UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Regression Test-Driven Extension of PLEXIL5: Support for Arrays

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Jiménez Martí, Alejandro

Tutor: Escobar Román, Santiago

ACADEMIC YEAR: 2022/2023

Acknowledgements

This work is the culmination of a collaboration carried out throughout nearly the entire year 2023, which reached its highest point during my stay at the National Institute of Aerospace under NASA, in the state of Virginia, USA. I would like to show my gratitude to Marco Feliú Gabaldón for his mentoring during my visit and the development of my work, my tutor Santiago Escobar Román for offering me this incredible opportunity and helping me throughout the entire process, and to all who have collaborated in making this experience possible.

Resumen

Este proyecto describe el esfuerzo de extensión de PLEXIL5, un intérprete formal de PLEXIL, especificado en el lenguaje de reescritura Maude para alcanzar mayores grados de corrección y completitud con respecto al PLEXIL Executive, intérprete oficial. PLEXIL es un lenguaje creado para la representación de planes de automatización principalmente usado en robótica y vehículos autónomos. PLEXIL5 se basa en una versión anterior de PLEXIL y ha quedado obsoleta. Este proyecto pretende introducir en PLEXIL5 el soporte para matrices. La estrategia seguida consiste en aprovechar las pruebas de regresión oficiales del PLEXIL Executive. La comparación automática entre las ejecuciones de las pruebas oficiales ejecutadas en el ejecutivo oficial y en PLEXIL5 es una medida de la corrección y completitud del intérprete formal con respecto a la implementación de referencia.

Palabras clave: regresión, prueba, intérprete, semántica, matriz.

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 interpreter. PLEXIL is a language created for the representation of automation plans mainly used in robotics and autonomous vehicles. PLEXIL5 is based on a former version of PLEXIL and has become deprecated. This project aims to introduce support for arrays 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 Executive and in PLEXIL5 is a measure of the formal interpreter correctness and completeness with respect to the reference implementation.

Keywords : regression, test, interpreter, semantics, array.

Resum

Aquest projecte descriu l'esforç d'extensió de PLEXIL5, un intèrpret formal de PLEXIL especificat en el llenguatge de reescritura Maude, per aconseguir majors graus de correcció i completitud respecte al PLEXIL Executive, l'intèrpret oficial. PLEXIL és un llenguatge creat per a la representació d'esquemes d'automatització utilitzats principalment en robòtica i vehicles autònoms. PLEXIL5 es basa en una versió anterior de PLEXIL i ha quedat obsolet. Aquest projecte pretén introduir en PLEXIL5 el suport per a matrius. L'estratègia seguida consisteix a aprofitar les proves de regressió oficials del PLEXIL Executive. La comparació automàtica entre les ejecucions de les proves oficials executades en l'executiu i en PLEXIL5 és una mesura de la correcció i la completitud de l'interpretació formal respecte a la implementació de referència.

Keywords : regressió, prova, intèrpret, semàntica, matriu.

Contents

1. Introduction ... 9
 1.1 Motivation ... 10
 1.2 Objectives ... 10
 1.3 Structure ... 11
 1.4 Workflow ... 12
 1.5 Work proposal ... 15
2. State of art ... 17
 2.1 PLEXIL ... 17
 2.2 Maude ... 20
 2.3 PLEXIL5 ... 23
 2.4 Git and GitHub ... 25
 2.5 Continuous Integration ... 26
3. Problem Analysis ... 29
 3.1 PLEXIL5 Interpreter ... 29
 3.2 PLEXIL5 Translator ... 32
 3.3 State of Array Tests ... 39
4. Solution's design ... 43
 4.1 Executor script ... 43
 4.2 Translator and interpreter ... 44
 4.3 Continuous Integration tasks ... 45
5. Solution's development ... 47
 5.1 Executor script ... 47
 5.2 PLEXIL5 ... 49
  5.2.1 First Iteration ... 50
  5.2.2 Second Iteration ... 51
  5.2.3 Third Iteration ... 53
  5.2.4 Final Iteration ... 53
 5.3 PLEXIL5 Pipeline and Docker image ... 55
 5.4 PLEXIL Pipeline and Docker image ... 58
6. Testing ... 61
7. Conclusion ... 63

7.1 Relation with the degree 65
8. References 67

List of Figures

FIGURE 1: WORKFLOW DIAGRAM 13
FIGURE 2: SIMPLE HIERARCHICAL PLAN REPRESENTATION [5] 18
FIGURE 3: PLEXIL EXECUTION DIAGRAM 19
FIGURE 4: MAUDE REWRITING LOGIC DIAGRAM WITH CODE SNIPPET 21
FIGURE 5: SIMPLE MAUDE MODULE EXAMPLE [8] 22
FIGURE 6: DIAGRAM OF TRANSLATOR OPERATION 24
FIGURE 7: EXAMPLE OF THE DIFFERENT STEPS IN OUR PIPELINE 28
FIGURE 8: PLAN TOPOLOGY REPRESENTATION WITH CURSOR 34
FIGURE 9: TRANSLATOR'S STRUCTURE SCHEMA 36
FIGURE 10: EXAMPLE OF ARRAY FROM TEST ARRAY1.PLX 38
FIGURE 11: EXAMPLE OF ARRAY FROM TEST ARRAY1.PLX 38
FIGURE 12: VARIABLE DECLARATION FROM TEST ARRAY1.PLX 39
FIGURE 13: RESULTING REPRESENTATION IN MAUDE 39
FIGURE 14: RUN.MAUDE TEMPLATE 40
FIGURE 15: EXECUTOR SCRIPT OPERATION DIAGRAM 43
FIGURE 16: DIAGRAM OF TRANSLATION OF AN ARRAY 44
FIGURE 17: PIPELINE OPERATION CONCEPT 45
FIGURE 18: TESTS' STATE PREVIOUS TO OUR WORK 49
FIGURE 19: TESTS' STATE AFTER THE FIRST ITERATION OF WORK 51
FIGURE 20: TESTS' STATE AFTER THE SECOND ITERATION OF WORK 52
FIGURE 21: TESTS' STATE AFTER THE FINAL ITERATION OF WORK 55
FIGURE 22: PIPELINE'S INITIAL STATE 56
FIGURE 23: PIPELINE'S EXECUTION AFTER OPTIMIZATION 57
FIGURE 24: PIPELINE'S EXECUTION IN CURRENT STATE 58

Glossary

PLEXIL Plan Execution Interchange Language

PLEXIL5 PLEXIL Formal Interactive Verification Environment

NASA National Aeronautics and Space Administration

XML Extensible Markup Language

PR Pull Request

CI Continuous Integration

CD Continuous Deployment

1. Introduction

Software reliability has been a focal point for both software developers and consumers ever since it took its first steps, with a clear upward trend in importance. The latter is evident in today's business world, with an increase in product support time, which exceeds the development time itself, and a greater commitment to teams focused on quality and product maintenance.

In the context of achieving a reliable and robust product, developers work on the creation of a series of tests, metrics and analyses to be performed on the software, so that they can fully guarantee the correctness of a program with respect to values that are considered acceptable.

Test suites are not only useful in an initial development context, but also exploit their potential in an iterative work environment, in which software is constantly updated, new elements are introduced, previous ones are removed, etc. Test suites serve teams as a reference point in the development of a product, being key to detect possible bugs introduced in such updates and to locate these bugs within the totality of a product. The process just described is known as regression testing.

The actions related to product quality can become very complicated and tedious tasks in certain software systems, having a directly proportional relationship as the size of the system grows. Therefore, it is desirable for such systems to possess a series of properties and characteristics that facilitate their own analysis. Because of it sometimes it is chosen to make representations of certain systems in other languages for the application of different techniques, in this case we are going to speak about the representation of PLEXIL in Maude named PLEXIL5, with the objective of guaranteeing the correctness and leaning on the techniques of Formal Verification that Maude offers.

PLEXIL was first created by NASA to satisfy the demands of adaptable, effective, and dependable execution of plans in space missions, and is mainly focused on the automation of tasks and interacting with the environment. Maude is a high-performance reflective language that is commonly used for system modelling due to its high degree of expressiveness and supports rewriting logic and equational logic.

1.1 Motivation

The motivation of this project is to extend the functionality of the PLEXIL Formal Interactive Verification Environment (PLEXIL5), a tool that contains a formal PLEXIL interpreter, in an effort to obtain a complete representation of the original PLEXIL interpreter. PLEXIL5 offers access to formal verification techniques such as static program analysis, model-checking or theorem proving.

Formal verification can be described as a smart way of verifying a system, a manner of avoiding the process of checking the correctness of every single bit of code. Even if that hypothetical scenario of examining a system entirely line by line could be possible in theory, it would be extremely inefficient to conduct and nearly impossible to apply to larger projects. However, Formal Verification provides us a way of exploring the execution space as a whole and can be used to find errors, eliminate them and most importantly demonstrate that design defects are not present. The latter is an important feature that differentiates Formal Verification from testing, emulation and simulation techniques, so that the design of a system is considered more reliable given that full coverage is reached. Efficient data structures and algorithms are the key ingredients of Formal Verification, enabling it to capture all potential design behaviors and swiftly search for counterexamples when design requirements are not met [1].

In a context of automation such as that of the PLEXIL system, all these functionalities and features offered by formal verification are more than desirable, as they can provide greater guarantees in critical systems such as those in which PLEXIL operates.

1.2 Objectives

The ultimate goal of this project is to achieve a representation of PLEXIL5 as correct and complete as that of the original PLEXIL interpreter. It is necessary to point out that the creation of a model exactly like the original system is practically impossible, so a more logical approach to the problem has been chosen. This procedure is based on the idea of using the regression test suite used by the official PLEXIL interpreter to guarantee its correct functioning, also in the new interpreter PLEXIL5. In this way we aim to ensure that the results of these tests are exactly the same, thus guaranteeing the same degree of correctness and completeness as PLEXIL. It should be noted that

in this context and for this test suite, when we talk about equal results, we do not mean correct or incorrect results. In this case the test results are a series of state transitions, which will be explained later, and what we are looking for is that the states and transitions are the same in both runs of the same test.

This document will focus on the efforts related to the work done to extend the PLEXIL5 interpreter, with the aim of including full support for arrays. In order to achieve this, a series of small objectives have been set, in order to get to know how the system works and to gain fluency within it, as well as to work efficiently and to introduce changes in a safe way, supported by tests that go with these modifications. These subobjectives can be grouped as follows:

- Reading papers and documentation related to PLEXIL's operation.
- Analysis and understanding of the PLEXIL5 interpreter, as well as of the different parts that make up the representation, identifying how they relate to each other and what each one does.
- Examine the current state of the system to see which parts need to be corrected and which are not yet addressed. Obtaining in this way a complete list of missing or incorrect features.
- Duplicate the official PLEXIL5 repository to get our own copy to make changes in parallel.
- Consider the different solutions and implement the one considered most appropriate at any given time.

1.3 Structure

The report begins by putting the original PLEXIL system in context, as well as the different technologies used to develop PLEXIL5, explaining the choice of Maude as the language, reviewing the advantages of formal verification in more detail and briefly explaining the state of development of the new representation.

The following part will explain how PLEXIL5 works, breaking down the system into its different parts and describing what each part does and what it contains. It will also contain an explanation of how the comparison of test results is done between both representations, and what is considered to be the exact same results.

Afterwards, the initial analysis of the system prior to the introduction of changes will be presented, along with the results obtained from such analysis. After this section, specific examples related to arrays will be shown, detailing the problems they had initially along with their incorrect result, and the process carried out to solve them will be displayed.

Then, a number of tools created to streamline the execution and comparison of test findings will be provided. The explanation of these tools will describe the many components that make them up, how they function, and the inspiration for their creation.

Finally, the work will undergo a retrospective analysis of everything that has been added throughout the course of its development, gathering the issues and tests that have been resolved both directly and indirectly.

1.4 Workflow

The way in which the work process has been approached to achieve the objectives is extremely important to understand the title of this work. The concept of regression test suite and the existence of one in the PLEXIL source code has been previously presented. Well, this test suite is a pivotal element in the way of working. It is a reference element because it is the way to determine which parts of the system are incomplete or have an incorrect implementation.

The working procedure begins with the selection of a test from the original suite and its execution in both the PLEXIL and PLEXIL5 interpreters. From there, the test results obtained by both interpreters are checked, in order to ensure that the results are exactly the same. In the case we are going to focus on, the tests related to the arrays, these presented some kind of difference with the original results, and from there we started with a process of analysis and debugging in the different parts that make up the PLEXIL5 system (the interpreter and its different parts). Once the cause of these variations was located and prior to the implementation, a unit test was created to check the faulty behavior of the system with the aim of making it fail in the first instance. Subsequently, different possibilities for a solution were considered and developed. Finally, the regression test created in association with

the problem was run to verify that the defect had been adequately corrected along with the rest of the unitary and acceptance tests.

It is worth noting the last part of the previous point since the existence of said unit and acceptance tests are of the highest importance to guarantee the correct functioning of PLEXIL5 and serve as support and verification that the introduction of new functionalities has not generated any conflict or error with what was previously implemented correctly.

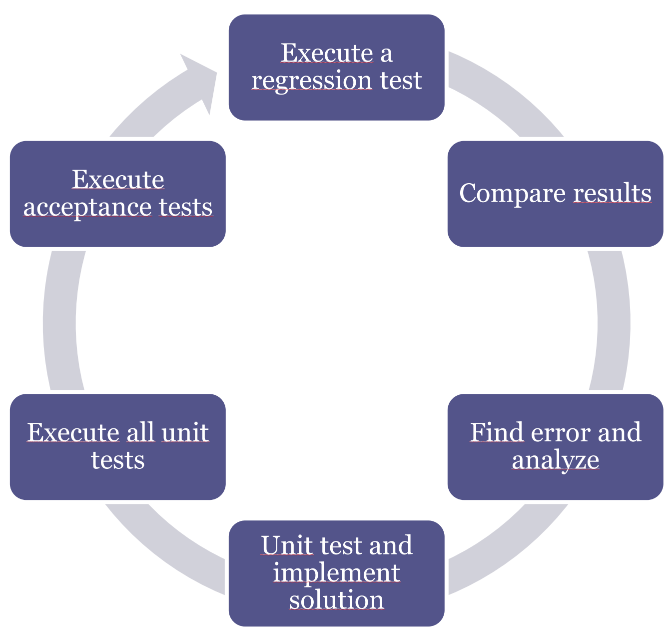


Figure 1: Workflow diagram

As it can be seen in Figure 1, we distinguish between unit, acceptance and regression tests. The concept of regression testing has already been covered previously in the introduction and what it means for this context. On the other hand, and as we will see later, unit tests are those that test an isolated functionality of the system to confirm that each unit works according to its specifications and always seeks to encompass the smallest possible part of the code, while acceptance tests are the opposite, those that test the product as a whole to ensure the correct functioning of the product when all the components that form it work together [2]. To finish with this explanation on testing, it is necessary to clarify that regression testing can be

performed at any level (from unit to acceptance), and what marks it as regression is its purpose, to detect potential side effects and errors introduced in the code.

To finish with the methodology, it is necessary to talk about the role that Git has played in the work carried out, being a focal point when developing. Git is a well-known version control system (VCS), specifically GitHub has been used, a recognized web-based hosting service for Git repositories. It provides a platform for collaborative development and offers additional features like issue tracking, project management tools, and code review.

The way we worked in relation to Git was as follows:

1. A regression test was chosen in which different results were obtained in both interpreters and a GitHub issue was created to represent this test, with a brief description or commentary of where the problem was thought to be in the system.
2. A branch was also created from GitHub associated to that issue, where work on a solution was started.
3. Once the same test results were obtained for both interpreters, a Pull Request (PR) was opened for the rest of the team to review the correctness of the work done.
4. Finally, once the changes had been approved, the branch associated with the issue was merged with the main branch of the repository and the issue was closed, allowing the process to be carried out again.

All the technology-related concepts mentioned above are explained in detail in Section 2.4.

In this way we get several things, such as keeping the main branch always stable and error-free, and related to this, by separating the branches by specific functionalities facilitates the possibility of reversing the changes made in case an error is inadvertently introduced. In addition and to complement all the aforementioned, when uploading changes to the repository, the unit and acceptance tests were automatically executed in a pipeline to give more security to the whole process. Both the pipeline concept and the explanation of its development are covered in depth later in the document.

1.5 Work proposal

The general objective of this work was to extend PLEXIL5 so that the system reaches a state of completeness on par with that of PLEXIL, and to this end we have worked to ensure that as many regression tests as possible generate the same result in both systems.

However, this paper focuses on what has been developed to introduce full support for arrays and the tests that contain them, as this is a fundamental data structure for any computing environment. Nevertheless, the study of the system and subsequent modification has led to changes that have helped to introduce improvements that affect different parts of the system.

2. State of art

2.1 PLEXIL

Plan Execution Interchange Language (PLEXIL) is a language originally created by NASA in collaboration with Carnegie Mellon University to represent automation plans, and in turn a technology capable of executing those plans in real or simulated environments [3]. Its main use since its creation has been related to robotics, control of unmanned vehicles, and in general demonstration and prototyping tasks, such as the Ocean Worlds Autonomy Testbed for Exploration Research and Simulation which uses PLEXIL for onboard lander autonomy or its use in a coordinated demonstration of Human Robot Interaction with K10 Mars Rover.

PLEXIL was first created to satisfy the needs of executing plans in space mission operations in a flexible, effective, and dependable manner. Given the same sequence of external occurrences it is deterministic and also semantically unambiguous, and compact. The language may also describe branching, loops, timed and event-driven activities, concurrent activities, sequences, and temporal limitations, and it is also highly expressive. The language's basic syntax is straightforward and consistent, facilitating the use of validation and testing procedures while also facilitating easy and effective plan interpretation.

Included with PLEXIL is the Universal Executive, an execution engine capable of implementing PLEXIL efficiently, that offers interfaces to controlled systems [4].

A PLEXIL plan is composed of a series of nodes, which mark the behavior of the system. A node defines an action to be performed, and this action can range from an interaction within the plan to a communication with the external environment. In PLEXIL there are several sorts of nodes, these include list nodes, update nodes, command nodes or assignment nodes, among others; all with different types of structure and definitions, with disparate objectives and specifying a particular type of behavior [5].

PLEXIL nodes are organized within the PLEXIL plan with a tree structure, in which there is only one root node. As can be deduced from the structure, the nodes are

ordered from top to bottom according to the degree of granularity, i.e. they are placed hierarchically. Thus, the nodes closer to the root represent more general tasks, while the leaf nodes are the representation of more concrete and "close to programming" tasks such as variable assignment, library calls or command executions.

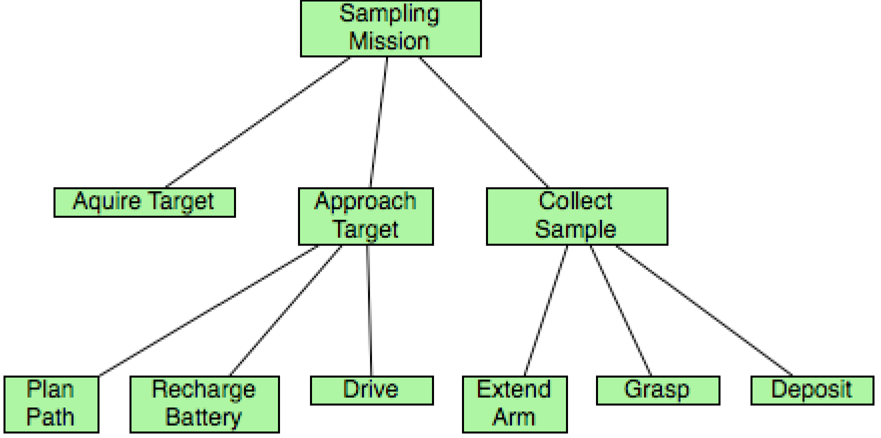


Figure 2: Simple hierarchical plan representation [5]

Nodes are always in one of the following states:

- Inactive
- Waiting
- Executing
- Finishing
- Iteration Ended
- Failing
- Finished

When reaching the finished state, the nodes will have one of the outcomes, whose initial value will be unknown and then can take one of the following values:

- Success
- Failure
- Interrupted
- Skipped

The internal transitions of the nodes that make them pass from one state to another are defined by the different internal conditions of each node. Among these conditions are some such as start condition, end condition, pre and post condition. All of them also influence the outcome of the nodes, and even in some cases certain conditions of predecessor and successor nodes are taken into account, due to the hierarchical structure of the plans.

Plans are executed in a series of steps, which can be classified as micro steps, quiescence steps or macro steps. The micro step can be defined as the synchronous application of the atomic relation to a plan's largest possible collection of nodes, being the atomic relation the state transition of an individual node. An outside event triggers a macro step. Micro steps are used to move each of the nodes waiting for that event to its next state in parallel. If these transitions cause more condition changes because of local data changes, they are carried out until there are no more enabled transitions. Quiescence step refers to the process of continually applying micro steps until no further transitions are enabled. A new external event is handled after a macro step has been executed [6].

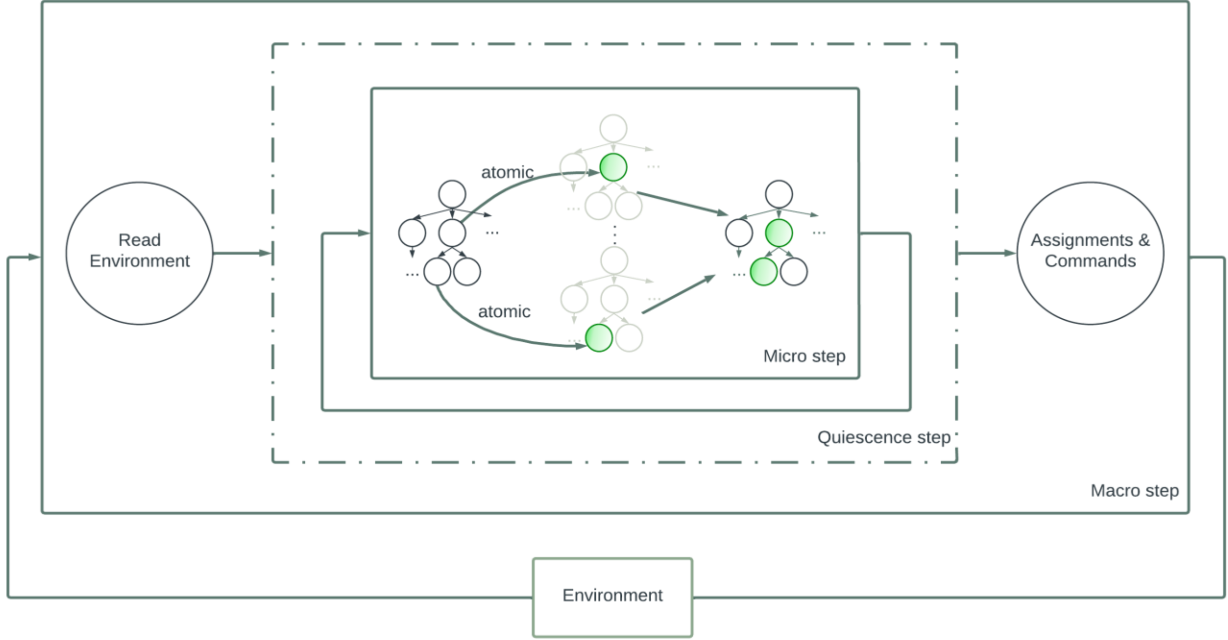


Figure 3: PLEXIL Execution diagram

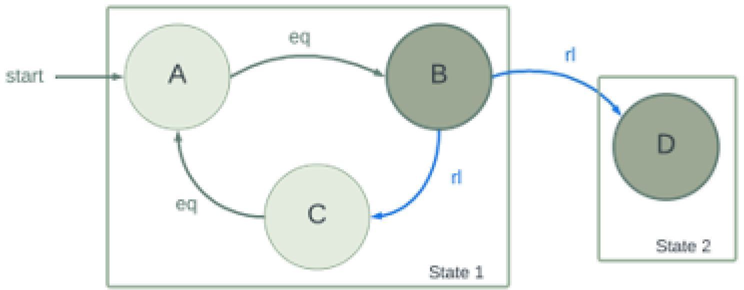
In addition to the plans, it is important to explain the function of the scripts to understand this work. Scripts are used by the Test Executive, which is essentially a plan execution simulator, to represent the behavior of the external world [5]. Along with carrying out the plan, the Test Executive examines the events and answers in the script throughout plan execution. If necessary, the event sequence or beginning state of the script might be left empty. The main objective of this procedure is to make sure that the plan is carried out properly and reacts appropriately to developments and updates. Both compiled plans and compiled scripts are defined in XML, and this is crucial to understand the task conducted by PLEXIL5.

2.2 Maude

Rewriting logic makes it possible to describe a wide range of computational and logical models while also enabling concurrent modification. However, there are practical difficulties when implementing the rewriting semantics of a synchronous language like PLEXIL in Maude. This is due to the fact that, despite the possibility of concurrent synchronous specifications in rewriting logic at the mathematical level, Maude manages the maximum concurrency of rewrite rules by interleaving concurrency during execution [7].

Maude is a language for formal reasoning and the study of complex systems. It is a high-level declarative programming language. Rewriting logic, on which it is built, enables concurrent and non-deterministic computations. Maude is made to facilitate the formal specification, verification, and validation of systems, including distributed systems, software and hardware systems, and protocols. It has a wide range of applications in fields including software engineering, artificial intelligence, computer security, and mathematical logic. A built-in interpreter and model checker are also included with Maude, making it simple to experiment with and examine system requirements.

Maude, in the context of rewriting logic, uses expressions to describe the state of the system, using equations for deterministic transitions and rules for non-deterministic transitions.




```
eq A = B .
eq C = A .

rl B = C .
rl B = D .
```

Figure 4: Maude rewriting logic diagram with code snippet

Maude supports in a systematic and efficient way logical reflection. This makes Maude remarkably extensible and powerful, supports an extensible algebra of module composition operations, and allows many advanced metaprogramming and metalanguage applications. Indeed, some of the most interesting applications of Maude are metalanguage applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation [8].

The members of the Maude team, apart from developing the language itself, are working on applying Maude in different areas, such as [9]:

- General Logics and Logical Frameworks
- Specification Languages
- Semantics of Programming Languages and Models of Computation (our case)
- Concurrent and Distributed Systems
- Formal Tools and Formal Interoperability
- Reflection and Metaprogramming
- Object-Oriented Modeling and Programming
- Real-Time Systems
- Bio Informatics
- Mobile Languages

- Network Protocols and Active Networks
- Multi-level modelling

Next, we are going to analyze a simple Maude example to see what the code can look like:

```
mod SIMPLE-VENDING-MACHINE is
    sorts Coin Item State .
    subsorts Coin Item < State .

    op null : -> State .
    op _ _ : State State -> State [assoc id: null] .

    op $ : -> Coin .
    op q : -> Coin .
    op c : -> Item .
    op b : -> Item .

    var St : State .

    rl [r1] : $ => c .
    rl [r2] : $ => b q .
    rl [r3] : q q q q => $ .
endm
```

Figure 5: Simple Maude module Example [8]

In this small module of Maude, we are representing the operation of a vending machine in a simple way. In the sorts we have defined coin, item and state, being the first two subsorts of the last one. Then we have defined the operators that represent the empty object, a list of objects, a dollar, a quarter of a dollar, a coffee and a biscuit respectively. Finally, we have the rewrite rules that determine:

- r1: we get a coffee in exchange for a dollar.
- r2: we get a biscuit and a quarter of a dollar in exchange for a dollar.
- r3: we obtain a dollar from four quarters.

In conclusion, the use of Maude to make a representation of PLEXIL can be attributed to the expressiveness it presents, the high degree of performance it provides,

the ease of use it offers as a modeling tool and the infrastructure it provides for formal verification, ranging from search command and built-in model checker to theorem proving applications.

2.3 PLEXIL5

PLEXIL Formal Interactive Verification Environment or PLEXIL5 is a tool that was originally created to support the formal verification of PLEXIL. An efficient interpreter and reference implementation of PLEXIL have been produced as a result of the language's formalization [10]. Additionally, it gives a clear and useful interpretation of the language that may be used.

The outermost goal of PLEXIL5 is to help ensure the correctness and reliability of mission critical systems that use PLEXIL plans, and this can be done thanks to all the features offered by Maude for formal verification [11].

Focusing more now on the different parts that make up the system and its structure, it is worth mentioning two very different parts, the translator and the interpreter in Maude. Up to this point PLEXIL5 had been talked about as a homogeneous system with some indication that there were different elements making it up, but the reality is that there are two clearly separate parts: the translator and the interpreter.

The main purpose of the translator, as can be deduced from its name, is the generation of plans and scripts in Maude from the original XML files. The idea is to obtain files equivalent to those defined for the original interpreter, but executable by the PLEXIL5 interpreter. As previously explained, there is a suite of regression tests published in the PLEXIL repository, and the purpose of the translator would be to run it on each and every one of the test files that make up that suite, in order to generate an equivalent test suite for PLEXIL5, this being the first step towards the ultimate goal of achieving a system with the same guarantees of correctness and completeness.

The translator is defined in Haskell and is also composed of two parts. A first part can be defined as a parser, which is the part in charge of accessing the XML files and traversing the information trees to extract the information in an Abstract Syntax Tree. After this process, the pretty print task begins, accessing the tree generated by

the parser, and from there generates an appropriate Maude code, with a correct syntax and all the particularities that this may bring. Internally the translator is known as Plexil2Maude.

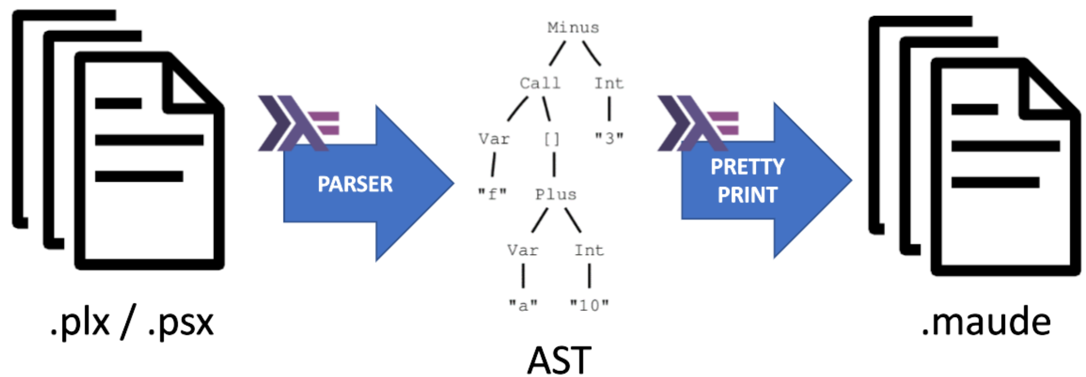


Figure 6: Diagram of translator operation

The second major component of PLEXIL5 is the interpreter or semantics, which is the part defined in Maude. This is the most complex section of the whole system, since it is the part in charge of modeling and representing the original PLEXIL, therefore all the particularities of the language are implemented. All types of nodes, attributes, conditions, node transitions, etc. are defined in this part of PLEXIL5.

Despite being differentiated, both the translator and the interpreter must work together and be on the same page, since the translator generates the plans and scripts in Maude, and these files must be built according to the definition they have in the interpreter, so that they can be understood, executed and generate a readable output.

There are a series of commands that are important for the functioning of the whole system. Some of these commands are:

- *plx2maude and psx2maude:* as their name indicates, these commands puts the translator into operation, generating the Maude files from XML. The first is used for plans and the second for scripts.
- *plexiltest:* used to run original tests that are passed as arguments
- *plexilog-diff*: it compares the results of the original test and its corresponding PLEXIL5 version

To finish with the description of PLEXIL5, it is necessary to emphasize the existence of both unit and acceptance test suites. To define it in a simple way, unit tests are those that test a concrete part of the system, for example the translation of an array of integers, and acceptance tests are those that test a system in its totality in a final environment, in this case it would be the execution of a previously defined plan together with its input script.

As previously explained, the existence of these suites has been of vital importance for the development carried out in this work, and it is something that will continue to be emphasized throughout the document.

2.4 Git and GitHub

Git is a distributed version control system (VCS) that makes it possible for numerous developers to work together on a project by keeping track of file changes and organizing their efforts. It offers capabilities like branching, merging, and versioning to efficiently handle source code and other text-based files [12].

The following are some essential Git concepts:

- Repository: a group of documents and folders that also includes a record of all the changes that have been made to them. It might be a remote repository hosted on a server or a local repository on your machine.
- Branch: a distinct path of development. Allows multiple developers to work on multiple features or fixes at the same time. Branches offer a versatile approach to managing work in progress, as they are easy to set up, merge, and delete.
- Merge: the integration of modifications from one branch to another. Conflicts that could occur if modifications overlap are resolved by combining commits from multiple branches.
- Commit: a record of the modifications made to the files in the repository at a certain moment. Each commit is identified by a special hash and includes a message describing the modifications, the identity of the author, and a timestamp.

GitHub is a platform created to host Git repositories as well as all the references it contains, and being a system deployed on the internet aims to facilitate the work

together. It has a great popularity and apart from the version control features, it adds many other features that are very useful for development teams, below I comment those that are more relevant for this project [13]:

- Issue: allows you to keep track of tasks, problems, feature requests, and other project-related issues. It offers a focal point for the discussion and resolution of project-related problems. An issue is created, then it is assigned to someone (using a GitHub's username) and it can be used to discuss given that you are able to comment under it. Once the project achieves the objective (such as introducing a feature or fixing a bug) that the issue englobes it is usually closed, and although it does not appear in the main page of the issues anymore, it can be consulted in the closed issued window.
- Pull Request (PR): a mechanism for proposing and discussing changes to a project. You can propose that changes are made in a branch to be merged into another branch, usually the main branch. Code review and collaborative development frequently employ pull requests. When creating a PR, for the suggested modifications you select the source and target branches. Then, reviewers can easily observe the changes performed since PRs show the differences between the source and target branches. Reviewers are invited to discuss the suggested code changes, offer suggestions for improvements, and make comments. Finally, when the PR has been approved the changes from the source branch are merged into the target branch and the PR is closed.

Apart from the mentioned features, GitHub offers functionalities related to access control, continuous integration and even setting up Wikis for every project.

2.5 Continuous Integration

Nowadays, the development of applications is focused on the joint work of different people at the same time on the same system, which leads to having different versions of the code simultaneously. If it were decided to merge all the versions on the same day, this would potentially cause many problems and the task of merging would become very complicated, mainly because the modifications of some developers can generate a conflict with the versions of others.

In order to fight this issue, Continuous Integration (CI) is a software process that requires code to be committed to a common repository on a regular basis. More frequent code commits help to catch errors faster and decreases the amount of code a developer must debug when determining the cause of an error. Code updates on a regular basis also make it simple to combine modifications from different members of a team. This is beneficial for the developers since it allows them to spend more time building code and less time debugging or resolving merge conflicts [14]. To sum up, thanks to Continuous Integration developers can frequently submit code in small increments (i. e. on a daily basis). These code changes are automatically subjected to building and testing processes before they are integrated into repository.

Closely linked to continuous integration is the concept of pipeline. A pipeline is a tool used for continuous integration/continuous deployment (CI/CD), although we are only going to focus on the CI side. A pipeline refers to a series of steps that the code must go through from initial commit to deployment [14]. Typically, pipelines are separated by phases and steps, which encompass different tasks. An example of how to use the different phases of a CI pipeline could be as follows:

1. Build: When the pipeline is triggered a build process begins automatically
2. Integration: The changes are merged with another stream of development, usually the main branch
3. Testing: The newly merged code is tested automatically
4. Reporting: the pipeline provides logs of the process and the results of the tests

The pipelines are highly customizable allowing adaptation to all kinds of needs that development teams may have, as well as the integration of other automatic functionalities that bring development value to the team. For example, prior to the build step the pipeline is triggered by some event, which can be anything from a commit (it usually is) to a time of day.

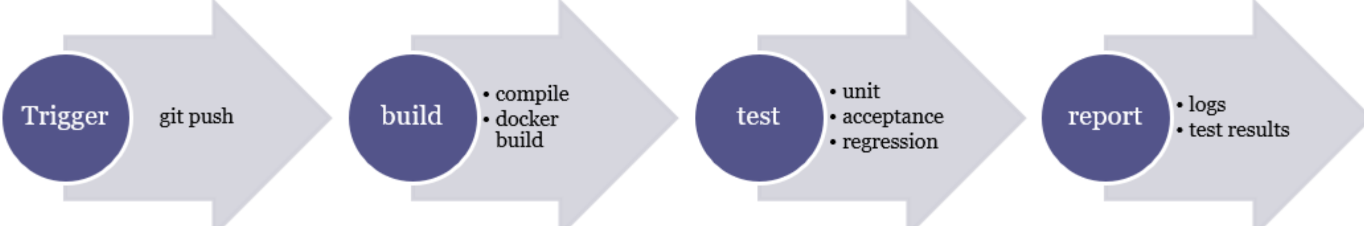


Figure 7: Example of the different steps in our pipeline

In order to obtain an appropriate execution environment docker is commonly used in the context of a pipeline. Docker is a popular platform for creating and managing lightweight, portable containers. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. [15].

3. Problem Analysis

After introducing PLEXIL5 in a general manner and ensuring familiarity with the system's terminology, we will conduct a comprehensive review of all its components, their functioning, outstanding components, as well as those yet to be introduced.

Initially, our focus will be on commenting on the structure of the source code of the PLEXIL interpreter in Maude. This will entail analyzing the most pertinent aspects relevant to the system's overall operation. Following this, we will proceed to conduct a thorough study of the translator, with particular emphasis on the distinctions between plans and scripts and delving more deeply into the XML parsing process. Lastly, we will scrutinize the preliminary outcomes of the array tests, which were produced while the system was still in its nascent stage, and we will establish a correlation between these outcomes and their potential root causes.

3.1 PLEXIL5 Interpreter

As previously described, PLEXIL5 is divided into two main parts and although both are important and add great value to the system, the most crucial part of it is the interpreter defined in Maude. The reason why this part is the most important is because everything that is developed is focused on what is defined in this interpreter, since it will be in charge of executing the plans and generating a result. It is also an irreplaceable section, since the translator could be replaced by other alternatives, which will be discussed later.

In terms of structure, the interpreter is divided into a series of files that are responsible for different parts of the execution and which communicate with each other. The separation of functionality is loosely based on the structure of the source code of the original PLEXIL interpreter written in C++, although the functionality of the modules is mainly delimited by logical groups of operators or declarations.

To better exemplify what we mean, it is best to talk about the main modules of the interpreter, along with what their function is and some small code snippets. In this way, it is easier to understand what the interpreter's job is when executing a plan.

- plexilite.maude

It is one of the most important classes in the whole interpreter since it loads the PLEXIL specification in Maude runtime. The PRELUDE module is a library that offers a selection of fundamental concepts and tools that are often used in formal specifications. Definitions of fundamental data types, mathematical operations, and popular data structures like lists and sets are all included. prelude.maude additionally includes operations and methods that may be used to manipulate terms and create formal specifications. It is usually loaded automatically when you launch the Maude console.

Apart from that, in this module there are also defined some basic functions along the lines of `isTrue` or `isFalse` that are used from all around the code. To get plexilite loaded in the Maude interpreter you would first launch it and then once inside you would type the following command:

```
load path/to/plexilite.maude
```

- compiler.maude

This class is also very important because it compiles a PLEXIL plan into a set of processes. It contains all the functions to compile the different types of nodes and convert them from simple Maude code to more complex objects with their different characteristics.

Beyond compilation, this class also contains the methods for sanitizing node IDs, and by sanitizing we mean changing the ID references to nodes or variables to their fully qualified version. A fully qualified ID in Maude is a mechanism to refer to a particular PLEXIL entity within the Maude PLEXIL5 Interpreter. The name of the module and the name of the identifier itself are separated by a dot to form a fully qualified ID [16].

- funpred.maude

All auxiliary system definitions are included in this module. It contains all the checks on the status of the conditions of the nodes in the hierarchy, e.g., whether the parent node is waiting, whether all successor nodes have finished or checking the invariant condition of all ancestors. Such checks are of vital importance for the definition of node state transitions, it is the node conditions that mark from which state to which state the

node transitions to. This module is based on the official node state diagrams, which are a series of illustrations that describe the node state transition semantics of PLEXIL nodes [5].

There are also other operators such as `nodeType` that return the type of a node or `hasAborted` that check if a node has aborted.

Another relevant operator is `getLookups` that recursively computes the lookups occurring in any expression.

- interface.maude

This module contains everything related to the PLEXIL interface. The interface in PLEXIL describes how a plan or component interacts with its surroundings and other components. It offers a standardized method for communication and information sharing between various PLEXIL system components. In other words, the interface is used to simulate the interaction with the environment, and this module covers this functionality [17].

- nodes.maude

This class is very important and has a similar function to that offered by funpred.maude seen earlier, although it was focused on access to the attributes and conditions of the ancestor/successor nodes. In this case, nodes.maude is the one that marks the attributes that each node has, for example *initialValue*, *outcome*, *handle*, *arguments*; and also sets the conditions, such as *precondition*, *postcondition*, *endcondition*, etc. Along with the definitions are also the methods for accessing and replacing the values of conditions and attributes. As in funpred.maude, these values are very important for node transitions.

- Reductions

Unlike the rest of the topics listed above, this does not reference a concrete module, but an entire directory. This directory contains all the files and modules that are related to node transitions, and it is a key part of the system because it is the one ultimately causing that plans give the same outputs as the original results.

An error in the operation of PLEXIL5 always brings us here and given the erroneous transitions in the result we look for the error in the corresponding operator.

In spite of this, the errors do not have to be caused directly by the definitions of the transitions, in many cases the defect is found in some other operator that is declared somewhere else, but in this way, we can limit the defects exclusively to what that transition uses.

Finally, note that all the PLEXIL transitions defined in Section 2 are included in this directory, atomic, micro step and macro step (quiescence is not because it is more a concept than an actual computation).

- values.maude

This module is not of such high importance for the system as all the previously mentioned ones. I only wanted to highlight it here because it does have more weight for the work developed since everything related to arrays in the context of the interpreter has been added in this module.

To end this section on the interpreter, it is necessary to clarify that these are only some of the most important modules that make up PLEXIL5. The reality is that there are dozens of files with various functions and sizes, all of which contribute with some functionality to the overall system. For example command.maude that defines everything related to the commands as its own name indicates or the environment.maude module that defines how the interpreter interacts with the inputs that arrive from the scripts.

3.2 PLEXIL5 Translator

It has already been explained that the translator is divided into two parts, the parser and the pretty print, so let's look deeply into the technology behind both parts. It is important to remember that both are defined using the purely functional language Haskell.

The translator for plans and scripts, however, is specified independently, as commented before, and it is separated principally for two reasons. The first explanation has to do with the workflow of development, i.e., initially the emphasis was on producing the plans since they were more significant and the scripts were separated as the system's development went on. The second, more significant cause is connected to the previous one and the legacy code. An independent implementation was chosen as opposed to expanding the current capabilities once the translator's development in

relation to the scripts got underway, because adding new functionality to the plan translator would have required excessively lengthy and intricate code. Additionally, a different technology was chosen for the parser that would result in cleaner, more readable code. Although this may sound like a minor peculiarity of the system, the fact that different technologies are used to perform the same task increases the complexity of the system and contributes to making the learning and familiarization curve of PLEXIL5 more complicated.

Once the particularities that make up plexil2maude have been discussed, we are going to focus first on the plan parser. As previously mentioned, the entire translator is defined in Haskell, regardless of plans or scripts, parser or pretty print. Well, for the plan parser we use a Haskell programming pattern known as cursors. A cursor is a type of data structure that enables you to navigate and change a group of objects in a sequential order while keeping track of where you are within the group [18]. The standard components of a cursor include a reference to the currently selected element, a set of actions that enable cursor movement to the next or previous element, as well as the ability to insert or remove items at the current place. In Haskell, cursors are frequently used for navigating and altering collections like lists, trees, and other data structures, although we are only interested in the traversing functionalities. Specifically, we are using the cursor defined in the Text.XML module, a library created for parsing and manipulating XML documents in Haskell. The cursor module provides functions like 'element' and 'attribute' for selecting XML elements and attributes, and functions such as 'child' and 'descendant' for navigating the XML document tree.

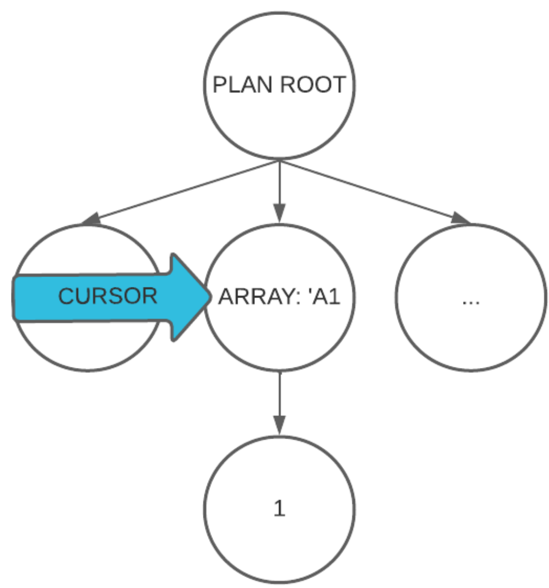


Figure 8: Plan topology representation with cursor

Looking at Figure 8, a cursor could be understood as a node selector within an XML tree. That is, through a series of operations performed on it we can modify the node of the document to which it refers. Using Figure 8 as an example and taking into account that the cursor "is" in the declaration node of the array of one element, to access its contents we would perform the *child* operation on the cursor to move the reference to the node that contains the value, and finally we would apply the *content* function on that resulting cursor to access the number. Note that this example is based on the actual topology of PLEXIL plans, though it has been simplified to make the cursors easier to understand.

To finalize with the parsing of plans I wanted to comment on the way errors are handled. Given that Haskell is a purely functional language, side effects are not allowed by the language [19] and this makes working with the outside of the program complicated, in our case, the task of reading plans (external XML files). Haskell utilizes a notion known as "monads" to overcome this problem. A monad is constituted by 3 parts:

1. The “monad type” or context.
2. A “return” function that wraps the value in the context.

3. A "bind" function that combines a monadic value and a function that generates another monadic value into a single monadic value.

There are several types of monads, but we are going to focus on the Maybe monad. It is frequently used when a calculation may fail, and we want to handle the failure in a safe and predictable manner. There are two constructors for the Maybe monad: Just and Nothing. The Just constructor is used to wrap a value representing a successful calculation, whereas the Nothing constructor is used to indicate a failed computation [20].

Now that we have explained the technology used for the parsing of the plans, let's review the alternative used in the scripts, the *picklers*. *XHT* is a Haskell library that provides picklers, and they are used to serialize and deserialize Haskell data structures into and out of several external formats [21], although we are only interested in XML. It allows specifying how your data types should be translated to and from XML using a collection of type classes and combinators that are provided. In our case we define as many instances of *XmlPickler* as the number of different kind of nodes that we can find in a script. If we take Figure 8 as an example (even though it represents a plan), there would be an instance of XmlPickler for the type to which the root node belongs, another for that of the array declaration node, and a third one for the node containing the value; each one containing in its definition how to encode/decode the XML that represents that node.

Regardless of the technology used, in both cases to go through the XML we rely on the XML schema that determines that these plans are valid. An XML Schema is in charge of defining the structure of XML documents that are assigned to that schema and the valid data types for each element and attribute. Thus, the possibilities of control over the structure and data types are very broad [22]. It is an XML vocabulary that provides a collection of rules and restrictions for verifying XML documents against a set of criteria. In this way we can know in a delimited manner what elements and attributes a node can contain to make the parsing task much simpler.

Additionally, we have the technology that transforms the information extracted by the parser (whether it be plans or scripts) and formats it so that it is syntactically well-constructed code in Maude, and also matches the semantics defined by the interpreter, so that it is able to execute the plans and generate a result. The technique used is known as pretty print, a method typically used to modify code and data in a way

that is cleaner and more readable. By displaying the code or data in a structured manner, pretty printing aims to make it simpler for people to read and comprehend, and to help programmers in their debugging and maintenance tasks [23], however in our case we take advantage of its features in order to transform data into Maude programs. Specifically, we use the *Text.PrettyPrint* Haskell module which is part of the *pretty* package.

Again, we find differences between plans and scripts, because while the plans have both the parser and pretty print altogether in the same file, the scripts have them defined in different places, so that readability is much greater and its code is much cleaner in general. For plans it is difficult to find the corresponding pretty print function that goes along with the parsing call, or even sometimes it is defined together in the same function and this can cause problems when introducing new elements to the translator or when modifying it. On the scripts side it is clearer given that every instance of an *XmlPickler* has it corresponding Pretty instance, so that everything is strictly related.

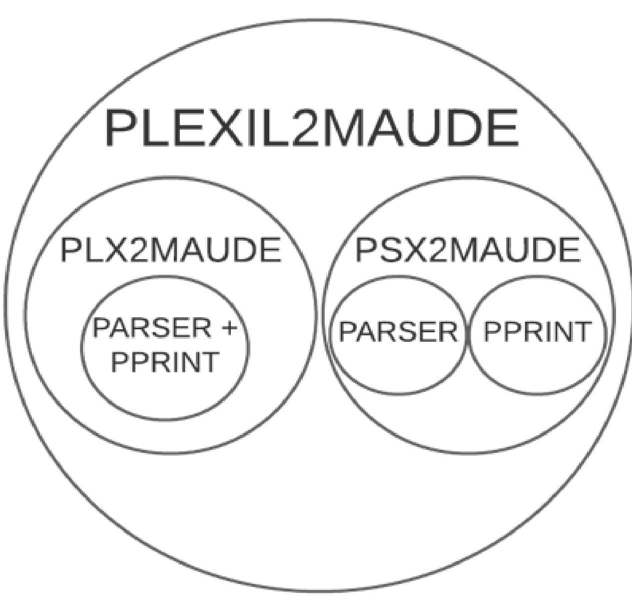


Figure 9: Translator's structure schema

Apart from the plan translator itself, the plexil2maude section also contains a suite of tests to ensure its proper functioning. The tests are defined in Haskell using the Tasty library, a framework that aims to make it easier to create dependable and high-quality software. It offers a full set of tools and functionality for designing and running

Haskell tests, and it is based on the Test Anything Protocol (TAP) for interoperability with a variety of testing environments [24]. Once again, tests work differently for plans and scripts with the major difference residing in the amount of things they cover. For the plans, the input of the tests are concrete extracts of an XML that encompass an aspect of the translator (a piece that represents an array for example) and the expected result is the Maude code generated after the pretty print. On the other hand, the tests of the translator related to the scripts are divided in two, having on the one hand a test that takes as input XML and expects a result in the intermediate representation (it tests the *xpickle*), and on the other hand a test that takes as input the intermediate representation and whose expected result is that Maude code. This difference is further evidence of the diversity that makes up the system and highlights the superiority of the script translator in terms of readability and best practices because by having the tests separated by functionality, it is much simpler to specify where the translation error can be found.

These problems are compounded by the state of the suite in general, mainly due to the lack of extension and little coverage for many of the features offered by the translator, as well as the need for the creation of new, more general executor functions, since those originally implemented were very specific and were not useful for running new tests. Even so, it is a very useful tool, and has a great importance in the work done.

To give more dimension to this subsection, we must talk about the reason for the translator's existence, because perhaps after reading all the difficulties associated with it, the question arises as to why the tests are not translated by hand. The main reason is due to the size of the test suite and the length of the tests themselves, because being XML files it is not always trivial to check the composition and hierarchy of the nodes manually. For this reason it is considered that the time invested in making this manual translation would be better spent generating a translator that does it automatically, obtaining several advantages in the process such as greater reliability and homogeneity in the conversion of the XML files, eliminating the need of having to translate tests that could be introduced in the future or facilitate re-generating tests in cascade if it is decided to change the implementation of the interpreter.

Moreover, as the test suite has been developed since PLEXIL was first introduced, and the language has been evolving for a long time now, the syntax of the language has been contemplating new options and therefore this has made the test

suite heterogeneous, that is, it has created the possibility of representing two equal elements in a different way as in the following example:

```
<DeclareArray ColNo="2" LineNo="3">
  <Name>a1</Name>
  <Type>Integer</Type>
  <MaxSize>3</MaxSize>
  <InitialValue>
    <ArrayValue Type="Integer">
      <IntegerValue>1</IntegerValue>
      <IntegerValue>2</IntegerValue>
    </ArrayValue>
  </InitialValue>
</DeclareArray>
```

Figure 10: Example of array from test Array1.plx

```
<DeclareArray>
  <Name>a1</Name>
  <Type>Integer</Type>
  <MaxSize>3</MaxSize>
  <InitialValue>
    <IntegerValue>1</IntegerValue>
    <IntegerValue>2</IntegerValue>
  </InitialValue>
</DeclareArray>
```

Figure 11: Example of array from test Array1.plx

As it can be deduced from Figures 10 and 11, both XML fragments represent an array of 3 elements in size, which have an initialized 1 and 2. Contrary to what it may seem, both representations are correct and accepted by the interpreter. To help us with this problem we check the XML schema as it has been explained before.

To finish with the translator let's see an example of how a simple piece of code in a PLEXIL plan would be transformed to Maude. In this case we are dealing with the declaration of a variable *i* of type *Integer* initialized to 0 (`i = 0`). The result can be seen in Figure 13 where the name of the variable is used as an identifier and its value is wrapped with the *val* constructor.

```xml
<VariableDeclarations>
  <DeclareVariable LineNo="6" ColNo="2">
    <Name>i</Name>
    <Type>Integer</Type>
    <InitialValue>
      <IntegerValue>0</IntegerValue>
    </InitialValue>
  </DeclareVariable>
</VariableDeclarations>
```

Figure 12: Variable declaration from test Array1.plx

```
('i : val(0))
```

Figure 13: Resulting representation in Maude

3.3 State of Array Tests

Before looking at the results of the array tests and prior to the implementation of our solution, we are going to see with an example of how a test execution process would look like. The name of the chosen test is array1.ple.

First, we have to compile the test, so that we obtain array1.plx. To do so we employ the following command:

```
plexilc path/to/array1.ple
```

Once we have the compiled version of the test, we can translate it to Maude, given that the translator works with XML.

```
plx2maude path/to/array1.plx > path/to/store/array1.maude
```

In the previous command we execute plexil2maude and redirect the output to a directory where we want to keep the plans in Maude. For scripts, it would be the same but using ‘psx2maude’ instead.

Now that we have both plan and script ready, it is time to execute them. For the original array1.plx it is a simple task. We execute the command (it has been simplified to improve readability) below and redirect the output.

```
plexiltest -p path/to/array1.plx -s path/to/array1.psx > output.plexil
```

For Maude plans it is more complicated, as we use a template called run.maude, that helps us load every module into the Maude interpreter. The template looks something like what is showed in Figure 14.

```
load ../../src/plexilite.maude
load path/to/plans/array1.maude
load path/to/scripts/array1.maude

set print attribute on .
set print color on .

mod PLAN-SIMULATION is
protecting array1-PLAN .
protecting INPUT .
endm

rew run(compile(rootNode,input)) .
q
```

Figure 14: run.maude template

Then to run this file we would do:

```
maude path/to/run.maude > output.maude
```

To end with this process we compare outputs using plexilog-diff as follows:

```
plexilog-diff path/to/output.plexil path/to/output.maude
```

As you can see, this is a rather tedious and not very scalable process, as it would take a lot of time to run the entire test suite. It is also one of the problems that will be addressed later in the design and implementation of the solution.

In relation to the results of the tests, and after doing the above process for each of the tests where arrays are used, the result in all of them was the same: *Warning: bad token createArray*, and *Warning: no parse for statement*. What we can deduce from this is that there is part of the translator that is contemplating the arrays and transforming them to Maude but it is not doing it in a way that the interpreter can understand it correctly. Looking further, in the semantics code what we find is a very rudimentary and insufficient representation of the arrays; and regarding the translator we see the definition of some methods that do not offer complete coverage.

This previous analysis provides us with a roadmap for development, giving us a first idea of which parts of the system we are going to concentrate our work on.

4. Solution's design

4.1 Executor script

The need to expedite the test execution process has already been mentioned in the previous section. Therefore, before going into detailing the solutions for the translator and the interpreter, it is necessary to design a system that allows us to test those changes introduced later on quickly and easily.

The chosen solution was a Python script due to its ease of use, familiarity with the language, the large user community and the variety of libraries offered.

The idea is that the script automates the entire process previously explained, so that it allows us to compare all the tests at once by executing only one file. It needs to get the original plans from the appropriate directory, compile them, translate them to Maude, run both, and compare the results. The whole process is explained in Figure 15.

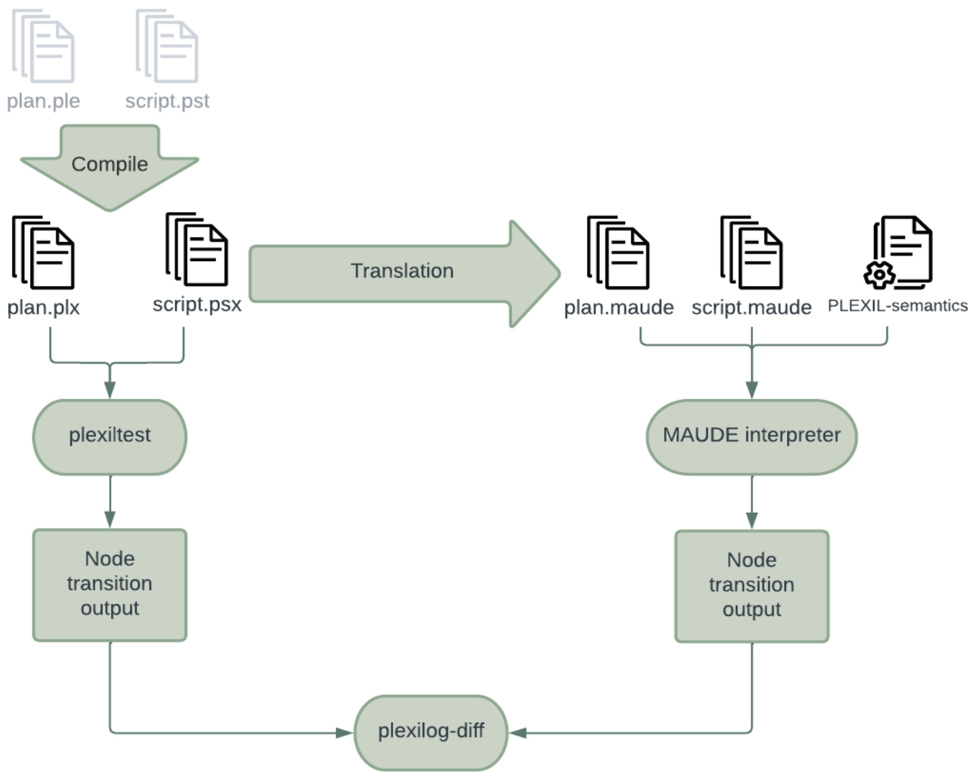


Figure 15: Executor script operation diagram

4.2 Translator and interpreter

At first, we decided to keep the solution to the problem as simple as possible. The idea was to extend the translator so that it would adapt to the simple definition offered by the interpreter in that version.

To do this, by analyzing the functionality offered by the translator, some functions that were related to arrays could be found, but in the first instance, two elements that had to be included stood out. These were the XML nodes *DeclareArray* and *ArrayValue.*

DeclareArray, as its name suggests, are the nodes that wrap the entire initial declaration of an array. It contains the type, size, elements, and values, as seen in Figures 10 and 11.

Additionally, *ArrayValue* does not appear in all the examples of arrays, but when it does, it is in charge of wrapping the values inside *InitialValue* in a *DeclareArray*, and they also have the data type as an attribute, which will be a problem like will be seen later. This can also be seen in Figure 10.

The translation that was devised for arrays at first is the following:

```
<DeclareArray>
  <Name>a1</Name>
  <Type>Integer</Type>
  <MaxSize>3</MaxSize>
  <InitialValue>                              ⇒        const(array(1 # 2))
    <IntegerValue>1</IntegerValue>
    <IntegerValue>2</IntegerValue>
  </InitialValue>
</DeclareArray>
```

Figure 16: Diagram of translation of an array

In Section 5 it will be explained that this decision of adapting to what exists in the interpreter was a mistake, and that the representation originally chosen was not descriptive enough to obtain a correct result. We will analyze why and the measures that were adopted to correct the problem.

4.3 Continuous Integration tasks

The ultimate goal of these tasks was to generate two pipelines. The first pipeline that was designed is a tool with more potential in the future, because its purpose is based on executing all the tests of the official PLEXIL suite every time a new one is added, to verify that everything continues to work correctly.

Additionally, the second pipeline arose as a need when developing. The problem that it seeks to solve is avoiding the mistakes that sometimes occurred when introducing new changes to the repository without first checking if the tests passed locally, i.e., it serves as a control tool that allows us to know if defects would be introduced when merging a PR. It does this checking by executing all of the tests of both the parser and the interpreter each time a push is made to the repository,

For the pipelines we used CircleCI, a platform that offers Continuous Integration functionalities. And to ensure that we had the correct environment we had to build two different Docker images that encapsulate our working environment. First, we designed an image that contained just PLEXIL with Ubuntu 22.04 as Operating System for the first pipeline, and then we used that image as a base for the one that adds Haskell, Maude and PLEXIL5. The implementation of such images will also be described below.

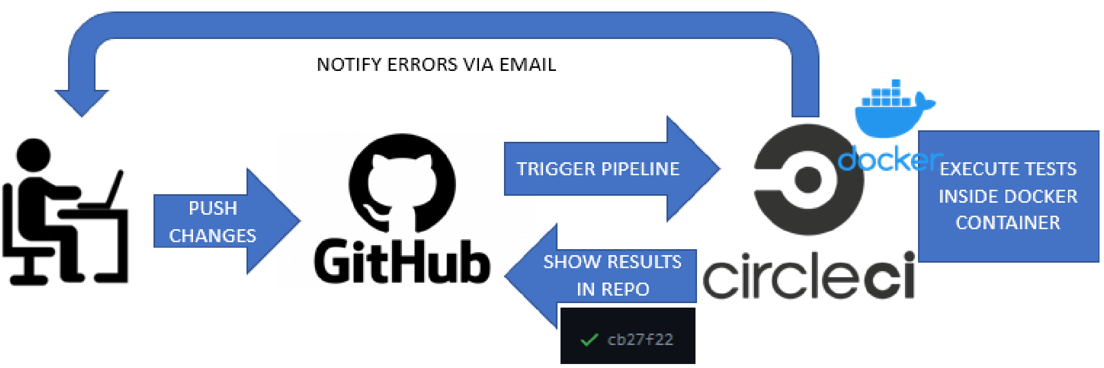


Figure 17: Pipeline operation concept

5. Solution's development

This section will be divided into the same subsections that we found in Section 4. In the first subsection we address the initial problem of speeding up the execution of the tests as well as the different ideas that emerged during the development. The second subsection details the process of extending PLEXIL5 in detail along with an analysis of the state of the tests. Finally, the last two subsections focus on the development and problems encountered when configuring the pipelines and creating the docker images.

5.1 Executor script

Although this script is one more element that makes up the PLEXIL5 system, it is necessary to separate it from the other parts of the development because the problem it addresses is totally different. As explained in the previous section, the main purpose of this script is to speed up the compile-translate-execute-compile-compare process that we perform to determine if a test generates a correct result.

For the same reason, the main operation of the system performs these tasks automatically, although there are some peculiarities in its operation.

The first particularity is to change the working directory of the script from the directory it is located into the root of the repository. This is done mainly because throughout the script we are using paths that are at different levels of depth, (either to access files, redirect outputs, etc.) and for convenience, if we are in the root directory we can use the symbolic paths from that point instead of having to use them from the file location, which would generate some pretty counter-intuitive paths. For this we use the function call `os.chdir(os.path.dirname(filename)),` to which we pass as filename a file that is in the root of the project and always will be (*.gitignore*). To run terminal commands we use the built.in Python *os* library as follows: `os.system('command')`.

Another remarkable aspect of this script is the parallelization of tasks. Parallelization is used for the compilation of plans and scripts, and its aim is to multiply the productivity of the script, since having a large number of files to compile would make the execution quite slow if it had to go one by one. For this task we use the python *subprocess* library. This same library is used to assign a timeout to the

execution of tests, in order to detect those that are still incorrect that enter an infinite loop and abort their execution.

This is what the script included once we started using it. However, as a result of its early use, a number of needs arose to make its operation more comfortable. These needs were met by introducing execution flags that altered the behavior of the script depending on whether they were used or not, using *argparse* to include them. The flags are the following:

- -c, --compile: This flag must be included after the script name if we want the original tests in *.ple/.pst* format to be recompiled. The reason why you have to specify to compile them instead of the other way around is because in the vast majority of cases it makes no sense to compile the original tests again once you have converted them to *.plx/.psx* for the first time, since they are rarely changed. In this way we avoid having to compile the original plans every time we want to execute the script, saving a lot of time in the process, but we maintain the possibility of doing it if necessary.

- -t, --test <name>: This flag is one of the most useful since it allows us to specify the name of the test with which we want to work. One of the biggest advantages is the flexibility it has to receive names as a parameter, since it does not need a specific name, but you can simply use words that are contained in the name of the test, for example, to execute all array tests you could only do it in the following way: `script.py -t array`

- -np, --no-parse: This flag is one of the most counterintuitive, but it is used to avoid performing the plan/script translation process again. This is mainly for debugging purposes, because sometimes we want to modify the translated files in a previous execution and see the behavior of the interpreter with those arbitrary modifications. By removing the translation, we prevent those changes from being automatically rewritten by a new execution.

- -r, --redirect <path>: Again, it is a flag designed for debugging errors. The concept of this flag is to redirect Maude's standard output to a file on the path we pass to it. The standard output of Maude is not the state transitions that are saved in the file to be compared (*output.maude*), but rather a file with detailed information about the execution, such as rewriting of variables.

- -p, --pipeline: This flag was included late in development and is explained in more detail in Section 5.3. It configures the script in a mode intended for pipeline execution.

As a final note, it must be acknowledged that all these flags can be consulted from the command line with the following prompt: `script.py -help`.

5.2 PLEXIL5

Before starting the development of the tests, we are going to clarify the state of the tests prior to the implementation of our solution. The main reason for carrying out this reasoning in this section is to give importance to the title of the thesis, since these tests are the ones that have marked the path of work, setting the objectives for the development and serving as a guide; That is why we say that our work has been Regression Test-driven

Test Name	Cause of error
array1.plx	Wrong translation
array2.plx	Wrong translation
array3.plx	Wrong translation
array4.plx	Wrong translation
array5.plx	Wrong translation
array6.plx	Wrong translation
array8.plx	Wrong translation
array9.plx	Wrong translation
ArrayAssignmentWithFailure.plx	Wrong translation
LibraryNodeCallWithArray.plx	Wrong translation

Figure 18: Tests' state previous to our work

(red)	Different results as the original suite
(green)	Same results as the original suite

It may seem trivial now given that it has been explained that the translation was incomplete, but since the development will be explained referring to the different iterations carried out to complete the arrays in PLEXIL5, this table will serve as a metric to know how far we are advancing with each introduced change.

The work will be explained in four iterations, which were not explicitly planned but emerged naturally during the development process. In this way it can be better understood what changes were made and why the final representation was chosen.

5.2.1 First Iteration

The work to be done for this first iteration was quite simple and straightforward. The first thing was to develop methods in the code for the parsing and consequent Prettyprint of the ArrayValue nodes.

To carry out this task, we analyzed the code of the plan translator and observed that many types of nodes with a similar character were concentrated in one area of the code, such as IntegerValue, StringValue, etc. In order to try to maintain the coherence and homogeneity of the code, we decided to introduce it in that same part. The implementation details are not excessively relevant, the only thing worth noting is the differentiation of boolean values and strings. Booleans had to be treated differently because in the PLEXIL code they appeared in uppercase, and we need to convert them to lowercase for our interpreter (*toLower*). In the case of strings we have to differentiate because, like most interpreters, the PLEXIL5 interpreter recognizes strings wrapped in double quotes (*doubleQuotes*).

In the case of DeclareArray, as it is the outermost XML node of an array, we simply define a method that considers the possibility that a node of this type might exist and from there we define a series of very simple external methods to perform checks. The final objective of this method, explained in a simple way, was to wrap the values parsed by ArrayValue with the array constructor (*array(values)*).

Even though this explanation has focused more on the plans, for the scripts the philosophy followed was the same, although due to the very topology of the scripts, in which the elements are notably simpler, their development is not of great interest for the description of the solution at the moment.

At the interpreter level, no changes were planned for this iteration, since we sought a solution that was as simple as possible.

Once the changes were introduced and we tried to run the tests again, the results were as follows.

Test Name	Cause of error
array1.plx	Wrong translation
array2.plx	Wrong translation
array3.plx	Wrong translation
array4.plx	Wrong Node Transition
array5.plx	Wrong translation
array6.plx	Wrong translation
array8.plx	Wrong translation
array9.plx	Wrong translation
ArrayAssignmentWithFailure.plx	Wrong translation
LibraryNodeCallWithArray.plx	Wrong translation

Figure 19: Tests' state after the first iteration of work

The results after this first iteration of work were rather disappointing since we expected to be able to correct at least half of the tests. The reality was different since the only test that showed signs of progress was array4.plx, but this was not good news either since showing a difference in the state transitions indicated that it was probably going to be necessary to make changes in the interpreter.

5.2.2 Second Iteration

For the second iteration there was no previous design, and the roadmap was determined by the debugging process carried out to determine what elements were missing to translate correctly, as well as a first look at the possible changes to be made in the interpreter. From this analysis we were able to observe an incorrect operation with everything related to the assignments in the context of the arrays, as well as the non-existence of a method to be able to compare elements of arrays through equality.

To solve what is related to the assignments, we divided the tasks into two, on the one hand the translator and on the other hand the interpreter. But in any case, it would be necessary to update the representation of the arrays.

```
const(array(1 # 2))
```

⇩

```
const(array(val(1) # val(2)))
```

The reason for updating the representation is due to the way the interpreter treats the values, since it only recognizes the values if they are wrapped by the constructor of the previous example, in this way we can already assign values to the elements of the array since the representation of the values is homogeneous for the entire system.

Therefore, we got a translation error since the interpreter did not understand the values by themselves but needed the constructor. To implement it in the translator we only had to add one more wrapper in the deepest part of the tree, the one that extracts the values. Regarding the interpreter, instead of looking for homogeneity we tried to add expressiveness, changing the way assignments are handled by adding explicit equations for different types of variables, and thus making sure that they always match.

For the equality of arrays, we added in the section of the translator where the different types of equalities were defined another case that was adapted to our needs.

The state of the tests after this new iteration of work was as follows:

Test Name	Cause of error
array1.plx	Wrong translation
array2.plx	Wrong Node Transition
array3.plx	Wrong translation
array4.plx	Wrong Node Transition
array5.plx	Wrong Node Transition
array6.plx	-
array8.plx	Wrong translation
array9.plx	Wrong Node Transition
ArrayAssignmentWithFailure.plx	Wrong Node Transition
LibraryNodeCallWithArray.plx	Wrong Node Transition

Figure 20: Tests' state after the second iteration of work

The results after this round of modifications were much more promising, eliminating practically all the translation errors, and even obtaining our first test with a correct result in array6.

5.2.3 Third Iteration

This iteration was the simplest of all but it is necessary to separate it from the others because it focused on the translation of the scripts.

The only task we carried out during this third stage was the creation of a series of value types to facilitate the parsing of the plans and subsequent pretty print. With this we got the scripts to work properly for all types of arrays (regardless of the values they contain).

It is not necessary to show the tests' status table since from the beginning we knew that this was not going to solve all the errors we had, therefore the tests' state is the same as in the previous iteration. In any case, it is an iteration as important as the others, since with this we solved the problem with the entry scripts and we guaranteed that we would not have more problems with this part of the translator in the future.

5.2.4 Final Iteration

For this part of the development, we had to stop again to analyze and reason. First, we started with a tedious process of debugging the errors obtained in the tests, and once the problems were detected we agreed on the most appropriate solution for each of them. As a note, changes introduced in the translator in this part are minor, because during this analysis we detected that the translation errors that occurred in the array tests were mainly due to other elements that are outside the scope of this report, in any case, these errors were fixed in parallel.

The modifications introduced in this part are the most complex at a conceptual level, and the ones that took us the longest to implement correctly. The main problem that we detected in the tests of arrays that had erroneous state transitions, was that these state changes began to diverge from operations on the arrays, either reading or writing. At first, we attributed these errors to the operations themselves, but after several approaches we realized that this was not the problem.

The real problem was in the representation of the arrays itself since we were not passing the value of the size of the array. That is, we were not contemplating the fact that, for example, an array can be created with a length of 5 values but only

initialize 3 at declaration time; and therefore all the operations that were carried out outside those initialized values were giving an error, because for the interpreter everything that went beyond that was outside the scope of the variable. This caused an error, and therefore the tests had a different state transition or simply ended their execution incomplete.

To solve it we adopted the following representation:

```
const(array(val(1) # val(2)))
```

```
const(array(4, val(1) # val(2)))
```

It is a trivial solution, to pass that value to the interpreter, and for it to be in charge of creating an array of that length but with all the "spaces" of the array initialized. We get the value from an XML node inside the DeclareArray called MaxSize.

At this point we thought that adapting the interpreter minimally to our solution would be enough, but we had to make quite big changes.

To begin with, we realized the need to use an empty value to represent those array positions that exist but are not initialized, because they cannot be easily represented in any other way. To do this we use an already existing value within the interpreter, the so-called ‘unknown’, which is used in other sections for the same purpose. In line with this, we decided to represent arrays without any initialized values as *unknown<Type>Array*, to have them initialized with unknown values, but type-distinguishable so that you know you cannot write any value other than one from its kind, thus avoiding accidentally having arrays filled with values of different types.

Finally, several methods had to be separated, first the arrays are created and then they are filled with unknown values in the remaining gaps if necessary using an auxiliary method. In addition, the assignment methods have been changed, using one for each type of value, which substitutes as many unknown elements as values that you want to introduce.

All these main functions led to the creation of other helper methods, such as one that determines the position of a value within an array or one that determines its total length. In general, it is the most complete iteration of all, from both a technical and

effortwise point of view, since together with the new introductions, it was also decided to refactor what already exists to homogenize the construction of the arrays.

Finally, and thanks to the changes introduced, the status of the tests is as follows:

Test Name	Cause of error
array1.plx	-
array2.plx	-
array3.plx	-
array4.plx	-
array5.plx	-
array6.plx	-
array8.plx	-
array9.plx	-
ArrayAssignmentWithFailure.plx	-
LibraryNodeCallWithArray.plx	Wrong translation

Figure 21: Tests' state after the final iteration of work

As you can see, the test that includes a LibraryNodeCall continues to give a translation error, but this is because this type of node is not yet defined within PLEXIL5.

5.3 PLEXIL5 Pipeline and Docker image

Leaving aside the development of the tests, we will now focus on the continuous integration work carried out to optimize the workflow of the previous section. As already explained in Section 4, specifically in the part that refers to this same topic, the objective of these pipelines was to make the system more robust, seeking to avoid the problems generated by forgetting to execute the unit and acceptance tests defined in the system.

The development of this pipeline is divided into two phases: creation of the docker image and generation of the *.yml* configuration pipeline.

Regarding the docker image, as previously mentioned, we need PLEXIL to run commands like `plexilc`, Haskell to run the translator, Maude to work with the interpreter and PLEXIL5 to have the source code. For PLEXIL there was no major problem, as we used another docker image that we had previously created, the development of which will be explained in the next subsection. Getting PLEXIL5 into the execution environment was no problem either, as CircleCI allows you to copy the repository with a simple checkout command in the configuration file.

The problems with the image came mainly from Haskell and Maude, not from getting them but from the specific version needed to make the system work. For Haskell, once the versioning problem was identified, it was only necessary to specify the version to apt-get, the Linux package manager. For Maude, however, it was not so easy, as the Linux package manager had 2.7 as the most recent version available, but we needed 3.1. At first, we looked for the option to download the files in a *.zip* via cli and *unzip* them with the unzip library, but this generated a lot of permission problems inside the image. Finally, it was decided to download the files through the browser locally and add them to the Docker image with the COPY command, and once the folder was inside the image perform the installation process recommended by Maude's team.

The image is uploaded to the docker repository and CircleCI pulls it and uses it as a runtime environment. With what has been described so far, the result was the following:

Figure 22: Pipeline's initial state

As can be seen, it worked correctly, but the execution time was excessively long for the few tasks that were being performed and the short real duration of the tests. This is because the plans and scripts translator has to be installed before it can be used in order to work properly, and this installation is what took up most of the pipeline's execution time.

Figure 23: Pipeline's execution after optimization

To solve this, we decided to redo the docker image with a static version of PLEXIL5 which was cloned in build time, and once we had it we installed the translator. This way the long install only had to be done once, when creating the image, and when using it in the repository we only had to update this installation to introduce the new changes, using the `--overwrite-policy=always` flag in the install command.

With this strategy we managed to reduce the execution time to approximately 1 minute, reducing it by 80%.

However, at this point the idea arose to use the pipeline to also monitor the status of the regression tests, i.e., to use the executor script in the pipeline runs. To do this, a record is kept of the tests that already have a correct result, and each time a commit is made, the result of all the tests with the new code is checked and compared with the previously stored results. In this comparison we get two things, both the new supported tests and the tests that previously gave a correct result and now fail. Obviously, the aim is only to increase the list of supported tests, but in this way we can also receive a warning in case we have introduced an unexpected error.

The pipeline execution mode was created for this purpose, and it throws an exception when the results differ so that the pipeline fails and notifies us of new tests that are supported, or of previously supported tests that fail, or both.

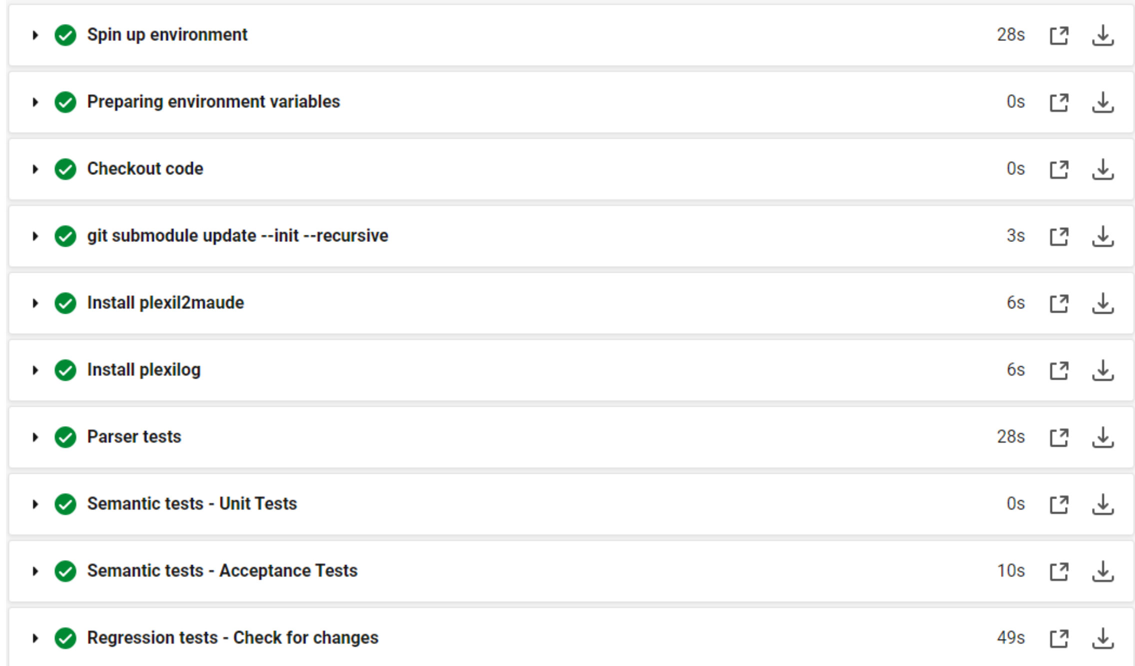


Figure 24: Pipeline's execution in current state

As it can be seen in Figure 24 in this step, we also refactored the configuration file to separate the commands installation steps from the logic itself and added a step to access the original PLEXIL tests that are in the repository as a submodule. With this addition we get a very fast and expressive pipeline as it checks all relevant aspects of the PLEXIL5 tests.

5.4 PLEXIL Pipeline and Docker image

For this pipeline we also created a docker image, the one mentioned above, which uses Ubuntu 22.04 as a base. At first it seemed as simple a task as cloning the official PLEXIL repository, adding it to the system PATH and running the ‘make’ command for its installation. And even though those were the main steps, we had to face a convoluted problem to solve from an environment such as creating a Docker image.

The problem was that some files that used *cstring* were missing the module import. Being an official NASA repository to which we did not have access, we could not solve it from the root, but we had to correct it in the local copy. After some deliberation we decided to use the following command combination:

```
grep -lR <exp> | xargs sed -i <include>
```

The explanation of each part of the command is as follows:

- `grep -lR`: grep looks for the pattern that is passed as an argument (in our case cstring). With the -R flag we indicate that a recursive search must be made for all the files in the directory, and with -l we indicate that it only returns the names of the files.
- |: also known as pipe, to pass the output of a command as an input for the following command.
- `xargs`: takes the list of names of files generated by the grep and passes them one by one to sed.
- `sed -i`: used for in-place file editing (with the -i option).

Once a solution had been found for this problem, the rest of the process did not present many inconveniences.

For the pipeline configuration there was only one small problem, again related to the PLEXIL source code. In this case, the difficulty came from how the results of the tests were reported, since although when there was an error, it was displayed on the console, no type of error was thrown (such as an exit 1 for example). Since there were no errors detectable by the execution, the pipeline always provided a correct result, regardless of the actual result of the tests. As, again we could not modify the source code, we had to look for an alternative solution.

The choice was the simplest, dump the test results into a plain text file and create a bash script that launches from the pipeline, and throws an exit 1 if it finds an error in the results file. As a final note, say that the execution of this pipeline is even faster than that of PLEXIL5, around 30 seconds of execution on average.

6. Testing

It has already been mentioned in other sections of the paper that there is a test suite for PLEXIL5. The tests could be divided into three parts, which would coincide with the execution steps of the PLEXIL5 pipeline: unit tests for the translator, unit tests for the interpreter and acceptance tests involving the whole system.

In this case, unit tests look like any other unit test in different languages. That is, tests consist of an input value, a method that handles that input and an expected result.

For the case of the translator the tests are quite simple if we talk about plans, they have an XML input that represents an isolated information extracted from a PLEXIL plan, for example the declaration of an array, and as expected value it has the expected representation in relation to that array. In this section of the tests, apart from adding tests as new functionalities have been introduced, we have also introduced tests on elements that already existed in the translator and were not being tested. In addition, a small refactoring has been done to have the executor methods isolated from the tests and thus have the possibility of being separated in different locations, since at first all the tests were in the same file.

The tests on the script translator are slightly different and also more comprehensive. As previously explained, this part of the translator is divided into two distinct parts, parser and pretty print, so its tests cover both functionalities separately. The parser tests take XML as input and have as expected result the intermediate representation, and the pretty print tests receive that intermediate representation and expect the final representation; and as it is logical, they have two different executors one for each part of the translator. At the level of things introduced by us in this section the contribution is much more discreet since we have limited ourselves to introduce tests for the functionalities created at that moment. However, this approach is more useful and more detailed because it helps us to specify the errors in a much more efficient way.

The unit tests of the interpreter do not have much to highlight either, they invoke the interpreter with an expression and have as expected result the rewritten value that they expect to get from that input. The most interesting thing about these unit tests are those related to state transitions since these are the most complete tests and those

that take more complex expressions as input, that is, they are the ones that most closely resemble what the interpreter performs when running a PLEXIL plan. These receive a hypothetical node of the PLEXIL tree with all its conditions, status, environment, interface, and from there they have to transition to the expected state according to the values contained in those variables.

The acceptance tests have not been modified or expanded during this work, but they are of great importance to guarantee the incorporation of new functionalities. These tests, being acceptance tests, have the objective of testing the entire system, and for this reason they have a workflow similar to that of the executor script explained in Sections 4.1 and 5.1, in other words, they have to compile, translate, execute and compare. The aspects covered by these tests are a core functionality that works in accordance with PLEXIL, and they are used to verify that despite having introduced changes the most important core functionality continues to work correctly.

In general, the suite of tests used in PLEXIL5 is very important for its development and despite the fact that they cover a large percentage of it, their degree of utility makes it seem that the number of tests is still limited. They are vital for a correct development of the application and mainly to avoid accidentally introducing errors.

7. Conclusion

In this section I would like to give a more personal point of view about the work done, as well as about the PLEXIL5 project.

In accordance with the objectives set out at first, we can say that the work has been a success since not only have the initial goals been achieved, but also new objectives that arose out of necessity during development were introduced and consequently completed.

As previously mentioned, a large part of what is explained in this report has been developed during a stay in the United States in collaboration with NASA. Therefore, it has meant great growth both professionally and personally, since I have had the opportunity to work with people with a very high professional level who have provided me with new perspectives as well as helped to improve in all aspects that form us as a computer scientist.

If we talk specifically about learned technologies, then we must mention many of those discussed in the report, which despite already knowing them at a basic level have been widely developed due to the work carried out.

From Haskell I have learned to use new libraries as well as to work with much more complex functions than those already known. In Maude I also had a general knowledge, but thanks to this work I have seen some of the peculiarities of the language, as well as its real potential, getting to use it for some certainly complex functionalities. In the use of Docker, I did have a little more experience, but the work has helped me to assimilate concepts and polish certain fringes unknown until now.

Moreover, the creation of pipelines has been learned from scratch, since despite knowing it conceptually, I had never worked on configuring one. I found these to be a fascinating concept due to their usefulness and ease of operation, and I will certainly use them again in my future as a computer scientist.

To finish with the technologies, I would like to mention Git, since even though it is taken for granted that someone at this stage of the degree should know how to use it, this work has revealed many of the hidden functionalities that it offers and the great utility it has if you exploit its potential to the fullest. I have discovered new ways of using

Git that can be very helpful when developing and working with different versions of code.

Regarding PLEXIL5, the system aims to increase the popularity of the PLEXIL concept in space applications. To achieve this, it intends to take the ideas proposed by the original language and maximize their potential, by providing rigorous validation for safety-critical systems.

Despite all this, and as mentioned throughout the work report, PLEXIL5 is still under development, and therefore its current utilization does not offer all the potential that it could deliver once completed.

Even so, the effort made by the entire team in this matter has raised the interest of other teams within NASA that might be interested in using it in the future when it is more stable. So much so, that there are already external people collaborating in the development of the features that are yet to be added. Their main function within the team is to review the changes introduced in order to give feedback on the chosen representation, suggest changes to be made, and in general, gradually use the system to generate real user experiences that are vital to achieve a robust, reliable and highly usable PLEXIL5.

Despite that, working with it has not been easy, as its development has been extended over time and this has caused parts of the code to become obsolete or very difficult to modify, so much so that some sections could even be considered legacy code. Moreover, the fact that it has been under development for so long makes that some functionalities are mixed between the different versions of PLEXIL that have been released over the years. Despite all the negative aspects, and the difficulty of the debugging processes that sometimes lasted several hours, I would say that the experience of working in PLEXIL5 has been very positive as it has given me new ideas and has enriched my skillset in a remarkable way.

If I could decide what I would have done differently from the beginning, I would choose to have planned the solutions in the cleanest and most correct way from the beginning. By this I mean that if in some cases we had tried to redefine everything fully from scratch, without trying to reuse existing legacy code and without taking shortcuts, we would have saved time in the long run as we would not have had to debug those intermediate steps, since in the end we would have arrived at practically the same solution.

7.1 Relation with the degree

The contents of this project are related to different subjects taken throughout the degree. The most obvious ones due to the use of functional languages such as Haskell and Maude would be LTP, MFI and even some aspects of AVD. AVD is also related in the context of testing, as all the foundations of this project are based on that. In relation to testing other subjects like ISW or MES come to mind. MES also introduced a lot of Git concepts that have been useful for this work. Beyond specific subjects, the general engineering skills acquired during the 4 years of the degree also have an influence, as well as things like working with Linux thanks to the numerous subjects that use it in their lab classes; something that I was totally unaware of before entering the degree.

8. References

[1] Erik Seligman, Tom Schubert and M V Achutha Kiran Kumar (2015), “Formal Verification”, Consulted in February 2023.

[2] Rogers, R.O. (2004). Acceptance Testing vs. Unit Testing: A Developer’s Perspective. In: Zannier, C., Erdogmus, H., Lindstrom, L. (eds) Extreme Programming and Agile Methods - XP/Agile Universe 2004. XP/Agile Universe 2004, Consulted in March 2023.

[3] V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso (2005), “Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences”, Consulted in January 2023.

[4] V. Verma, A. Jónsson, C. Pasareanu, M. Iatauro (2006), “Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations”, Consulted in January 2023.

[5] “PLEXIL Wiki”. https://plexil.sourceforge.net/wiki/index.php/Main_Page, Consulted in January 2023.

[6] T. Estlin, A. Jonsson, C Pasareanu, R. Simmons, K. Tso, V. Verma (2007), “Plan Execution Interchange Language (PLEXIL)”, Consulted in February 2023.

[7] Ádrián Riesco (2018), “Model Checking Parameterized by the Semantics in Maude”, Consulted in March 2023.

[8] “Maude Wiki”, http://maude.cs.illinois.edu/w/index.php/Maude_Overview, Consulted in December 2022.

[9] “Maude Project and Team”, http://maude.cs.illinois.edu/w/index.php/The_Maude_Project_and_Team, Consulted in April 2023.

[10] “PLEXIL5 Wiki”, https://github.com/nasa/PLEXIL5, Consulted in February 2023.

[11] Gilles Dowek, César Muñoz, and Camilo Rocha (2010), "Rewriting Logic Semantics of a Plan Execution Language", Consulted in February 2023.

[12] Chacon, S., & Straub, B. (2014), "Pro git", Consulted in April 2023.

[13] Alexey Zagalsky, Joseph Feliciano, Margaret-Anne Storey, Yiyun Zhao, and Weiliang Wang (2015), "The Emergence of GitHub as a Collaborative Platform for Education", Consulted in April 2023, https://doi.org/10.1145/2675133.2675284

[14] Shahin, Mojtaba & Ali Babar, Muhammad & Zhu, Liming (2017), "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", Consulted in May 2023.

[15] "Are You Cloud Native", https://www.c-sharpcorner.com/article/are-you-cloud-native/, Consulted in May 2023.

[16] M. Clavel, F. Duran, S. Eker, S. Escobar, P. Lincoln, N. Martı-Oliet, J. Meseguer, R. Rubio and C. Talcott (2020), Maude Manual (Version3.0). https://maude.lcc.uma.es/maude30-manual-html/maude-manual.html, Consulted in May 2023

[17] C. Rocha, H. Cadavid, C. Muñoz and R. Siminiceanu (2012), "A Formal Interactive Verification Environment for the Plan Execution Interchange Language.", Consulted in March 2023.

[18] "Navigating XML with Cursors", https://xmlbeans.apache.org/docs/2.0.0/guide/conNavigatingXMLwithCursors.html, Consulted in April 2023.

[19] P. Hudak, J. Hughes, S. P. Jones, P. Wadler (2007), "A History of Haskell: Being Lazy With Class", Consulted in March 2023.

[20] "Haskell Wiki", https://wiki.haskell.org/All_About_Monads, Consulted in March 2023.

[21] T. Schmorleiz (2007), "Haskell Programming Technologies", Consulted in April 2023.

[22] F. García, "Service for the Management of Clinical Practice Guidelines in XML", Consulted in April 2023.

[23] P. Wadler (1997), "A Prettier Printer", Consulted in March 2023.

[24] "Tasty Documentation", https://hackage.haskell.org/package/tasty, Consulted in April 2023.

ANNEX

SUSTAINABLE DEVELOPMENT GOALS

Degree to which the work relates to the Sustainable Development Goals (SDGs).

Sustainable Development Goals		High	Medium	Low	Not Applicable
SDG 1.	**End Poverty**				X
SDG 2.	**Zero Hunger**				X
SDG 3.	**Good health and well-being**				X
SDG 4.	**Quality education**				X
SDG 5.	**Gender Equaility**				X
SDG 6.	**Clean Water and Sanitation**				X
SDG 7.	**Affordable and clean energy**				X
SDG 8.	**Decent work and economic growth**				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 production**				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.	**Partnership for the goals**				X

Reflection on the relationship of the TFG/TFM with the SDGs and with the most related SDG(s).

Work focused on testing and Formal Methods play an important role in achieving the Sustainable Development Goals (SDGs), specifically Goal 9 and Goal 11. These goals focus on building resilient infrastructure, promoting sustainable industrialization, and making cities more inclusive, safe and sustainable. In the following, we will explore how testing and formal methods related to software contribute to these goals.

In the context of Goal 9, testing and formal software-related methods are fundamental to building resilient infrastructures. Software plays a crucial role in the operation of various infrastructures, such as transport, energy and communication systems. By using rigorous testing techniques, such as load and performance testing, security testing and interoperability testing, potential flaws and vulnerabilities in software that could affect the reliability and resilience of infrastructures can be identified. This allows problems to be corrected before crisis situations occur, such as power outages or public transport failures. In addition, formal methods related to software, such as formal verification and formal modelling, provide rigorous approaches to ensure software quality and reliability.

Regarding Goal 11, software security testing is essential to ensure the safety of cities in an increasingly digital environment. Security testing identifies vulnerabilities in software that could be exploited by cyber criminals, thus protecting critical systems and urban infrastructure from potential attacks. Furthermore, formal methods related to software help to understand and mitigate the risks associated with the use of software in urban environments and even serve to develop more autonomous cities, as can be exemplified by Paris metro line 14, where formal methods were instrumental in creating the first fully automated railway line in Paris.

Additionally, in a world that is becoming more digital, software security testing is crucial to ensuring the safety of cities. Critical systems and metropolitan infrastructure are shielded from prospective assaults by security testing, which finds software flaws that cybercriminals may exploit. Additionally, formal software-related techniques like modeling and risk analysis aid in the understanding and reduction of the dangers connected to the usage of software in urban settings.

In reference to the other SDGs, I do not think it would be appropriate to make a direct link of any kind to this work, either high or low. However, it could be said that as it is closely related to the development and future of technology, this work is indirectly linked to many other SDGs, due to the fact that technology is currently used in practically every area of society.

In conclusion, this work is directly relevant to those SDGs that are more focused on technology development itself, i.e. 9 and 11. On the other hand, and somewhat indirectly, it also touches on those that could be achieved with potential new technology-driven developments that are covered in this work.