is received.
- Library call node: when its child has finished

- List node: when all its children have finished

In the case where a node terminates abnormally and has an *Outcome* of *Failure*, the information of the reason behind the failure is also held by the node. It is initialized to *Unknown* and can take one of the following values:

- Precondition failed
- Postcondition failed
- Invariant condition failed
- Parent failed
- Exit condition became true
- Parent exited

Execution of PLEXIL plans

PLEXIL plans are executed following five relations that constitute its small-step semantics. Each of these relations constitutes a set of steps in the execution of the plan. [12]

- Atomic relation: describes an individual node's state transition
- Micro relation: describes the synchronous execution of the atomic relation for all active nodes in a plan.
- Quiescence relation: describes the repeated run of the micro relation until completion
- Macro relation: describes the execution of a plan inbetween queries to the environment.
- Execution relation: describes the N-step iteration of the macro relation.

Figure 2.2 is a diagram that represents the execution of a PLEXIL plan. The process is started with the execution relation, which evaluates all gate conditions for the root node. When an external input is received — in the form of responses to lookups or commands — a macrostep is set afoot. A cycle of quiescence is performed by repeatedly applying microsteps. Microsteps may only modify local data in the executive, that is, local variables, or node states and outcomes. Microsteps execute the atomic relation in parallel for all nodes, with priorities used to resolve conflicting assignments [12]. A cycle of quiescence completes when all nodes have no enabled transitions left to perform — they have achieved what in Section 2.1 is called their *normal form*. In that case, the system has achieved quiescence. The macrostep has ended and the system reads the next external input, beginning a new macrostep. This behaviour is known as run-to-completion semantics. External inputs are processed in the order they arrive. Gate conditions are evaluated and, if some nodes become active, a new cycle of quiescence is performed.

PLEXIL scripts

To simulate execution of PLEXIL plans, the PLEXIL Executive cannot provide a representation of the real world on its own. Therefore, complementary to the PLEXIL Executive there is a Test Executive that interleaves execution of a PLEXIL plan with simulated external events that represent the external world [16]. With the Test Executive, the scripts drive the execution of the plan. A PLEXIL script consists of an initial state (`initial-state`) and a sequence of events (`script`). Test execution starts by applying the initial state and advancing the executive one macrostep. Then, the sequence of events is provided to the executive in order and one at a time. In PLEXIL scripts, `initial-state` defines a set of state variables with their corresponding initial values, which represent lookups that

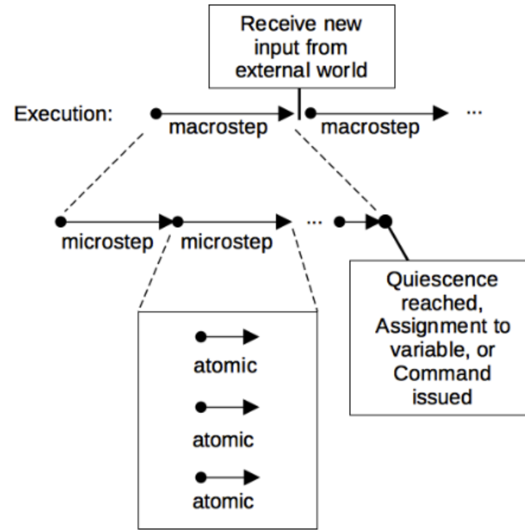


Figure 2.2: Execution diagram of PLEXIL plans [15]

would appear in the interface of a plan, reading from the real word. The variables in the state are those that may be updated from the plan via update nodes and read via lookup nodes. The sequence of events (`script`) contains the definition of state variables, results and aborts of commands, acknowledgements of commands and updates, delays and messages sending plans to the executive. It is also possible to define the simultaneous provision of any number of external inputs to the plan, which is useful to model real systems where inputs may happen at the same time.

2.4 PLEXIL Formal Interactive Verification Environment (PLEXIL5)

PLEXIL Formal Interactive Verification Environment (PLEXIL5) is a tool that implements a formal executable semantics of PLEXIL in Maude [17]. PLEXIL5 serves as a formal interpreter of the PLEXIL language.

PLEXIL5 is accompanied by a tool that translates PLEXIL plans and scripts to their representation in Maude. This translator, called *plexil2maude*, is implemented in Haskell and is divided into two tools: *plx2maude* for plans and *psx2maude* for scripts. PLEXIL plans and scripts, once compiled, are in formats *.plx* and *.psx* respectively, which are based on XML. Therefore, parsing the compiled forms of plans and scripts is a matter of traversing the XML tree and obtaining the information into a set of internal data structures that can be then translated into the syntax defined in the formal semantics of PLEXIL. The transformations are based on the PLEXIL XML Schema Definition, which can be found in any PLEXIL distribution under `plexil/schema`.

Additionally, PLEXIL5 also incorporates a tool called *plexilog* that compares the output of the PLEXIL Test Executive and the PLEXIL5 Executive to find differences between state transitions of nodes. However, this tool was already fully implemented and is not the focus of this work.

One of the core differences between a synchronous language such as PLEXIL and the rewriting logic framework implemented by Maude is that Maude's operational semantics is asynchronous [17]. This includes maintaining a set of operations such as assignments, commands and updates that must not be performed during any microstep, to account for the asynchronous execution that Maude follows and emulate the synchronous nature

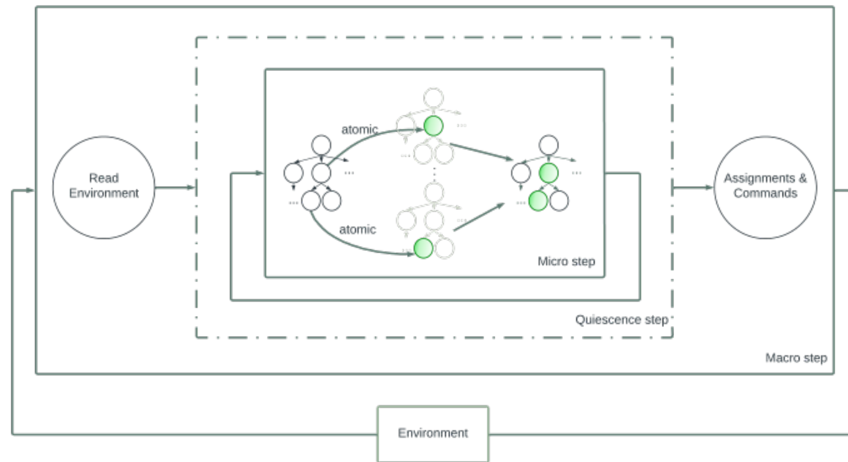


Figure 2.3: Execution diagram of PLEXIL plans adapted to PLEXIL5

of PLEXIL. These operations, as well as state changes, are performed at the end of a quiescence to account for changes in values while the macrostep is still in execution. Section 6.3 dives into the details of these kinds of operations. A full diagram that shows how the execution diagram of PLEXIL is adapted for PLEXIL5 is shown in Figure 2.3. The first step is, similarly to the PLEXIL Executive, to read the environment, which is done in order and initiates the run-to-completion execution. The internal state of nodes are reduced in parallel atomic steps to conform a sequence of microsteps. The conclusion of this sequence leaves the system in a state of quiescence, which conforms a macrostep. Once the macrostep ends, the environment is updated and the cycle begins again.

Modelling the environment is also a delicate topic, since one must make sure that events are provided to the Executive in order but simulating the non-deterministic nature of the external environment by randomly choosing inputs when may potential simultaneous inputs are received. Moreover, the input generator must account for the internal behaviour of the Test Executive, which generates empty inputs when a script ends but some nodes in the plan are still executing, in order for them to finish execution. This behaviour is detailed in Section 6.4. However, if a script ends with nodes waiting to execute and that cannot transition without external inputs, the plan will end in an unfinished state [16].

CHAPTER 3
Design of the solution

3.1 Design of the PLEXIL5 semantics

3.1.1. Syntax of plans

PLEXIL plan syntax in the PLEXIL5 semantics is given by a root node that recursively contains all the hierarchy of the plan. Every node has an identifier, local variable declarations and a set of attributes. Depending on the node type, they have extra information that is required by the semantics, except for the empty node, which has none.

- List nodes have a list of children nodes.
- Command nodes have information about the command they execute, which includes the name of the command, the parameters of the operation and the variable where the result is stored.
- Assignment nodes indicate what variable they modify and the expression they use.

Figure 3.1 shows a simple translated PLEXIL plan. It shows how the Maude translation of plans is conceptually similar to how plans are compiled into `.plx` form, in the sense that every node in the plan can be traced back to the root node, and every node holds all its information. However, due to the structure and verbosity of XML, Maude translations are much shorter and more readable. For reference, the original PLEXIL plan is shown in Figure 3.1a.

The PLEXIL compiled plan is shown in Figure 3.1b. It defines a lookup of an external variable `flag` to the environment in element `StateDeclaration` that returns a boolean value, and it is made up of an empty node with the element `Node` that has a user-defined precondition, where the value is obtained from executing the declared lookup in the element `LookupNow`. The Maude translation is shown in Figure 3.1c. This Maude module also has an empty node, declared with the operator `empty` with a precondition `pre` that executes the lookup `flag` using the operator `lookupNow`. However, the PLEXIL5 semantics does not require the declaration of lookups, because the identifiers for lookups are evaluated at runtime by searching for the *looked-up* variable in the environment of the plan. This is expanded in Section 3.1.3.

3.1.2. Internal representation

PLEXIL plans and scripts translate nodes into a set of constructors for every node type and a set of arguments that usually includes a node identifier, a set of local variables, a set of node attributes for node conditions and as any other attributes specific to the different

```
Boolean Lookup flag;

boolean1:
{
    PreCondition LookupNow(flag);
}
```

(a) Simple PLEXIL plan

```
<?xml version="1.0" encoding="UTF-8"?>
<PlexilPlan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    FileName="plexil/test/TestExec-regression-test/plans/boolean1.ple">
    <GlobalDeclarations>
        <StateDeclaration>
            <Name>flag</Name>
            <Return>
                <Name>_return_0</Name>
                <Type>Boolean</Type>
            </Return>
        </StateDeclaration>
    </GlobalDeclarations>
    <Node NodeType="Empty">
        <NodeId>boolean1</NodeId>
        <PreCondition>
            <LookupNow>
                <Name>
                    <StringValue>flag</StringValue>
                </Name>
            </LookupNow>
        </PreCondition>
    </Node>
</PlexilPlan>
```

(b) Simple compiled PLEXIL plan

```
mod boolean1-PLAN is

protecting PLEXILITE-PREDS .

op rootNode : -> Plexil .
eq rootNode =
  empty('boolean1,nilocdecl,((pre: (lookupNow('flag,(nilarg)))))) .

endm
```

(c) Simple translated PLEXIL plan

Figure 3.1: A compiled PLEXIL plan and its translated counterpart

node types. The PLEXIL5 interpreter performs a compilation step into an internal representation prior to any plan execution. This step also creates any locally declared node variables in the state of the system, more on Section 3.1.3.

Internally, every piece of information that makes up a plan is represented as an object. A node object in PLEXIL5 is given an identifier O, a type C and a set of attributes and values $a_i : v_i$ and becomes a term:

$< O : C \mid a_1 : v_1 , \ldots , a_n : v_n >$

These objects do not only represent PLEXIL nodes, but also PLEXIL5 nodes that hold extra information of the system. The following PLEXIL5 node object types are currently supported:

- PLEXIL node objects: represent nodes in PLEXIL
- Memory node objects: represent the memory of the system.
- Interface node objects: represent nodes that are executed asynchronously. Currently only commands that have been sent for execution and are waiting for a result, but this work introduces update nodes under execution in Section 6.1.

Identifiers in PLEXIL5 are of sort `Qid`, which is a type offered by the module `QID`, predefined in Maude. Identifiers are preceded by an apostrophe (`'`), and consist of a non-empty string that does not contain white spaces or the special characters `'{'`, `'}'`, `'('`, `')'`, `'['`, `']'` and `','`[1]. PLEXIL5 implements identifiers for PLEXIL nodes and variables. Since nodes and variables can be contained inside other nodes, the fully qualified ids are obtained via the functions `getFullyQualifiedNodeId` and `getFullyQualifiedVariableId`, which compute for each id its full *lineage* (the path in the node tree that leads to the current node) in the form:

```
nodeid . parentid . grandparentid . [...] . rootid
```

This eliminates the possibility of name collisions in nodes and variables that have the same name but are inside different nodes. Because the different nodes will have a different *lineage*, their fully qualified ids will be different. In the case of two variables with the same name, it enables their values in the memory to be different.

Moreover, the object representation of plan nodes allows for the specification of all the node attributes laid out in the PLEXIL documentation, as well as all the attributes that are internal to the Executive.

3.1.3. State of the system

The state of the system is represented as a sort named `Config`. It is made up of *cells* that hold information about different parts of the system. *Cells* can be classified according to the role they play in the description of the system.

The following *cells* provide internal information of the system.

- `nodes`: Holds information about the node hierarchy of the plan. It consists of a list of PLEXIL node objects.
- `environment`: Holds information about external variables, which are those declared as lookups.
- `memory`: Holds information about all the local variables of the system. For each variable that has been locally declared it includes its current, previous and original valuea. It consists of a list of memory node objects.

[1] Maude manual at https://maude.cs.illinois.edu/w/images/e/ee/Maude-3.0-manual.pdf

The following *cells* are related to the communication between the system and the external environment.

- `interface`: Holds information about the external environment. It is the way that the plan communicates with the external environment. For example, `interface` can contain the result or acknowledgement of commands. In short, it contains executing nodes that depend on inputs from the environment. It consists of a list of interface node objects.
- `generator`: Because the external inputs are modeled according to the inputs defined in a script, a generator holds all external inputs defined in the script and that will be orderly provided to the system.

The following *cells* emulate the synchronicity of PLEXIL. Because PLEXIL node state transitions happen in parallel but Maude transitions the system step by step, any change to the state of a node must be recorded to be executed at the end of a micro or macrostep.

- `microacts`: Holds information about the actions that must be performed at the end of a microstep.
- `macroacts`: Holds information about the actions that must be performed at the end of a macrostep.
- `trace`: Tracks the actions that have been performed to the system.

Plans are compiled into the aforementioned state of the system via a compilation module that takes a translated PLEXIL plan and generates the internal representations of the node hierarchy and the memory, as well as defining an empty environment, empty interfaces and void lists of *microactions* and *macroactions*. The trace is also empty and the generator is provided to the compiler according to the script.

After compilation by the PLEXIL5 interpreter, the system `Config` is used to create a `GlobalConfig` that is made up of the state of the system and also the execution step, or operation, that the system is found in. It is declared as a sort built with the operator `|` in the form `Operation | Config`.

```
op _|_ : Operation Config -> GlobalConfig .
```

The following operations are allowed:

- `start`: operation that starts the execution of the system.
- `macro`: operation that indicates that a new macrostep must take place selecting the next input.
- `macroND` (renamed to `NDChoice` during the duration of this work): operation that takes a list of possible sets of external inputs as an argument and indicates that a new macrostep must take place by non-deterministically (randomly) selecting one of those sets of inputs.
- `quiescence`: operation that indicates that the system has achieved quiescence and a new macrostep can start.
- `micro`: operation that indicates that a new microstep must take place.
- `stop`: operation that indicates that the system has reached termination.

3.1.4. Node state transitions - The atomic relation

In the atomic relation, node state transitions are modeled according to the PLEXIL node state diagrams, which can be found in [14]. An example diagram can be found in Figure 3.2. They are a set of diagrams that represent how every type of node may transition

its state, depending on the state that it starts with. They include the conditions that determine what state the node ends in.

PLEXIL node state diagrams are represented in a visual modeling language similar to UML[2]. Arrows are used to direct the flow of the diagram. They use squared boxes with rounded edges to represent the beginning and destination states, diamond-shaped boxes to represent the aforementioned conditions. The exiting arrows may be labeled to indicate flow control (T for true, F for false and U for unknown), and squared boxes with straight edges are used to represent actions that must be taken along the way, for example, as in Figure 3.2, changing the outcome attribute of a node to *failed* when a node fails while also changing the reason for the failure in another attribute.

To do so, operations that query the state of the system to reason about what nodes should start or halt execution are defined. In general, these operations look at the value of attributes of either the nodes under examination or their immediate ancestor. To exemplify this behavior, Figure 3.2 shows the node state diagram for assignment nodes that are found in state *executing*.

The first condition requires evaluation of the exit condition of the parent of the node under examination `Q`. Then, a function `ancestorExitTrue?` is defined such that it recursively traverses the node tree to find the parent of `Q` and evaluates its exit condition, held in the attribute `exitc:_`. Conveniently, the ancestor of `Q` can be easily found due to the qualification of every node as a concatenation of all nodes in the node tree, so, knowing that qualifier of `Q` will be of form `Q . parentofQ`, Maude can use pattern matching to efficiently find `parentofQ` and extract `exitc:_`.

Additionally, the last condition to check has to do with the post condition of `Q`. It is easily accessible from the attribute `postc:_`, and can be obtained by matching the pattern `postc:_` with the attributes of `Q`. To evaluate its value, a function `eval` is defined such that it transforms whatever condition is stored in, in this case `postc:_`, and obtains a boolean value `true`, `false` or `unknown`.

A full implementation of the function that takes care of the transitions of assignments nodes from *executing* in the atomic relation can be found in annex B.0.1 as an example of what atomic relation functions look like. The function uses pattern matching to execute the transition of node `A`, of type and status executing by obtaining a node that matches the pattern `< A : assignment | status: executing, ... >`. From there, the function follows the corresponding node state diagram querying the appropriate conditions and setting any required actions.

Actions are not performed immediately, but are instead set to be performed at the end of the current micro or macrostep, as outlined in Section 2.4. To signal that, they are added to the `microacts` or `macroacts` *cells* respectively. These actions may include: setting the outcome of node `A` with function `setOutcome`, setting its status with function `setStatus` or undoing an assignment using function `undoMem`. The first two update the outcome and status attributes in the node `A` and are performed at the end of the microstep, and the latter modifies the memory *cell* at the end of the macrostep.

However, in the current PLEXIL5 semantics not all operations are implemented correctly and some node transitions have changed and need fixing, as is the case with command nodes, whereas transitions for update nodes are not implemented because the node type was not supported.

[2]UML (Unified Modeling Language) is a standardized visual modeling language used in software engineering to depict, specify, construct, and document the artifacts of a software system. It provides a set of graphical notations for representing the structure, behavior, and interactions of software systems. See more at https://www.omg.org/spec/UML/2.5.1/About-UML/

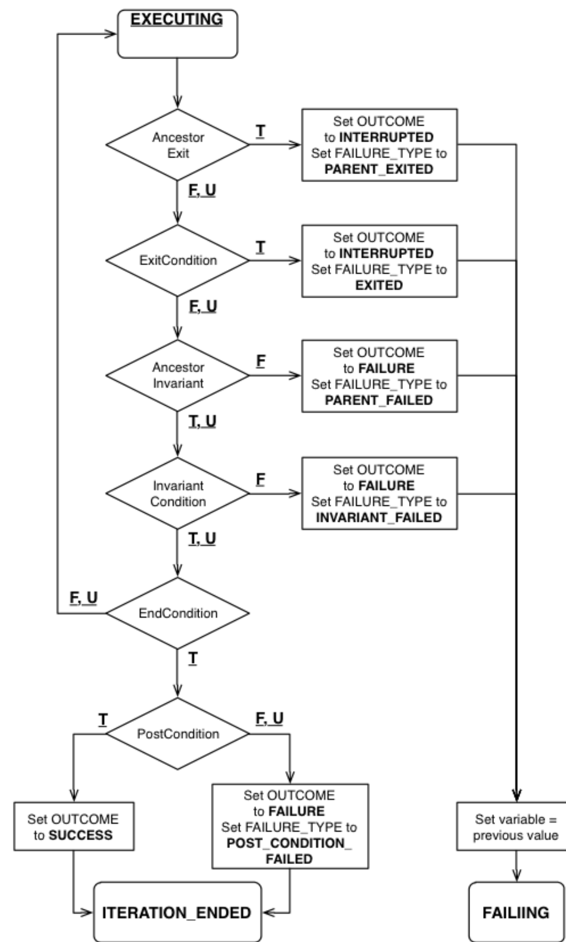


Figure 3.2: Node state diagram of assignment nodes in state *executing*

Every node state diagram is represented in a function that performs the state transition in PLEXIL5. State transition function only act on active nodes. Nodes are deactivated inside a given microstep when they perform one state transition so that no node transitions twice in a microstep.

3.1.5. The micro relation

The state transition operation `micro` signals the start of a microstep, that is, the simulated parallel execution of the atomic relation on the nodes of the system. The system then tries to transition nodes in the following order:

1. Inactive nodes
2. Waiting nodes
3. Executing nodes
4. Failing nodes
5. Finishing nodes
6. Nodes in state Iteration ended
7. Finished nodes

3.1.6. Launching a macrostep

A macrostep is initiated by the interpreter at the beginning of the execution of the plan or when the system has achieved quiescence. It then uses the `generator` cell to generate

the next input from the environment, and subsequently updates the generator. The input is used to update the `environment` cell for lookups, the `interface` cell for command handles and the `memory` cell to update local variables according to command results.

3.2 Architecture of the *plexil2maude* tool

The *plexil2maude* tool has already been introduced in this report. It is a tool that translates PLEXIL plans and scripts into their Maude equivalent under the PLEXIL5 semantics. It has been developed in Haskell. Since compiled PLEXIL plans and scripts follow an XML specification, the tool uses the `xml-conduit` package[3], which allows the traversal of an XML tree by using the concept of a *cursor*, which points to an element of the tree and by which one is able to obtain its information. Moreover, the tool uses the `pretty` package[4] that provides a set of functions that allow to write out texts in a consistent form, also known as *prettyprinting*. It also uses the `text` package[5], which adds support for Unicode text.

To build the tool, the Cabal framework is used. Cabal is a build system and package manager for Haskell [18]. It provides a way to build, test and manage Haskell packages, as well as manage dependencies between packages. Cabal reads from a file with extension .cabal, where executable commands, test-suites and dependencies of the project are specified. To build the tools, the following command is used from the root directory of the *plexil2maude* project — namely `plexil2maude`:

```
cabal v2-install --overwrite-policy=always
```
[6]

This command tells Cabal to build any specified packages and symbolically links their executable files into the default install directory `~/.cabal/bin`. Once made sure that the install directory is in the PATH, one can execute the `plx2maude` and `psx2maude` from any directory.

What follows is a detailed overview of the architecture of the tool. For reasons of inherited legacy code, *plexil2maude* is divided into two parts: *plx2maude*, which handles the translation of PLEXIL plans; and *psx2maude*, which handles the translation of PLEXIL input scripts.

3.2.1. Architecture of *plx2maude*

The tool consists of a `Main.hs` file that launches a `Parser.hs` file that contains all the mappings for XML elements that compose a compiled PLEXIL plan. The main file firstly reads the `.plx` plan and takes a cursor pointing to the root node of the XML hierarchy. It then starts the translation process by calling a function `elementVisitor` in the parser file. This function analyzes the type of node it is currently processing by obtaining its tag and applies the corresponding translation by calling auxiliary functions designed to process each element separately. If the PLEXIL plan XML node requires it, the function is able to be called recursively in order to process the children elements.

The first XML node that the translator finds is the root node, called `PlexilPlan`, which the translator processes creating the header of the Maude plan, by defining the plan module and creating the structure that will contain the root node of the plan. Then, the root

[3] See https://hackage.haskell.org/package/xml-conduit-1.9.1.2/docs/Text-XML.html for details
[4] https://hackage.haskell.org/package/pretty-1.1.3.6/docs/Text-PrettyPrint.html
[5] See https://hackage.haskell.org/package/text-2.0.2/docs/Data-Text.html for details
[6] See https://cabal.readthedocs.io/en/3.4/cabal-commands.html for details on the v2-install command

node contents are processed and the `elementVisitor` function is called again if the root node contains children nodes.

The parsing functions are defined according to the PLEXIL plan XML Schema. This schema defines every element that can appear in the XML tree with its attributes and sub-elements.

As evidenced by the lack of an internal data representation for the elements of the XML tree, the architecture of *plx2maude* is rather simple. This is a result of the legacy code inherited from previous versions of the tool, and there are plans to refactor the code into one that accounts for an internal representation in order to ease future development.

3.2.2. Architecture of *psx2maude*

In contrast to *plx2maude*, the *psx2maude* tool has a slightly more complex architecture, by dividing the translation process into two parts: one that parses the XML tree into an Abstract Syntax Tree (AST)[7] and another that *prettyprints* the AST into the Maude code. An AST is a data structure that represents source code in the form of a tree, containing only the information related to each construct that occurs in the code. The AST of a PLEXIL script starts from the root element with the `PLEXILScript` tag. This node has two children: the initial state and the script itself, as explained in Section 2.3).

In terms of the structure of the *psx2maude* tool, the parsing functions and the AST types definition is held in one Haskell file, and the *prettyprinting* functions are defined in another. The execution starts with a `Main.hs` module that reads the `.pst` file and executes the parsing functions to obtain the internal AST data structure in the module `PLEXILScript.hs`. Then, it executes the *prettyprint* functions in the module `PrettyPrint.hs` to obtain the Maude code. The *prettyprint* functions first create the header for the Maude script by defining the script module and, additionally, define the input generator that holds the external inputs defined in the script. Then, they may traverse the AST and append to the file the correct translated Maude code.

The parsing of the XML tree into the AST is performed using a technique known as pickling [19]. This technique provides a way to serialize and deserialize data structures by composing functions that are able to transform from XML to the internal data structure and vice versa. In this way, an XML element made up of sub elements can be processed independently.

In reality, the way that the *plx2maude* parser takes the XML tree and converts it into Maude code is very similar to how *psx2maude* converts to AST and then to Maude code. However, the latter does it in a more structured manner and, thus, its code becomes more human-readable and, by extension, more maintainable and extensible. As mentioned, future plans for the *plexil2maude* tool include a refactoring of *plx2maude* to separate parsing from *prettyprinting*.

[7] An abstract syntax tree is a hierarchical representation of the structure of a program's source code. It captures the syntactic relationships between different elements of the code, such as expressions, statements, and declarations. See more in https://courses.cs.washington.edu/courses/cse401/08wi/lecture/AST.pdf

CHAPTER 4
Methodology

A development methodology that utilizes GitHub[1] has been implemented to streamline the software development process and enhance task management. The PLEXIL5 project has had different contributors working under this methodology, coordinated by Marco A. Feliu. The methodology follows a set of steps, including:

1. Creating an issue (a task)
2. Assigning the task
3. Creating a pull request
4. Reviewing the changes
5. Merging the changes into the development branch

This approach has proved to be highly beneficial and has positively impacted the project in several ways.

The first step in the methodology is creating an issue. GitHub provides a convenient platform for creating and tracking issues, allowing contributors to document and discuss specific tasks, bugs, or features. By creating issues, the project's requirements and goals can be clearly defined. Issues centralise the information of the project, enabling a better organization between project contributors.

Once an issue is created, contributors can arbitrarily assign it to themselves to implement. This step promotes individual ownership and accountability. By allowing contributors to choose the issues they want to work on, the methodology empowers them to contribute to the project based on their expertise and interests. This self-assignment approach increases motivation and engagement among contributors, resulting in a more productive and efficient development process.

After completing the implementation, the next step is to create a pull request (PR). Pull requests are a formal petition for changes implemented by a contributor to be introduced to the main code of the project, requiring another contributor to review the changes. By creating a PR, the changes are isolated, and the review process can begin. This ensures that all code changes are thoroughly examined before they are merged into the main development branch [20]. Pull requests serve as a mechanism for collaboration, knowledge sharing, and maintaining code quality.

The review process plays a crucial role in this methodology. It involves a designated reviewer who carefully examines the changes made in the PR, providing feedback, suggesting improvements, and ensuring adherence to coding standards. Reviews enhance

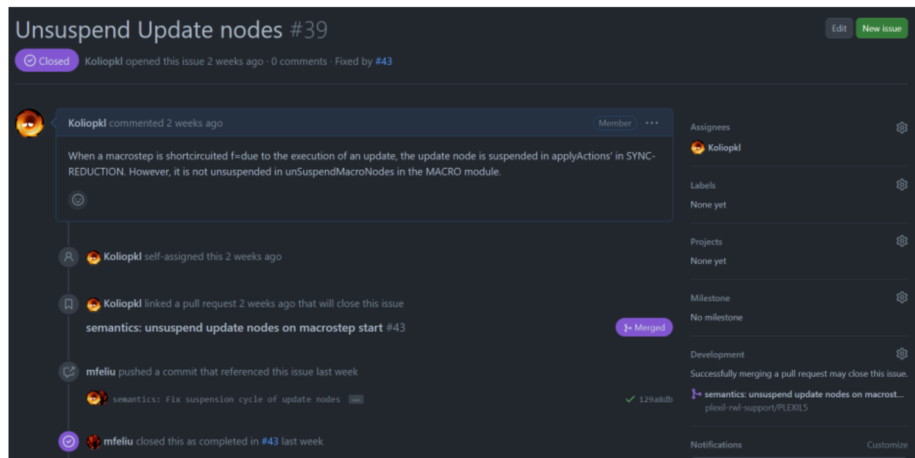
[1] GitHub is a web-based platform that allows developers to store, manage, and collaborate on code repositories. It provides version control tools, issue tracking, and integration with popular development workflows. GitHub facilitates open-source projects and serves as a central hub for developers to share and contribute to software projects.

the overall quality of the codebase and help catch potential bugs or issues early on. Given that the project has been overseen by Marco A. Feliu, he has served as the reviewer of the PRs that have been opened during the duration of this work. By leveraging his experience in the field, he has been able to pinpoint any issues unaccounted for. His expertise has proven vital for the development of the project.

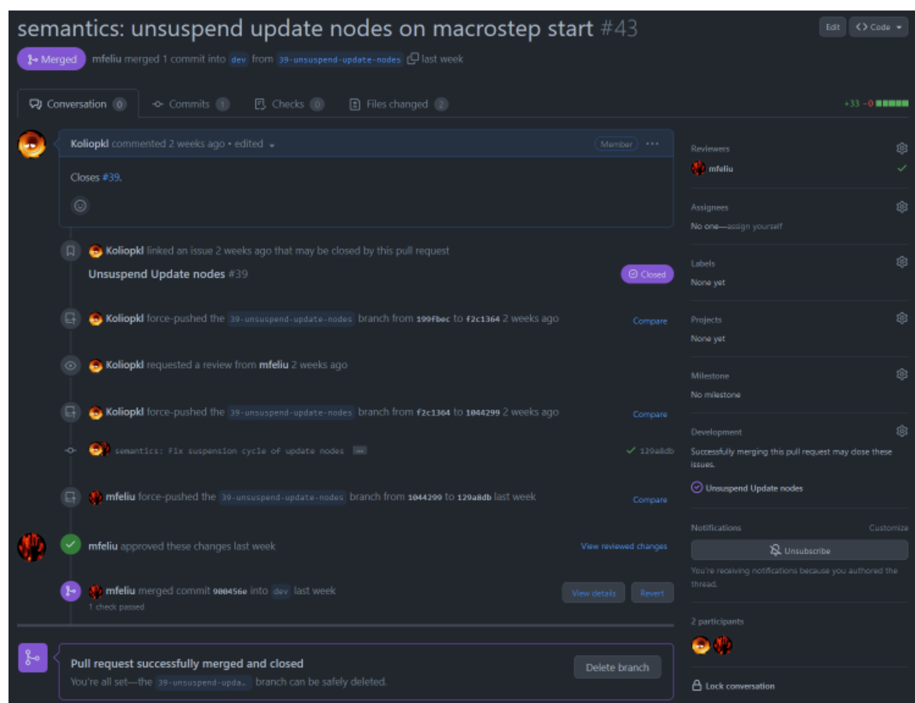
Once the changes are approved, they are merged into the main development branch. This approach minimizes conflicts and ensures that the main branch always contains functional and reliable code, while allowing contributors to independently work on new features.

The implementation of this GitHub-based methodology has significantly streamlined the development process in the project. Furthermore, the PR and review process has been instrumental for maintaining code quality and catching potential issues early on. The involvement of Marco A. Feliu as a reviewer has ensured that changes are thoroughly evaluated, and feedback is provided, leading to improved code quality and overall project success.

Figure 4.1 shows an example of the life cycle of a feature in the PLEXIL5 project. Figure 4.1a shows an GitHub issue. It was opened with a description and self-assigned by a contributor. When an issue is opened, it may be assigned to a development branch where the changes occur. A branch was opened and it can be seen in the *Development* section, and is named after the issue, e.g., `39-unsuspend-update-nodes`. When the contributor makes changes, they may open a linked pull request to review the changes. The PR is shown in Figure 4.1b, where the PR was firstly linked to the issue, followed by the implementation via commits. Then, it was assigned to a reviewer, Marco A. Feliu, who approved the changes and, finally, merged the branch `39-unsuspend-update-nodes` to the main development branch `dev`. The issue, linked to the PR, is successfully closed with the resolution of the linked PR.



(a) Issue that informs of the feature to implement



(b) Pull request that resolves the issue

Figure 4.1: Life cycle of a feature under development in the PLEXIL5 project

CHAPTER 5
Workflow

5.1 A test-driven approach

The development of the contributions described in this report have been guided by the technique of Test Driven Development (TDD) [21]. TDD advocates for a work ethic where the minimal unit of work is a test for a given piece of code, where the success of the test is the foremost priority. When tackling a new requirement, TDD indicates that before writing a single line of code, a test for an ideal version of a piece of code must be written. The test will assume an ideal code interface and an ideal code behaviour. The test *must* fail before the functionality is implemented. This technique allows tackling an herculean task by dividing it into small tasks that can be tested independently.

TDD builds up a suite of unit tests that grows with every functionality that is implemented, where these tests are very often run, to ensure that no change breaks any of the code that was previously written. Unit tests are pieces of code that validate a particular behaviour within the code. Within TDD, *the tests written before implementing the function are unit tests*.

TDD can be implemented in development by following a simple set of steps, referred to as the "Red-Green-Refactor" cycle [22]:

1. Red — Write a failing test: Write a test that fails. This test should be focused on a specific piece of functionality to be implemented.
2. Green — Write the simplest code to make the test pass: Write just enough code to make the failing test pass. This code should be the simplest possible solution that satisfies the test.
3. Refactor — Refactor the code: Improve the code without changing its behaviour. This may involve simplifying the code, removing duplication, or improving its design.
4. Repeat the cycle: Once the code has been refactored, the cycle can be repeated by writing a new failing test and then writing the code to make it pass.

This cycle encourages to focus on writing tests first, which helps to ensure that the code works as intended and is maintainable over time. The cycle also encourages continuous refactoring of the code, which helps to keep the code clean and easy to maintain.

For example, for a system that works with units of length that must:

a. Allow for the addition of two lengths
b. Allow for the conversion from miles to kilometers

This brief explanation assumes familiarity with Object-Oriented Programming. A programmer would:

1. Select one of the use cases, e.g. a.
2. Create a test that assumes that there is a function that works ideally in adding two lengths. For example they can assume that there is a class `Length` with a field `value` and a field `unit`. They can then assume that a method with header `addition(Length a, Length b)` exists and that an addition of an object of class `Length` with `value = 1` and `unit = 'km'` with an object of class `Length` with `value = 3` and `unit = 'km'` will lead to a resulting object of class `Length` with `value = 4` and `unit = 'km'`. They write up a test with these conditions.
3. Run the test and verify that it does not compile.
4. Implement the interface of the class `Length` and the method `addition`.
5. Run the test and verify that it fails. Failure of a test is a success because the interface of the functionality under test is correct.
6. Implement the function `addition` for the test to pass. This can be anywhere from the fully implemented function to a trivial function that returns exactly what the test expects.
7. Run the test and verify that it passes. A passing test without context does not imply that a function is correct. The programmer must now refer to the requirements of the system and come up with new test cases that assume that everything works ideally.
8. Modify the function `addition` to cover all test cases. Every time a modification is performed, the test suite must be run to verify its behaviour.

In this case, the programmer will realize that testing for `addition` to cover cases where lengths in kilometers and miles are added, they will need to implement the conversion use case b.. To do this, they will follow the same approach as for the multiplication functionality, creating basic tests and adding more test cases to cover the requirements. When b. is implemented, they can continue with a. to cover cases with mixed units. The main benefit of this approach is that it places the use cases of a system at the center of development, where the programmer approaches a problem from the top down: what the system should do first and then how it should do it.

TDD has been applied to PLEXIL5 in both the extensions to the plexil2maude tool as well as the executable Maude semantics. Each section of PLEXIL5 has its own set of tests that have grown organically as new functionality has been implemented. A discussion on the organisation of each section follows.

5.1.1. TDD in plexil2maude

The *plexil2maude* tool has been written in Haskell. It uses the framework Tasty to provide the required testing functionality, with support for unit testing via the package Tasty-HUnit. HUnit is a testing framework based on the xUnit family. Testing frameworks based on the xUnit family are based on a common architecture that includes a test runner, test suite, and test case classes [23]. Tasty-HUnit is also based on this architecture, where:

- `defaultMain` is the function used to run the test suite and generate the test results (the test runner).
- `testGroup` is the function used to create a test suite. It takes a name for the test suite and a list of test cases, which can also be other suites created with `testGroup`.
- `testCase` is the function used to define a test case. It takes a name for the test case and an assertion function that performs the actual test.

```
Tests
  PSX2Maude tests
    Pretty Printer
      Command
        Command ...(trimmed for space):
'                                                    FAIL
          tests/PSX2MaudeTests.hs:123:
          is not pretty printed as
          expected: commandResult('ac4,nilarg,array(val(1,1) # val(2,2)
               # val(3,3)))
           but got: commandResult('ac4,nilarg,array(val(1.1) # val(2.2)
               # val(3.3)))
          Use -p '/Command {cmdName = "ac4", cmdParams = [], cmdResult
              = Result {unResult = TypedValue {unTypedValue =
              TVRealArray [1.1,2.2,3.3]}}, cmdType = PXRealArray}/' to
              rerun this test only.

1 out of 192 tests failed (0.17s)
```

(a) Unit tests fail in *plexil2maude*

```
All 192 tests passed (0.03s)
```

(b) All unit tests pass in *plexil2maude*

Figure 5.1: Testing in *plexil2maude*

The definition of test suites and test cases create a tree-like structure for organizing test cases, conveniently reflected in the type `TestTree`.

In *plexil2maude*, a set of tester functions are defined that take a function with a given type signature and a tuple of the form `(test_name, input_string, expected_output_string)` and return a test case that, under the provided name, asserts that the expected output matches the produced output generated applying the function to the input. Different tester functions are defined to account for the different type signatures of the functions to be tested.

To run the test suite, the following command is used using Cabal as explained in Section 3.2:

```
cabal new-run test -- --hide-successes --ansi-tricks=false
```

This command tells Cabal to run the test-suite called "test", with options specified after the '`--`'. The `--hide-successes` option tells Tasty to only show the results of failed tests, while the `--ansi-tricks=false` option disables some ANSI escape code tricks that Tasty uses to make its output more colorful. The output of the command is the set of tests that have failed, and provides their expected and actual outputs. If all tests pass, the output is the number of tests passed. This behaviour can be seen in Figure 5.1, where a unit test fails because a floating point number is parsed with an incorrect decimal separator. This error is fabricated for the purposes of this explanation and resolving the issue leads to correct execution of the tests.

5.1.2. TDD in the PLEXIL5 semantics

Unit testing

Maude does not have any testing framework based on the xUnit family as Haskell has. Therefore, a script must be used to run the tests. Despite the lack of a testing framework, the tests are organized in a similar manner to xUnit-like frameworks:

- The concept of a test runner is covered by a set of Maude modules headed by the module `TEST-RUNNER` in the file `test-runner.maude`, which defines the functions that the test cases will be able to use and specifies how the test cases are reduced. The test runner operates by constructing a test suite using the syntax it defines, running each test, and reporting the results.
- The test-cases are defined according to the syntax specified by the `TEST-RUNNER` module. As in xUnit-like frameworks, test cases take a name, an input and an expected output.
- Test suites are built using an operator '`_+_`' that aggregates test cases.

Tests are run by reducing a Maude term that aggregates all test cases using the semantics specified in the `TEST-RUNNER` module. This behaviour is specified in a file named `suite.maude` which is abstracted by a file named `runSuite.maude`. Therefore, to execute the test suite, the contents of `runSuite.maude` must be executed.

To do so, a bash script `runTests.sh` has been defined that runs the file from the command line and refreshes everytime a `.maude` file changes to account for changes in the code or in the test base. That way, a developer can work on the semantics and check the status of the test suite in a terminal running `runTests.sh`.

The output of the script is, similarly to *plexil2maude*, the set of tests that have failed, also providing the expected and actual outputs. When all tests pass, an output void of errors is shown.

Acceptance testing

As the other half of the PLEXIL5 testing coin, there is acceptance testing. While unit testing allows testing for individual functions, acceptance testing is a form of testing that allows for testing of the whole system all together[24]. In a traditional software project, acceptance tests serve to assure the customer of the product that the system they receive does what is required of it. In some sense, acceptance tests mark the end goal of a project.

In line with this way of thinking, PLEXIL5 implements a set of acceptance tests. However, they differ from the traditional definition of an acceptance test. Since there is no customer to ship *the product* to, and the requirements of the semantics are that of the official PLEXIL executive, the acceptance test suite is composed of a set of tests that employ all parts of the system — that is, the translation of plans and scripts using the *plexil2maude* tool, the execution of the tests using both the PLEXIL Test Executive and the PLEXIL5 semantics and the comparison of their outputs using the *plexilog* tool.

These tests are defined under the `benchmark` directory. They are composed of a single plan and a set of 12 scripts that make the plan behave differently. These tests are executed using a custom bash script `runAT.sh`. The output of this script is whether the test plan differs in execution from the PLEXIL Test Executive.

The tests are used as acceptance test by being passed somewhat often when some feature is implemented. Since they use features that were already implemented in the semantics, they are a good indicator that the system behaves as expected.

```
Boolean Lookup flag;

boolean1:
{
    PreCondition LookupNow(flag);
}
```

Figure 5.2: boolean1.ple PLEXIL plan

5.2 Use of PLEXIL regression tests as PLEXIL5 acceptance tests

The development of new features of PLEXIL5 translates into adding support for PLEXIL functionalities that were not considered in the current PLEXIL5 version. The functionalities that are considered to add support for are part of the official PLEXIL Executive regression test suite.

The concept of regression testing has been introduced in Section 1.1. As a reminder, the regression test suite is provided with every PLEXIL distribution. The PLEXIL functionalities they test for are the target functionalities that PLEXIL5 must correctly implement in order to consider that it is semantically complete with respect to the official PLEXIL Executive.

While during the development of PLEXIL the regression tests serve its purpose as regression testing, in PLEXIL5 they take the role of acceptance tests. That is, the functionalities they test for become requirements for PLEXIL5, where acceptance is defined as equivalent state transitions with the same tests run with the PLEXIL Test Executive.

Figure 5.2 shows a test plan from the regression test suite as an example to illustrate the equivalence between the regression test suite and the requirements of PLEXIL5. The plan is called `boolean1` and it:

1. Declares a lookup for an environment variable called `flag` of type `Boolean`.
2. Declares an empty node with name `boolean1` with a user-specified precondition that is equal to performing the lookup on `flag`. If the variable `flag` in the environment is true, the node will execute. Otherwise, it will not.

Although an apparently simple plan, `boolean1` defines various requirements for PLEXIL5. According to `boolean1`, PLEXIL5 must support:

- The declaration of external lookup variables for the plan.
- The use of lookup expressions to query the environment.
- The type `Boolean`
- Nodes of type Empty
- User defined node preconditions
- The execution of lookups to the environment

These are a number of requirements that arise from a simple plan. However, most functionalities in plans are shared or related – such as empty nodes, the type `Boolean` and lookup expressions –, which means that implementing support for them to satisfy all the requirements of one plan will satisfy the same requirement for all subsequent plans. The case is the same for scripts, although they do not overlap between plans and scripts. That is, the set of all the requirements of all the plans is different from the set of all the requirements of all the scripts. This is why the *plexil2maude* tool is divided between *plx2maude* for plans and *psx2maude* for scripts.

```
<PLEXILScript>
  <Script/>
</PLEXILScript>
```

(a) empty.psx

```
mod INPUT is
  protecting PLEXILITE .
  op input : ->
      ExternalInputGenerator .
  eq input = sequenceGenerator(
    noExternalInputs
    #
    nilEInputsList
  ) .
endm
```

(b) empty.maude

Figure 5.3: Empty/trivial script to use with plans that do not require one

To simplify the amount of requirements, the requirements are not listed as the individual functionalities of each regression test, but as the regression tests themselves. Thus, in development, one does not set out to implement, following the example set by Figure 5.2, *The declaration of lookup variables for the plan* by itself. Instead, they set out to implement all the required functionalities to support the `boolean1` test, both in the plan and its corresponding script.

The regression test plans may be accompanied by a script with the same name that guides the execution of the plan. However, this is not always the case. For simpler plans, which do not require interaction with the external environment (defined in scripts), an empty script is used. It is a trivial script. In the PLEXIL Test Executive, the `plexiltest` command has an input parameter -p for the plan and an optional parameter -s for the script. This last parameter is optional because of the aforementioned cases where a plan does not need a specific script to guide its execution. In those cases, the parameter can be left empty and the PLEXIL Text Executive will run the plan using an empty, trivial script. This behaviour is replicated in PLEXIL5 by defaulting to an empty script `empty.maude` that is translated from the empty script `empty.psx`. The content of this script can be observed in Figure 5.3. The compiled PLEXIL script in Figure 5.3a declares a self-closing tag `<Script/>` that denotes that there are no external inputs, which is reflected in the translated Maude script in Figure 5.3b.

5.3 Development workflow

This section outlines how new PLEXIL5 features are developed. Section 5.2 explains how all new features stem from trying to make tests from the PLEXIL regression test suite behave identically when executed with PLEXIL5 as they do when executed using the PLEXIL Test Executive. Therefore, the first step is to execute a test in both interpreters and compare the results, using the `runRegression` script.

5.3.1. The runRegression script

`runRegression` is a script that has been developed with the goal of automating the process of comparing the state transition trace of a plan with a script executed both in the official PLEXIL Test Executive and the PLEXIL5 Interpreter. To do this, it makes use of the tools *plexil2maude* to translate the plans and *plexilog* to compare the results of the trace. It has several flags, but most importantly for these purposes it has a flag `-t` that allows the user to specify what particular test to compare.

The output of the script is a list of the differences in the trace of both executions of a plan under a script, or `EQ` if they are equal. When there is a parsing error, a message that reads 'bad token' is shown.

`runRegression` has been developed in Python[1]. Python is a programming language commonly used for scripting due to its readability, versatility and easiness to develop in. It is not compiled, which allows for rapid development and prototyping, as scripts can be written and executed directly. It provides a wide range of modules and libraries to make use of pre-existing code to enhance scripts. It is also multi platform, so it can be executed in any system with a Python distribution.

The `runRegression` script was discussed to be developed in Bash[2]. Bash is a command language and shell for Unix-like operating systems, providing a scripting environment to automate tasks, execute commands, and interact with the operating system through a command-line interface. Bash was considered due to its integration with the underlying shell environment of Unix-like systems, leveraging the power of terminal commands to include in scripts. It is also portable and very efficient, and readily available in all Unix-like systems by default. However, Python offers much faster development due to its readability. Bash, on the other hand, is very succinct and technical. Python is also much easier to debug and offers great flexibility with its libraries. It is also able to use terminal commands like those in *plexil2maude* and *plexilog*. Therefore, Python was the optimal choice to develop a script to automate PLEXIL execution logs.

5.3.2. Parsing errors

When the `runRegression` script finds a parsing error and shows the message 'bad token', what follows is the name of the XML element that the PLEXIL5 interpreter did not recognize. Often, these kinds of errors arise when the parsed plan or script code is not up to specification with the PLEXIL5 semantics. These may include wrong function names, missing arguments, etc. However, the most common occurrence during the development of this work was that of unsupported elements both in the *plexil2maude* tool and the PLEXIL5 semantics. Detecting these missing elements is straight-forward because the default behaviour for unsupported elements in *plexil2maude* is to print the element literally. Therefore, the missing element can be identified and its implementation both in *plexil2maude* and the PLEXIL5 semantics can commence.

To exemplify this behaviour, the following extract shows how an unsupported node with tag `NEBoolean` would be translated using `plexil2maude`. The `NEBoolean` tag has two numerical child elements, which are supported and represented using the `const` operator. The `NEBoolean` tag is printed literally and without the constructs to represent the equivalent element in the PLEXIL5 semantics.

```
NEBooleanconst(val(true)),const(val(false))
```

5.3.3. Debugging a plan

When differences in the trace of execution are found, the next step is to build a runner to debug the code. The debug runner imports the translated plan and script and starts the Maude rewrite engine with the root node of the Maude plan. In reality, the debug runner is just a copy of the runner that the `runRegression` script generates for the input plan and

[1] https://www.python.org/doc/
[2] https://www.gnu.org/software/bash/manual/bash.html

script. Note that what the debug runner does is the same as the runner, but it is better not to modify the latter since it is overwritten with every `runRegression` execution.

The difference between the two runners is that, in the debug runner, Maude commands are used to closely follow and examine the execution traces. Mainly, the `trace` and `break` commands.[3]

- The `trace` command provides detailed information of every rewriting step. It is enabled with `set trace on` To provide the information only when some operators are rewritten, `trace select <operator>` can be used. This command is especially useful to analyze the state of the modeled system, to examine nodes, interfaces, the memory, etc.
- The `break` command sets break points along the code. It is enabled using `set break on`. To select the operator to break on, the `break select <operator>` command is used. Break points are especially useful when the operator that may be incorrectly defined is known. When a break point is reached, the state that reached the break point can be consulted with the command `where` The command `step` is used to proceed with the execution step by step. The execution can be resumed using the command `resume`.

As mentioned, the debugging process involves analyzing the execution trace in detail to detect inconsistencies in the PLEXIL5 semantics. This often proves to be an arduous task and consumes most of the time in developing the PLEXIL5 semantics. However, the test-driven approach explained in Section 5.1 and used during the project means that the number of instances instances where the debugging process was needed was reduced. Even then, cases where it was needed arose from insufficient test cases.

This situation is suboptimal. However, it is enabled by the fact that the semantics of PLEXIL are not clear cut, and the documentation does not inquire into most procedures with enough detail. These instances benefited from the experience that the tutor under who this project was developed, Marco A. Feliu, had working with PLEXIL, and was able to explain them in detail.

[3] https://maude.cs.illinois.edu/w/images/e/ee/Maude-3.0-manual.pdf

CHAPTER 6

Extending the semantics of PLEXIL5

The following section examines the contributions made to the semantics of PLEXIL5. This will involve an in-depth discussion of the innovative approaches and insights that have played a pivotal role in extending the behaviour, expressive power, and reliability of the tool.

The project has seen contributions in five areas. First and most important has been the implementation of update nodes, followed by the completion of the semantics of command nodes. These two types of nodes are used throughout the regression test suite. Their accurate implementation aids in executing tests that uncover new required features. Additionally, macrostep short-circuiting has been incorporated to align with the execution semantics of the PLEXIL Test Executive. Furthermore, input generation has been revamped to consider nodes that are still running when a plan exhausts external inputs. Lastly, this section also encompasses other specific features that, though smaller in scale, were implemented to ensure correct execution of regression tests.

6.1 Adding Update Nodes to PLEXIL5

The introduction of update nodes stands out as a significant contribution to the semantics of PLEXIL5, particularly due to its broad impact on the functional scope of the PLEXIL5 project. Previous versions of PLEXIL5 did not incorporate this node type, possibly because it was beyond the project's initial scope or because the concept of update nodes did not exist at the time of development. Determining the exact reason is challenging since neither the PLEXIL documentation[1] nor the project's changelog[2] specifies the timing of update nodes' introduction. Nevertheless, the documentation does currently offer valuable insights that have guided the implementation of this feature.

Update nodes have been explained in Section 2.3. As a reminder, update nodes modify information of the interface of the system. They have a `pairs` attribute that holds associations of variables of the interface with their corresponding values to be updated and a boolean `acknowledgement` attribute that indicates whether an update of the interface has completed.

The implementation of the update node type has the format followed by other node types in the sense that it extends the node definition that all other node types have. A constructor `update` has been defined such that it accepts four arguments: an identifier for

[1]https://plexil-group.github.io/plexil_docs/PLEXILLanguage/PLEXILReference.html
[2]https://github.com/plexil-group/plexil/blob/develop/ReleaseNotes.md

the node, the declaration of local variables of the node, a set of node attributes and the list of pairs of variable-value to update the interface. This constructor creates an element of sort `PlexilUpdate`, which is a subsort of `Plexil`, a general type to represent PLEXIL nodes.

```
op update : Identifier LocalDecls AttributeSet List{Pair} -> PlexilUpdate .
```

The sort `Pair` has also been introduced as an association of a variable identifier and an expression that represents its associated value. Expressions are a super-type that all types of expressions inherit, making use of polymorphism to improve the modularity, readability and extensibility of the code by writing functions that apply to any kind of expression.

```
op pair : Qid Expression -> Pair [ctor] .
```

Along with the update node type, an update *acknowledgement* `updateAck` has been defined according to Section 2.3. It is an external input that carries an `Ack` value associated to a node identifier.

```
op updateAck : Identifier Ack -> UpdateInput .
```

The sort `Ack` is an abstraction of a boolean value. This decision was made – instead of using a simple boolean value – to provide a self-descriptive representation of update acknowledgements, as well as for the sake of readability.

```
subsort Bool < Ack .
```

In the compilation step, the node is transformed into the internal object notation. It assigns the provided node identifier, assigns the type *update*, sets the `pairs` attribute with the provided list and sets the `acknowledgement` attribute as false for not completed. It also adds the default node conditions according to Table 2.1 and enriches them by conjunctively applying the given termination condition with the receipt of an update acknowledgment. as illustrated in the implicit end conditions Table 2.2.

This means that an update node translated as:

```
update(
    'anId,
    nilocdecl,
    endc: const(val(false)),
    pair('taskId,var('waypointId)) pair('taskId2,var('waypointId2))
  )
```

Would be compiled into:

```
< 'anId : update |
    status:     inactive,
    outcome:    none,
    active:     true,
    suspended:  false,
    pairs:      pair('taskId,var('waypointId)) pair('taskId2,var('waypointId2)),
    startc:     const(val(true)),
    skip:       const(val(false)),
    pre:        const(val(true)),
    inv:        const(val(true)),
    exitc:      const(val(false)),
    repeatc:    const(val(false)),
    post:       const(val(true)),
```

```
------------------------------UpdateTest.maude------------------------------

plexilog-diff: ParsingMaudeException "\"stdin\" (line 1, column 1):\nunexpected
    \"W\"\nexpecting white space or macro step"
```

Figure 6.1: Test UpdateTest fails to execute because update nodes are not implemented

```
------------------------------UpdateTest.maude------------------------------

EQ
```

Figure 6.2: Test UpdateTest executes correctly

```
    endc:       const(val(false)) and hasRcvAck?('anId . 'aRootId),
    ack:        false
  >
```

The `nilocdecl` constant is of sort `LocalDecls` and means that the node has no locally declared variables. Once the node is defined, the next step is to extend the semantics of the atomic relation to account for the introduction of the new nodes. Atomic relation functions as explained in Section 3.1.4 follow the node state transition diagrams that can be found in [14]. The full implementation and node diagrams are omitted due to their excessive length.

For update nodes, the only atomic relations that behave differently from other nodes are the transitions from executing and failing, linked to the necessity to check whether the update has been executed, either explicitly in the case of the state failing or as part of the end condition in the case of the state executing. To solve this problem, a function `hasRcvdAck?` is defined that checks the value of the `ack` attribute of the node, which is updated via the interface. When an update is executed – that is, the update node enters the state executing – a node in the interface *cell* is created that signals that the update is executing (a `update-on-execution` node). This node holds information about the update acknowledgment – the just mentioned `ack` attribute.

There are several test plans in the regression test suite that make use of the update node. As an example, there is a test named `UpdateTest` used to specifically test the behaviour of updates. Figure 6.1 shows how the plan did not parse prior to the implementation of the node, while Figure 6.2 shows how, after their implementation, it executes correctly.

6.2 Fixing the semantics of Command Nodes in PLEXIL5

The other most significant contribution to the PLEXIL5 semantics entails rectifying the semantics of the command node. The concept of a command node existed in the semantics. Yet, the semantics had not been fully implemented, similarly to update nodes, because their specification had changed with recent versions of PLEXIL.

The first issue with the command node semantics was related to the node state conditions, since the semantics did not reflect the special end condition that command nodes have, as noted in Table 2.2. Thus, the compilation of command nodes has been augmented through the enhancement of the end condition, adding logical disjunction with the helper function `cmdHandleIs?`, which checks whether the command handle of the node with the provided identifier is denied (CommmandDenied) or has failed (Com-

```
[nodes :
  < 'myNode : command | [...], status: executing, [...] >]
[interface :
  < 'myNode : command-on-execution | [...], aborted: false, [...] >]
```

Figure 6.3: A command node *myNode* enters execution and creates a command-on-execution node

mandFailed). With this change, the node will prematurely end under these conditions, even if the user-defined end condition is false.

As presented in Section 2.3, command nodes have command handles that, received from the environment, inform about the state of the command. Managing command handles is similar to how it is done in update nodes. When a command is executed, a node in the `interface` *cell* is created that signals that the command is executing (a `command-on-execution` node, Figure 6.3). This node holds information about the command handle. It is what the previously mentioned `cmdHandleIs?` function consults in order to obtain the command handle. However, command aborts are not considered in this scenario, and so a method to track them has been implemented. An attribute `aborted` has been introduced that holds a boolean value indicating whether the command has been aborted.

To be able to signal abortion, the script must support the provision of command aborts as external inputs to the plan. An operator `commandAbort` has been introduced for a script to be able to signal the abortion of a command. When a `commandAbort` is read from the script, the interface *cell* is modified by finding the `command-on-execution` node with the provided identifier and arguments and setting its `aborted` attribute to `true`. It is significant to remark that the `commandAbort` operator supports a parameter value intended for future-proofing this implementation in the case that more abort values are introduced. However, the current behaviour mimics the Test Executive by assuming that whenever a command abort is received for a certain command, the abort attribute must be set to true. That is, `commandAbort` is called as:

```
commandAbort(node_identifier, arguments, val(true))
```

This single behaviour implicitly takes care of both successful and failed command aborts, because only successfully aborted commands are signaled to the plan.

Command abortion is crucial in the atomic relation for command nodes because the state transitions from failing requires this information. It waits for the command to abort and then transitions the node. To check for successful command abortion, a function `hasAborted` is defined such that it queries the interface *cell* to find a `command-on-execution` node with the same identifier as the command node, returning its corresponding boolean value.

To test the implementation of command abortion, a new test has had to be written. This is because there is no test in the regression test suite that tests only for command abortion. While command aborts can be found throughout the test suite, they are part of very large plans that test for a lot of functionalities, some of which are not implemented yet. Therefore, a `commandabort1` test and script have been created to test the correctness of the command abortion functionality. Figure 6.4 shows a simple plan and accompanying script that executes a command, which is aborted. It also shows the Maude translation of said plan and script. Notice how the plan has a user-defined exit condition because according to the node state diagrams in [14], command abortion is checked when the command node transitions to failing. To force this exit condition, a lookup to the environment variable `time` is used. Since command aborts were not implemented, execution

```
Boolean Command cmd_name;

commandabort1:
{
  ExitCondition Lookup(time) ==
       1;
  cmd_name();
}
```

(a) Simple plan that tests for command abortion

```
mod commandabort1-PLAN is

 protecting PLEXILITE-PREDS .

op rootNode : -> Plexil .
eq rootNode =

   command(
     'commandabort1,
     nilocdecl,
     ((exitc: (_equ_(lookup('
          time,(nilarg)),const(
          val(1)))))),
     (('cmd_name) / (nilpar))) .

endm
```

(b) Plan translation to Maude

```
script {
     state time () =  1: real;
     command-abort cmd_name () =
          1 : bool;
}
```

(c) Simple script that tests for command abortion

```
mod INPUT is
   protecting PLEXILITE .
   op input : ->
        ExternalInputGenerator .
   eq input = sequenceGenerator(
     noExternalInputs
     #
     commandAck('cmd_name,nilarg
          ,CommandSentToSystem)
     commandAbort('cmd_name,
          nilarg,val(true))
   ) .
endm
```

(d) Script translation to Maude

Figure 6.4: Simple plan and script that test for command abortion

before applying the changes leads to the result in Figure 6.5. Applying the changes leads to Figure 6.6, where the test executes correctly.

6.3 Short-circuiting the macro relation

Another important contribution to the semantics involves the execution semantics of a macrostep. It has been observed that the PLEXIL Test Executive deviates from the nominal execution stated in the official PLEXIL documentation. According to the documentation, the nominal execution model requires the system to reach a state of quiescence within a macrostep before processing the next input. However, the PLEXIL Test Executive follows a different approach. In the PLEXIL Test Executive, when a node of types as-

```
------------------------------commandabort1.maude------------------------------

plexilog-diff: ParsingMaudeException "\"stdin\" (line 1, column 1):\nunexpected
     \'W\"\nexpecting white space or macro step"
```

Figure 6.5: Test commandabort1 does not execute correctly

```
------------------------------commandabort1.maude------------------------------
EQ
```

Figure 6.6: Test commandabort1 executes correctly

signment, command, or update enters execution, the corresponding action is performed immediately. However, the system adheres to the principle of performing actions at the end of the macrostep. To reconcile this discrepancy, it is inferred that the macrostep is short-circuited, allowing the actions to be executed immediately. Specifically, when an assignment, command, or update node transitions from waiting to executing, a short-circuit occurs in the middle of the quiescence cycle, resulting in a new macrostep. The PLEXIL Test Executive then proceeds to read the next input from the environment.

Regarding assignment nodes, any assignment is performed temporarily and can be undone if the assignment node fails. This approach ensures that the parallel atomic relations truly operate in parallel, which allows PLEXIL5 to mimic the synchronous nature of PLEXIL. In other words, other nodes that access the variable being assigned receive the latest value, even if the assignment node fails and the value needs to be reverted to its previous state. This mechanism, known as a *dirty read*, enables the plan to proceed synchronously, maintaining the time consistency of variable values. That is, any access to the variable from any node at the same point in time – in the same microstep – will lead to the same value.

Short-circuiting the macrostep involves updating the `macroacts` *cell* with the action specified by a command, assignment or update node when it enters execution. Actions are consequently defined as a sort with operators that encapsulate the behaviour required by the execution of assignment, command and update nodes:

- An action `runCommand` to execute command nodes
- An action `runUpdate` to execute update nodes
- A pair of actions `setMem` and `undoMem` to handle the possibilities that executing an assignment node offers. The former executes the assignment to ensure that all further accesses to the variable read the latest value, therefore ensuring effectively synchronous execution and leading to the aforementioned *dirty reads*. The `undoMem` action undoes the variable assignment in case the assignment node enters a failing state.

With these functions defined, nominal execution is followed in the atomic relation and, when one of the nodes enters state executing – in the node state transition from state waiting – the action is added to the `macroacts` *cell* and the node is suspended in the `microacts` *cell*. This suspension ensures that these nodes cannot transition anymore and force the microstep to finish. When the microstep finishes – first executing actions in the `microacts` *cell* – the `macroacts` *cell* is queried to check for macroactions. If any are found, the system is set to have achieved quiescence. This behaviour can be observed in the following function, where `Cfg` represents a system configuration that includes all the *cells*.

```
ceq [micro-relation-withMacroactions] :
  micro      | Cfg
=
  quiescence | Cfg
if hasMacroactions?(Cfg)
.
```

```
MACRO
    'UpdateTest from inactive to waiting
micro
    'UpdateTest from waiting to executing
micro
    'UpdateTest from executing to iterationEnded
micro
    'UpdateTest from iterationEnded to finished
micro
```

Figure 6.7: The macrostep is not short-circuited on update node execution

Once the system has reached quiescence, the following sequence of events unfolds:

1. Assignments are performed
2. Suspended assignment, command and update nodes are awaken
3. Commands are executed
4. Updates are executed

A new input is read and the plan execution proceeds normally. This behaviour can be seen in the following function, which executes when the system reaches quiescence, that is, when the current macrostep end:

```
eq [quiescence] :
  quiescence | Cfg
=
  macro |
    executeUpdates(
      executeCommands(
        unSuspendMacroNodes(
          performAssignments(
            Cfg
          ))))  .
```

The effects of the implementation of these short-circuits can be verified in plans such as `UpdateTest`, which has been previously mentioned. Figure 6.7 shows the execution of the `runRegression` script with `UpdateTest` without short-circuiting. It shows how the update node `'UpdateTest` enter execution and is not immediately suspended to stop the macrostep and execute the command, instead continuing the node's nominal execution. In this case, the actual execution of the update is not performed as the node enters execution, but it waits for the quiescence cycle to ends, which is not how the PLEXIL Test Executive works. Figure 6.8 shows how, after implementing the feature, the result is `EQ`, meaning that the state transitions are exactly the same for the PLEXIL Test Executive and the PLEXIL5 semantics. In this scenario, the macrostep is short-circuited right after the node enters execution and the update is performed at that point.

It must be noted that this behaviour is complementary to the generation of empty inputs explained in the following Section.

6.4 Extending the generation of inputs

The use of PLEXIL scripts means that the execution of any plan has a very concrete start and a very concrete end, driven by the succession of external inputs. When taking into ac-

```
MACRO
    'UpdateTest from inactive to waiting
 micro
    'UpdateTest from waiting to executing
 micro
MACRO
    'UpdateTest from executing to iterationEnded
 micro
    'UpdateTest from iterationEnded to finished
 micro
```

Figure 6.8: The macrostep is short-circuited on update node execution, which starts a new macrostep

count the short-circuiting of the quiescence cycle to leap one macrostep forward, there is a chance for nodes to be *active* and *executing* but not have achieved termination. To tackle these cases, the Test Executive appends empty inputs to the input stream to force these nodes to terminate. When two successive states of a PLEXIL plan are exactly identical, that is, no state transitions have arisen from the last macrostep, the plan is terminated.

This behaviour is not found in the PLEXIL documentation as of the writing of this report, which lead to the incorrect definition of the PLEXIL5 semantics in the past. However, it can be checked by analyzing the state transitions on a plan with any of the aforementioned node types and a script with a set amount of inputs.

Take the following example. Figure 6.9a shows the code for a plan that assigns the result of a lookup `look` to a local variable `myLook`. Figure 6.9b shows the accompanying script, which supplies a result for said lookup. Seeing as the script only contains one input, one could expect for the execution diagram to be conformed by only one macrostep. However, this theory is disproved by the transition logs of the PLEXIL Test Executive in Figure 6.10, which show two macrosteps, signified by two separate `==>Start cycle n`. Indeed, the first macrostep is short-circuited when the assignment node `myNode` enters state *executing*. This means that one extra empty input has been appended to the end of the input script to allow for `myNode` to eventually arrive to state *finished*.

The PLEXIL5 semantics did not account for the extra input appending behaviour because it had no way of analysing the current state of the system to reason whether any of the nodes are active but have not reached termination. It also did not include a step in the input generation process that was able to compare the current state of the system with the previous. These capabilities would allow the input generator to only add extra empty inputs when the state of the system had changed in some manner and some nodes still needed to terminate.

Figure 6.11 shows what the transitions log yielded with the past implementation. In total, only one macrostep was executed, corresponding with the only explicit external input in the script. The node `'myNode` is left at an unfinished state. This could be understood as a side-effect of short-circuiting the macrostep, because the provision of inputs has leapt forwards. Since `'myNode` is an assignment node, it has been suspended upon entering execution because the assignment needs to be executed immediately. This means that the macrostep ends and there are no inputs to consume: `'myNode` has been left executing and suspended. A closer inspection of the system state at the end of the plan shows this in the nodes *cell*, as shown in Figure 6.12.

However, what must result is the transition log in Figure 6.13, which shows how two macrosteps are executed, even if there is, still, only one external input. See in Figure 6.14 how the node does correctly terminate now.

```
Real Lookup look;

myNode:
{
  Real myLook;

  myLook = LookupNow(look);
}
```

(a) Example plan with an assignment node and a lookup

```
initial-state {
  state look () = 3.14 : real;
}

script {
}
```

(b) Example script that provides the result of a lookup

Figure 6.9: PLEXIL plan and script that demonstrate the appending of extra empty inputs to the script

```
[PlexilExec:cycle] ==>Start cycle 1
[PlexilExec:step][1:0] Check queue: myNode 0x55e8df836140
[PlexilExec:step][1:0] State change queue: myNode 0x55e8df836140
[PlexilExec:step][1:0:0] Transitioning node myNode 0x55e8df836140 from INACTIVE
     to WAITING
[PlexilExec:step][1:1] Check queue: myNode 0x55e8df836140
[PlexilExec:step][1:1] State change queue: myNode 0x55e8df836140
[PlexilExec:step][1:1:0] Transitioning node myNode 0x55e8df836140 from WAITING
    to EXECUTING
[PlexilExec:cycle] ==>End cycle 1
[PlexilExec:cycle] ==>Start cycle 2
[PlexilExec:step][2:0] Check queue: myNode 0x55e8df836140
[PlexilExec:step][2:0] State change queue: myNode 0x55e8df836140
[PlexilExec:step][2:0:0] Transitioning node myNode 0x55e8df836140 from
    EXECUTING to ITERATION_ENDED
[PlexilExec:step][2:1] Check queue: myNode 0x55e8df836140
[PlexilExec:step][2:1] State change queue: myNode 0x55e8df836140
[PlexilExec:step][2:1:0] Transitioning node myNode 0x55e8df836140 from
    ITERATION_ENDED to FINISHED
[PlexilExec:cycle] ==>End cycle 2
```

Figure 6.10: Log of the transitions of the example plan under the example script that demonstrates the appending of extra empty inputs to the script

```
MACRO
    'myNode from inactive to waiting
 micro
    'myNode from waiting to executing
 micro
```

Figure 6.11: Log of the transitions of the example plan under the example script translated to Maude, executed with the PLEXIL5 semantics before implementing the appending of extra empty external inputs

```
[nodes :
  < 'myNode : assignment | status: executing, suspended: true, [...] >]
```

Figure 6.12: The nodes *cell* when the plan terminates without appending an extra empty external input, leaving a node executing and suspended

```
MACRO
    'myNode from inactive to waiting
 micro
    'myNode from waiting to executing
 micro
MACRO
    'myNode from executing to iterationEnded
 micro
    'myNode from iterationEnded to finished
 micro
```

Figure 6.13: Log of the transitions of the example plan under the example script translated to Maude, executed with the PLEXIL5 semantics after implementing the appending of extra empty external inputs

This behaviour has been implemented via the concept of input sequence generators. Input sequence generators are a structure that holds a sequence of external inputs that are fed to the plan. Under nominal execution, every time a macrostep is started a new input is generated from the head of the sequence of inputs in the generator. The input generator is updated right after the choice of input and the interface is updated.

When the generator consumes its last input in the sequence, the update yields a generator with an empty sequence. However, when an input is to be generated from a generator with an empty sequence, it is only signaled that there are no inputs left when all the nodes in the plan have finished. This is done by matching a conditional equation in Maude with a condition that checks for all nodes termination. Another conditional equation will try to match, checking if the last and current configurations are the same. This was only possible after updating the interface of the input generator to include two configurations to compare. Before this, input generators were stateless, without knowledge of what state the system had been in.

Note that, to be able to compare two system states, some filters had to be passed beforehand, due to some cells that change with every step of the execution semantics – mainly the `interface-history` and `generator` *cells*.

6.5 Other contributions

The rest of the contributions to the PLEXIL5 semantics came in the form of implementing features that help tests in the regression suite have identical state transitions as under the PLEXIL Test Executive. As a reminder, the measure of completeness considered for

```
[nodes :
  < 'myNode : assignment | status: finished, suspended: false, [...] >]
```

Figure 6.14: The nodes *cell* when the plan terminates after appending an extra empty external input, correctly terminating the node that previously did not

the PLEXIL5 interpreter is the extent to which its execution semantics mirror those in the PLEXIL Test Executive.

For this purpose, the approach relies on a prior analysis of the features that are yet to be implemented. By cycling through the plans and scripts in the regression suite, a list of tasks was curated to use as a basis of all the implementation. The list contains the features that are yet to be implemented, and is accompanied by an annex of what tests are associated with each feature. In this manner, attention can be directed towards supporting tests that require less new features, so that the task of supporting more complex tests in the future can be lightened by shared features by other tests. Table 6.1 compiles the result of the analysis, relating a feature with the tests that make use of it.

Table 6.1: Table analyzing what features are left to implement into PLEXIL5

Feature	Tests that use it
Library node call	AssignmentMain Increment-test Increment-test2 lib1 lib2 libcall LibraryNode6 LibraryCallWithArray ... (Trimmed for space)
Node timepoint value	AncestorReferenceTest node-order-bug old-style-ref-test TestTimepoint TimepointVariableConstructionOrder
Max	maxTest
Min	minTest
Abs	TestAbsSqrt
Sqrt	TestAbsSqrt
Node state variable	AssignToParentInvariant contention3 DoubleInvariantAssignment GrandparentAccess isKnown1 old-style-ref-test SimpleDrive TestAbsSqrt TestNodeNameScope TestNodeNameScopeHack TestRepeatCondition
Concat	concat1 concat2 command2 command5
Table continues	

Table continuation	
Feature	**The tests that use it**
Tolerance	ChangeLookupTest SiteSurveyWithEOF TestTimepoint unknown_lookup
Round	RoundTest
Update	array8 ArrayInLoop SiteSurveyWithEOF UpdateLookupTest UpdateTest
LookupOnChange	array1 conjuncts lookup1 repeat5 repeat7 repeat8 SimpleDrive SiteSurveyWithEOF TestEndCondition unknown_lookup ... (Trimmed for space)
Delay	AtomicAssignment delay1 OnMessageFailureTest Resource4HvmRepeatCond closedloop-command-multipleAck SiteSurveyWithEOF

6.5.1. String concatenation

A feature common to most programming languages is the ability to concatenate strings into a longer string. PLEXIL also supports string concatenation via an operator `+`. Compiled plans and scripts include concatenation elements via a tag `Concat`, which contains at least two elements, up to `n` elements.

Some tests that implement the concatenation feature are tests `concat1` and `concat2`. Figure 6.15 shows how executing the `runRegression` script leads to an error because the feature is not implemented in the form of an unexpected character, because it is parsed incorrectly.

Fortunately, Maude does natively support string concatenation via an infix operator `_+_` that takes two strings as arguments and returns the concatenation of said arguments. Hence, the solution of the problem resides in defining a parser function that translates an element of form:

```
<Concat>
    <Element1>...</Element1>
    <Element2>...</Element2>
```

```
--------------------------------concat1.maude--------------------------------

plexilog-diff: ParsingMaudeException "\"stdin\" (line 1, column 1):\nunexpected
    \"W\"\nexpecting white space or macro step"

--------------------------------concat2.maude--------------------------------

plexilog-diff: ParsingMaudeException "\"stdin\" (line 1, column 1):\nunexpected
    \"W\"\nexpecting white space or macro step"
```

Figure 6.15: Tests concat1 and concat2 lead to errors because the feature is not supported

```
--------------------------------concat1.maude--------------------------------

EQ

--------------------------------concat2.maude--------------------------------

EQ
```

Figure 6.16: Tests concat1 and concat2 execute correctly

```
    ...
    <ElementN>...</ElementN>
</Concat>
```

Into:

```
(((parse Element1) + (parse Element2)) + ...) + (parse ElementN)
```

Implementing this function leads to Figure 6.16, where the above tests execute identically to the PLEXIL Test Executive.

6.5.2. Node *self* references

Nodes in plans can incorporate references to other nodes in the plan. These references can be to children of a node, siblings of a node and the node itself to, for example, set as a postcondition that all children of a node must have reached termination.

’self’ node references were not supported in PLEXIL5, leading to a plan like `skip1` to be unable to run in the semantics, as shown in Figure 6.17.

Implementing this node reference required first defining an adequate parsing function. ’self’ references are represented in a compiled plan as an element `<NodeRef dir='self' />`. In the semantics, the operator has been defined as an `Identifier` with no arguments, making use of polymorphism to incorporate it into the set of identifiers that can be used in a function to reference a node.

```
op self : -> Identifier .
```

```
--------------------------------skip1.maude--------------------------------

plexilog-diff: ParsingMaudeException "\"stdin\" (line 1, column 1):\nunexpected
    \"W\"\nexpecting white space or macro step"
```

Figure 6.17: Test skip1 does not execute correctly

```
-------------------------------skip1.maude-------------------------------

EQ
```

Figure 6.18: Test skip1 executes correctly

Thus, the only extra required addition was extending the function `getFullyQualifiedNodeId` to translate its occurrences to fully qualified ids.

Once implemented, the test executes correctly in Figure 6.18.

6.5.3. LookupOnChange

In the past, `lookupOnChange` was supported through an operator that took a set of arguments arguments and a tolerance. However, their semantics was not defined, nor was it correctly translated.

Therefore, focus was set to give support to `lookupOnChange`. This first approach involved creating a suitable translate function in *plexil2maude* and, in the semantics, extending the ID sanitizing function and modifying the loopkup expression evaluation function to retrieve the desired value. In *plexil2maude*, a translation for lookup arguments was introduced, since they were not translated previously and empty arguments were blindly used.

`lookupOnChange` poses an interesting design challenge, because a system of subscription to interface values would have to be devised in order to detect when variables change more than a given tolerance in order to be fed to the environment.

Upon further examination it became evident that microsteps were never interrupted when new external inputs were received from the environment in the PLEXIL Test Executive – in fact, inputs are only read at the very beginning of a macrostep. What is more, even if they could ever potentially be interrupted to receive value changes, our measure of completeness would not change. This lies in the regression tests executed in the PLEXIL Test Executive, where microsteps cannot be interrupted as evidenced by how, when assignment, command and update nodes enter execution and the macrostep is short-circuited, the current microstep is allowed to finish by suspending the involved nodes, still letting other nodes execute nominally during the current microstep. In other words, microstep uninterruption is a precondition for the PLEXIL5 project. This means that, in this context, lookupOnChange behaves as *lookupNow* for all intents and purposes.

Therefore, the decision has been made to transform all lookups – both *Now* and *OnChange* – into the simple form of `lookup`. It is an operator that takes an identifier for an external variable in the interface and a set of arguments. On evaluation of a `lookup`, the system finds the value of the external variable with the given identifier and arguments, returning the corresponding value or unknown otherwise.

This means that a test like `array1` that made use of `lookupOnChange` did not execute in the PLEXIL5 semantics. Because `lookupOnChange` was not properly supported in the past, trying to compare execution of these types of tests gave a similar result as that of Figure 6.15, where there was a parsing error for an unrecognized element. Figure 6.19 shows how the test currently executes correctly, given how the execution of `lookupOnChange` now aligns with the PLEXIL Test Executive.

```
------------------------------array1.maude------------------------------
EQ
```

Figure 6.19: Test array1 executes correctly

6.6 Results

The results of the contributions to the project can be quantified by comparing the number of tests in the regression test suite that initially executed correctly against the number after applying all contributions. Predictably, the number has increased significantly. Before any contributions, 33 tests executed correctly from a total of 155 tests in the test regression suite. After the contributions, 67 tests execute correctly. This entails a 103.03% increase in test correct execution and can be considered a success.

Still, in the grand scheme of the objective of the project to provide a tool to formally analyze PLEXIL plans, PLEXIL5 is currently able to execute 43.23% of the tests in the regression test suite. Compared to the previous 21.29%, it is a noticeable increase, but it is only a moderate success.

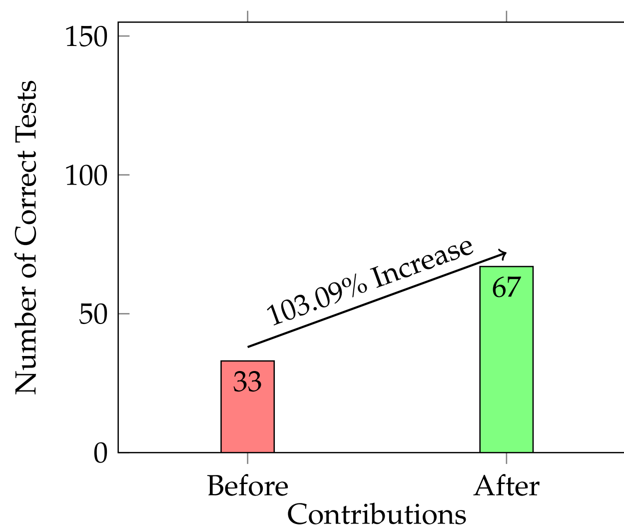


Figure 6.20: Bar chart showing the number of correct tests before and after.

In terms of specific goals, the project has succeeded in implementing Update nodes and reviewing and completing the semantics of Command nodes. Moreover, it has successfully added support for specific features that make more plans execute correctly.

Additionally, this work has yielded an unexpected and favorable outcome. It serves as a contribution of additional documentation that complements the official PLEXIL documentation by filling in the gaps that are left in the realm of the execution semantics of the PLEXIL Executive. This work has delved into the generation of inputs and the process of short-circuiting macrosteps, thus expanding the notion of quiescence within the PLEXIL5 semantics. Consulting this report in the future can greatly assist developers in acquiring a more profound comprehension of the PLEXIL execution semantics.

CHAPTER 7
Conclusions and future work

The presented results demonstrate the amount of work that is required to extend the PLEXIL5 project. Although the underlying numbers indicate a significant improvement in test regression correctness, it is important to note that 57.41% of the tests still encounter execution errors.

As such, the project's next natural step involves continuing to provide support for currently unsupported features, thereby increasing the number of successfully executed tests. This process may involve reworking certain existing features and diligently debugging them to uncover any potential edge cases that could result in failures. This goal can now be easily pursued following the analysis performed in Table 6.1.

Another direction in which the PLEXIL5 project could be extended is the representation of `lookupOnChange`. While the current behaviour of `lookupOnChange` in the PLEXIL Test Executive is analogous to that of `lookupNow`, further analysis would be required to evaluate if there may be some value in implementing a subscription model for node gate conditions (Start, Skip, End, Repeat) that use `lookupOnChange`, as was the first approach to the implementation of `lookupOnChange`.

Moreover, discussions have already been made regarding the architecture of *plx2maude*. Comparatively to that of *psx2maude*, the latter is much more organized due to the logical separation of the tasks of parsing and *prettyprinting*. Therefore, a reorganization of *plx2maude*'s architecture is intended.

In conclusion, the extension of the PLEXIL5 tool has proven to be an overall success. It has not only contributed to the development of a very promising project with vast potential application across various domains but has also significantly enhanced understanding of Rewriting Logic, the PLEXIL language, synchronous languages in general, and real-time systems.

Furthermore, the tool is starting to be used by some industry professionals, which will contribute to further refining the project's requirements and improving its overall functionality in the future.

The extension of PLEXIL5 has undoubtedly resulted a valuable endeavor, demonstrating its capability to drive innovation and foster advancements in the field. Its successful implementation and ongoing utilization promise to yield fruitful outcomes and contribute to the future success of the project.

Relationship between the work performed and the studies taken

It is noteworthy to mention that most of the understanding of Rewriting Logic prior to the start of the work came from the subject of *Industrial Formal Methods* (MFI), which

was studied in third year. It provided a general understanding of the framework and background in working with Maude. It also opened perceptions on the importance of formal verification and how having related tools can be important in areas related to automation and how it can help make existing systems more sustainable.

The importance placed on testing was also a notion that was emphasized during the study of the degree in subjects like *Software Engineering* (ISW) or *Software analysis, validation and debugging* (AVD), but specially in *Software maintenance and evolution* (MES) in the context of ensuring that past functionalities are not broken and making future development easier. This emphasis gave place to the use of TDD in the project.

Finally, the subjects of *Software Process* (PSW) and *Software Engineering Project* (PIN) provided an introduction to agile methodologies that gave a framework to create the used methodology using the resources at hand.

CHAPTER 8

Acknowledgements

I want to dedicate these lines to everyone that has made this project possible. Firstly, to Santiago Escobar, the tutor of this work and without whom neither the project nor the stay at NIA under NASA would have been able to happen, and to whom I am eternally grateful. I also want to thank Marco A. Feliu, who has served as a mentor figure and has provided guidance and lessons that are worth a lifetime, as well as to her wife Laura Titolo for treating us like family many miles away from home.

Bibliography

[1] Pezzè Mauro and Michael J. Young. *Software testing and analysis: Process, principles and Techniques*. J. Wiley, 2008.

[2] Universities Space Research Association (USRA). *PLEXIL Wiki*. URL: https://plexil.sourceforge.net/wiki/index.php/Main_Page.

[3] Brian Dunbar. *Habitation systems project - NASA's Deep Space Habitat*. Dec. 2011. URL: https://www.nasa.gov/exploration/technology/deep_space_habitat/.

[4] B. J. Glass et al. "Robotic and human-tended collaborative drilling au..." In: *56th International Astronautical Congress of the International Astronautical Federation, the International Academy of Astronautics, and the International Institute of Space Law* (2005). DOI: 10.2514/6.iac-05-a5.2.01.

[5] Michael Lowrly et al. URL: https://plexil-group.github.io/plexil_docs/_downloads/7f1a07c5770478fa5d0d2c8a5a3d2c39/AOS.pdf.

[6] José Meseguer. "Conditional rewriting logic as a unified model of concurrency". In: *Theoretical Computer Science* 96.1 (1992), pp. 73–155. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(92)90182-F. URL: https://www.sciencedirect.com/science/article/pii/030439759290182F.

[7] Gilles Dowek, César Muñoz, and Camilo Rocha. "Rewriting logic semantics of a plan execution language". In: *Electronic Proceedings in Theoretical Computer Science* 18 (2010), pp. 77–91. DOI: 10.4204/eptcs.18.6.

[8] Manuel Clavel et al. "Two decades of Maude". In: *Lecture Notes in Computer Science* (2015), pp. 232–254. DOI: 10.1007/978-3-319-23165-5_11.

[9] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. "The Maude LTL Model Checker". In: *Electronic Notes in Theoretical Computer Science* 71 (2004). WRLA 2002, Rewriting Logic and Its Applications, pp. 162–187. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(05)82534-4. URL: https://www.sciencedirect.com/science/article/pii/S1571066105825344.

[10] Vlad Rusu and Manuel Clavel. "Theorem Proving for Maude's Rewriting Logic". In: (Jan. 2007).

[11] Vandi Verma et al. "Plan EXecution Interchange language (PLEXIL) for executable plans and command sequences". In: 2005.

[12] Vandi Verma et al. "Universal-executive and Plexil: Engine and language for robust spacecraft control and Operations". In: *Space 2006* (Sept. 2006). DOI: 10.2514/6.2006-7449.

[13] Universities Space Research Association (USRA). *PLEXIL Documentation - Detailed semantics*. Apr. 2021. URL: https://plexil.sourceforge.net/wiki/index.php/Detailed_Semantics.

[14] Universities Space Research Association (USRA). *PLEXIL Documentation - Node state diagrams*. May 2015. URL: https://plexil-group.github.io/plexil_docs/Appendices/NodeStateDiagrams.html.

[15] Jason Biatek et al. "Analysis and testing of PLEXIL plans". In: *Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering* (2014). DOI: 10.1145/2593489.2593496.

[16] Universities Space Research Association (USRA). *PLEXIL Documentation - Simulating plan execution*. Apr. 2021. URL: https://plexil.sourceforge.net/wiki/index.php/Simulating_Plan_Execution.

[17] Camilo Rocha et al. "A formal interactive verification environment for the Plan Execution Interchange language". In: *Lecture Notes in Computer Science* 7321 (June 2012), pp. 343–357. DOI: 10.1007/978-3-642-30729-4_24.

[18] URL: https://cabal.readthedocs.io/en/stable/index.html.

[19] Andrew J. Kennedy. "Functional pearl pickler combinators". In: *Journal of Functional Programming* 14.6 (2004), pp. 727–739. DOI: 10.1017/s0956796804005209.

[20] Mohammad Masudur Rahman and Chanchal K. Roy. "An insight into the pull requests of GitHub". In: *Proceedings of the 11th Working Conference on Mining Software Repositories* (2014). DOI: 10.1145/2597073.2597121.

[21] Kent Beck. *Test-driven development by example*. Addison-Wesley, 2002.

[22] Gio Lodi. "Getting Started with Test-Driven Development". In: *Test-Driven Development in Swift*. Berkeley, CA: Apress, 2021, pp. 27–42. ISBN: 978-1-4842-7002-8.

[23] P. Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, 2004. ISBN: 9780596552817.

[24] Roy Miller and Christopher T Collins. "Acceptance testing". In: *Proc. XPUniverse* 238 (2001).

[25] Charles L. Marohn. *Confessions of a recovering engineer: Transportation for a strong town*. John Wiley & Sons, Inc., 2021.

APPENDIX A
Sustainable Development Goals

Sustainable Development Goals	High	Medium	Low	None
SDG 1. **No poverty.**			X	
SDG 2. **Zero hunger.**			X	
SDG 3. **Good health and well being.**				X
SDG 4. **Quality Education.**	X			
SDG 5. **Gender Equality.**				X
SDG 6. **Clear water and sanitation.**			X	
SDG 7. **Affordable and clean energy.**				X
SDG 8. **Decent work and economic environment.**			X	
SDG 9. **Industry, innovation and infrastructure.**	X			
SDG 10. **Reduced inequalities.**			X	
SDG 11. **Sustainable cities and communities.**			X	
SDG 12. **Responsible consumption and productions.**			X	
SDG 13. **Climate action.**			X	
SDG 14. **Life below water.**				X
SDG 15. **Life on land.**				X
SDG 16. **Peace, justice and strong institutions.**				X
SDG 17. **Partnerships for the goals.**				X

Table A.1: Degree to which the work impacts the United Nations' Sustainable Development Goals (SDG).

The PLEXIL5 project primarily supports quality education and research that fuels innovation (SDGs 4 and 9). While its current direct impacts on other SDGs are limited, PLEXIL5 signals an open and collaborative ethos that could someday benefit communities through applications in domains like transportation, manufacturing, agriculture, and beyond.

PLEXIL5 most meaningfully contributes to SDG 4 by providing an open platform for students and engineers to learn skills that will drive progress on sustainability. By implementing PLEXIL in a formal verification framework like Maude, PLEXIL5 enables learning about autonomous systems design, programming languages, logic, semantics, and more. These kinds of computing skills will help researchers and developers build applications to address challenges we haven't even thought of yet.

PLEXIL5 also substantially supports SDG 9, to build resilient infrastructure and foster innovation. PLEXIL itself was developed by NASA for spacecraft applications, and implementing it in Maude could lead to discoveries enabling continued advancements in fields like robotics, autonomous vehicles, and hybrid systems. By leveraging these

open-source technologies and collaborative research, PLEXIL5 helps catalyze future applications to meet sustainability goals.

While PLEXIL5's current direct impacts on other SDGs seem limited, the promise of applications like autonomous systems in public transit, freight, agriculture and manufacturing hint at significant future potential to benefit communities (SDGs 11 and 2). Widespread adoption of automation in public mass transit may reduce accidents and make transportation more accessible for elderly, the disabled, and sectors of the population with scarce resources, decreasing social and economic disparities (SDGs 1 and 10). Public transit automation and overall improvement may help develop more tightly connected cities and communities, recovering cities from suburban sprawl by reducing the maintenance costs of road infrastructure and creating safer and more human communities [25]. Precision autonomous farming could also optimize crop yields and reduce water waste (SDGs 2 and 6). Also, the increased productivity from intelligent automation may drive job growth in developing, maintaining and interacting with these systems (SDG 8). Likewise, automation in areas such as manufacturing may help improve resource efficiency, leading to a more responsible form of consumption in regards to the ecologic cost of manufacturing (SDGs 12 and 13).

However, risks and challenges remain around privacy, security, and safety. Autonomous technologies could also significantly disrupt labor markets if not thoughtfully implemented. And their long-term sustainability impacts depend on how they are developed and deployed. Overall though, the openness demonstrated by PLEXIL5 suggests a commitment to responsible, equitable innovation that aligns with the UN's vision.

In summary, while PLEXIL5 itself currently focuses primarily on research, its platform and applications have significant potential to someday support communities and benefit society in meaningful ways. By enabling an ecosystem of learners and builders, open-source projects like PLEXIL5 drive the creativity required to navigate complexity and work toward sustainable development. They help cultivate the human connections and ingenuity that will be instrumental in envisioning and achieving a just, inclusive and prosperous future for people and planet. Although many ambitious sustainability goals remain over the horizon, education and research are the essential first steps.

APPENDIX B
Code

B.0.1. Code for execassignment

```
 op execassignment : Config ~> Config .
ceq execassignment( [ nodes : < A : assignment | status: executing, active:
    true, inv: InvC, post: PostC, endc: EndC, exitc: ExitC, AtS > PS ]
                        [ environment : GAMMA ] [ memory : MEM ] [ microacts : AS
                            ] [ macroacts : AS' ] CONF)
  = execassignment( [ nodes : Nds' ] [ environment : GAMMA ] [ memory : MEM ]
      CONF
       if ancestorExitTrue?(GAMMA, Nds MEM, A)
       then [ microacts : setOutcome(A, interrupted(parentExited)),setStatus(A,
            failing),logTransition(A,executing,failing,1),AS ]
            [ macroacts : undoMem(assignmentVariable(A, Nds)),AS' ]
       else
         if isTrue(eval(GAMMA, Nds MEM, ExitC))
         then [ microacts : setOutcome(A, interrupted(exited)),setStatus(A,
              failing),logTransition(A,executing,failing,2),AS ]
               [ macroacts : undoMem(assignmentVariable(A, Nds)),AS' ]
         else
           if ancestorInvariantFalse?(GAMMA, Nds MEM, A)
           then [ microacts : setOutcome(A, failure(parentFailed)),setStatus(A,
                failing),logTransition(A,executing,failing,3),AS ]
                [ macroacts : undoMem(assignmentVariable(A, Nds)),AS' ]
           else
             if isFalse(eval(GAMMA, Nds MEM, InvC))
             then [ microacts : setOutcome(A, failure(invariantFailed)),
                  setStatus(A, failing),logTransition(A,executing,failing,4),AS
                  ]
                   [ macroacts : undoMem(assignmentVariable(A, Nds)),AS' ]
           else
               if isTrue(eval(GAMMA, Nds MEM, EndC))
               then
                 if isTrue(eval(GAMMA, Nds MEM, PostC))
                 then [ microacts : setOutcome(A, success),setStatus(A,
                      iterationEnded),logTransition(A,executing,iterationEnded
                      ,6),AS ]
                       [ macroacts : AS' ]
                 else [ microacts : setOutcome(A, failure(postconditionFailed))
                      ,setStatus(A, iterationEnded),logTransition(A,executing,
                      iterationEnded,5),AS ]
                       [ macroacts : AS' ]
                 fi
               else [ microacts : AS ] [ macroacts : AS' ]
               fi
             fi
           fi
```

```
        fi
      fi
    )
  if Nds  := < A : assignment | status: executing, active: true, inv: InvC,
      post: PostC, endc: EndC, exitc: ExitC, AtS > PS
  /\ Nds' := < A : assignment | status: executing, active: false, inv: InvC,
      post: PostC, endc: EndC, exitc: ExitC, AtS > PS .
  eq execassignment(CONF)
   = CONF [owise] .
```