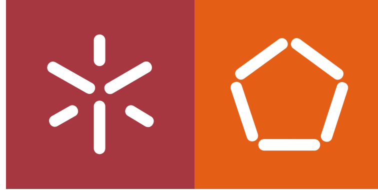
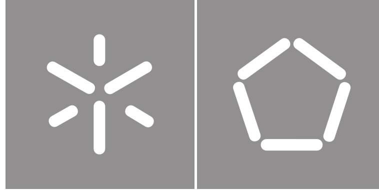


Universidade do Minho
Escola de Engenharia

Rui Filipe Moreira Mendes

A new models editor for the IVY Workbench

October, 2022



Universidade do Minho
Escola de Engenharia

Rui Filipe Moreira Mendes

A new models editor for the IVY Workbench

Master Thesis
Integrated Master's in Informatics Engineering

Work developed under the supervision of:
José Francisco Creissac Freitas Campos

October, 2022

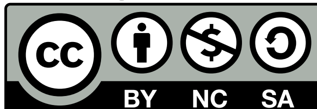
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en

Acknowledgements

First of all I would like to thank my instructor Professor José Creissac, for all the support given throughout the development of this Thesis, as well as Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC TEC) for providing me with the perfect conditions to develop my work.

Secondly, I'm thankful for all the support I got from my family and girlfriend.

Also a special acknowledgement to my grandfather, without him I would not have been able to start my Master.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

________________________, ______________________________________

(Place) (Date)

____

(Rui Filipe Moreira Mendes)

"You cannot teach a man anything; you can only help him discover it in himself." (Galileo)

Resumo

Um novo editor de modelos para a IVY Workbench

Para que as interfaces de sistemas críticos possuam um nível de qualidade que permita o seu uso em segurança, devem passar por um processo rigoroso de análise. A verificação formal de interfaces é uma das formas de realizar essa análise. Para tal, é importante que os desenvolvedores dessas interfaces consigam editar e criar os modelos que acharem mais adequados para as suas interfaces. Tanto os desenvolvedores mais experientes como os menos experientes. A Ivy Workbench é uma ferramenta que permite descrever o funcionamento das interfaces e verificar propriedades sobre o seu comportamento, de forma a identificar potenciais problemas na interação. Deste modo, fornece informação relevante para os desenvolvedores que utilizem o Ivy, para que se possa melhorar o software sem ter de necessariamente passar por um processo de teste manual longo e exaustivo.

O atual editor do Ivy é difícil de manter e não fornece ajuda suficiente nem guia novos utilizadores adequadamente. Por isso, é necessário que haja uma melhor forma de editar os modelos na linguagem Model Action Logic (MAL), a linguagem de programação da Ivy Workbench. O objetivo desta dissertação é construir uma solução que permita que todos os tipos de desenvolvedores consigam construir os seus modelos através de orientações do próprio editor. É bastante desafiante desenvolver uma solução deste gênero, que permita alcançar o nível de apoio pretendido, dado que precisamos de ter em conta com o que é que os utilizadores estão mais confortáveis e quais as ferramentas que usam com maior regularidade, para que seja possível desenvolver uma solução o mais abrangente possível.

Para que se concretize o principal objetivo, enquanto também se alcança o máximo número de utilizadores, optou-se por desenvolver uma extensão de VS Code. Trata-se do editor de código mais utilizado e fornece várias ferramentas para desenvolvedores de extensões, assim como uma vasta documentação. É possível tirar partido das funcionalidades que esta ferramenta já apresenta, típicas de um Integrated Development Environment (IDE) comum, que nos permitem criar novas formas para os utilizadores da Ivy escreverem modelos MAL, e fazendo isso, aumentar a sua produtividade.

Depois da extensão estar concluída, é expectável que esta solução seja mais fácil de manter no futuro, e mais utilizadores achem esta nova solução menos complexa para trabalhar, levando a que estes se sintam mais satisfeitos a utilizar a ferramenta e a própria linguagem, ajudando assim o crescimento da utilização da Ivy Workbench assim como da qualidade do software.

Palavras-chave: MAL, Ivy Workbench, Utilizadores, Guia, VS Code, Verificação

Abstract

A new models editor for the IVY Workbench

In order for the interfaces of critical systems to have a quality level of security that allows for its safe usage, they should be subject to rigorous analysis process. Formal verification is one of the alternatives to perform that analysis. So, it is important that developers can edit or create the models which they find the most suitable for their interfaces. Both the most experienced developers as well as the least ones. The Ivy Workbench is a tool that allows for the modeling of user interfaces, and for properties about the interface behaviour to be verified, so that potential problems in the interaction can be identified. By doing this, it provides information for the developers who use Ivy, so that their software can be enhanced without having to perform extensive manual testing.

Ivy's current editor is difficult to maintain, and does not provide enough help nor guidance to inexperienced users. So, there is the need of a better way for users to write in the MAL language, the modeling language of the Ivy Workbench. The goal of this thesis is to build a solution that allows every level of developer to build their own models based on guidance by the editor itself. It can be challenging to put together an editor or code editor extension that would allow such goal, because there is the need to consider what the users are comfortable with, and what their most often used tools are, in order to build the more embracing solution.

In order to achieve the main goal, while also reaching as many users as possible, it was considered that the best option would be to develop a VS Code extension. VS Code is the most widely used code editor and provides various tools for extension developers, with a vast documentation about their development. Also, it is possible to make use of the features this code editor already presents, common amongst the most used IDE, to build new ways for the users to write MAL, and in doing so, increase their productivity.

After the extensions is completed, it is expected that this new solution will be easier to maintain in the future, and that more users will find it less complicated to work with, leading users to get more satisfied when using the editor and the language itself, thus helping the growth of Ivy Workbench as well as the quality of the software.

Keywords: MAL, Ivy Workbench, Users, Guidance, VS Code, Formal Verification

Contents

List of Figures xiii

List of Tables xv

List of Listings xvi

Glossary xvii

Acronyms xviii

1 Introduction 1

1.1 Motivations . . . 2

1.2 Objectives . . . 2

1.3 Document Structure . . . 2

2 Tools for the formal verification of user interfaces 4

2.1 Formal verification of user interfaces . . . 4

2.2 The example . . . 7

2.3 ADEPT . . . 8

2.4 CIRCUS . . . 10

2.5 PVSio-Web . . . 10

2.6 Action Simulator . . . 11

2.7 Ivy Workbench . . . 12

2.8 Discussion . . . 15

3 Ivy Workbench new editor 16

3.1 Current state of Ivy Workbench editor . . . 16

3.2 Requirements . . . 17

3.3 Possible implementation approaches . . . 19

3.3.1 Building a web based solution . . . 19

3.3.2 Extension method . . . 20

3.3.3 Building a new editor plugin 23
3.4 Decision 23

4 Development of the VS Code Extension 25
4.1 Overview of developing a Language Support Extension 25
4.1.1 Package.json 25
4.1.2 Syntaxes folder 26
4.1.3 Language Configuration 27
4.2 Technology and implementation 28
4.2.1 Syntax Highlight 28
4.2.2 Semantic Highlight 29
4.2.3 Diagnostics 33
4.2.4 Quick Fixes 33
4.2.5 Hover Information 38
4.2.6 Go to definition 39
4.2.7 Snippets 39
4.2.8 Code Completion 40
4.3 Additional Functionalities 41
4.3.1 Web Views 41
4.3.2 Axioms analysis 44
4.3.3 Properties Creator 45
4.4 Results 48

5 Usability testing 49
5.1 Research questions 49
5.2 Procedure 49
5.3 Participants 50
5.4 Material 50
5.5 Description of the test 51
5.6 Data Collecting 51
5.7 Demographic of participants 52
5.8 Observations during the tests 52
5.9 Open questions' answers 53
5.10 User experience questionnaire results 53
5.11 Result analysis 54
5.12 Answers to research questions 54
5.13 Threats to validity 56

6 Conclusion **58**
6.1 Results 58
6.2 Future work 60

Bibliography **61**

Annexes

I extension.ts **65**

II mal.tmLanguage.json **68**

III language-configuration.json **77**

IV Air Conditioner MAL example **79**

List of Figures

1 Integration of modeling in development as in [4] 6
2 State machine diagram 8
3 Interactive Cooperative Objects (ICO) like language diagram 11
4 Emucharts Diagram example 12
5 Action Simulator table example 13

6 Ivy's Workbench current code editor 17

7 Process of building the tokens. 30
8 Example of errors fixable through quick fixes 33
9 Example after the fix for the wrong type, when the type is a number. 34
10 Example after the fix for a non declared action. 35
11 Example after the fix to change an incorrect type. 36
12 Example after the fix to a duplicate declaration. 37
13 Example after the fix for a non declared attribute. 37
14 Example after the fix for a member not present in an enumeration. 38
15 Example of the extension present when **per** is hovered. 39
16 Web view interface for the action determinism functionality. 44
17 Ivy Workbench's property creator. 45
18 Extension's property creator. 46
19 Choose interactor drop down. 47

20 Results obtained via User Experience Questionnaire (UEQ) data analysis tool, where the blue represents the answers given to the Ivy editor and the red represents the answers given to the VS Code editor. 54
21 Statistics from the UEQ analysis tool, where STD means standard deviation and N is the number of tests. 54
22 Results obtained via UEQ data analysis tool, where the participants started with Ivy Workbench. 55
23 Statistics from the UEQ analysis tool, where the participants started with Ivy Workbench. 55

24 Results obtained via UEQ data analysis tool, where the participants started with the VS Code editor. 56
25 Statistics from the UEQ analysis tool, where the participants started with the VS Code editor. 56
26 Difference in UEQ results for both tools. 56

List of Tables

1 Automation Design and Evaluation Prototyping (ADEPT) example 9

List of Listings

4.1 Comment pattern matching example inside the grammar file. 27
4.2 Comments entry in the language configuration file. 28
4.3 Snippet for creating a new interactor 39
4.4 View Containers 42
4.5 View inside the container 42
Chapters/Anexos/extension.ts 65
Chapters/Anexos/mal.tmLanguage.json 68
Chapters/Anexos/language–configuration.json 77
Chapters/Anexos/AC.txt 79

Glossary

Emucharts Diagram Emucharts diagram is the representation of an extended state machine in the form of a directed graph composed of labelled nodes and transitions xiii, 10, 11, 12

JMenu The object of JMenu class is a pull down menu component which is displayed from the menu bar. 23

JPanel JPanel, a part of the Java Swing package, is a container that can store a group of components. 23

Acronyms

ADEPT Automation Design and Evaluation Prototyping xv, 8, 9

CIRCUS Computer-aided-design of Interactive, Resilient, Critical and Usable Systems 10

CTL Computer Tree Logic 12, 15

HAMSTERS Human-centered Assessment and Modeling to Support Task Engineering for Resilient Systems 10

ICO Interactive Cooperative Objects xiii, 10, 11

IDE Integrated Development Environment viii, ix, 19, 20, 21, 22, 23, 59

INESC TEC Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência v

JSON JavaScript Object Notation 22, 26, 28, 39

MAL Model Action Logic viii, ix, 1, 2, 12, 13, 14, 15, 16, 17, 18, 21, 22, 23, 25, 26, 27, 28, 30, 38, 40, 49, 51, 58, 59

PetShop Petri Net Workshop 10

PPS Propositional Production System 11

SWAN Synergistic Workshop for Articulating Notations 10

UEQ User Experience Questionnaire xiii, xiv, 50, 51, 52, 53, 54, 55, 56, 60

UI User Interface 2, 3, 6, 7, 8, 9, 11, 12, 14, 15, 23, 41, 43, 56, 58

UML Unified Modeling Language 15

UX User Experience 6, 7, 46, 49, 53, 54, 55, 56, 58, 60

WYSIWYG What you see is what you get 19, 20

XML Extensible Markup Language 22

Chapter 1

Introduction

Software analysis and testing are both important parts of the development process, as the analysis assures that all work was done right, and the purpose of the program has been achieved. In other words, it assures the quality of the code written, as well as guaranteeing that some pre established properties are respected throughout the software, to build a robust and useful system [1], especially in critical environments.

The experts of such critical systems know how their systems function and the expected behaviour of the interfaces used in those systems. However, trying to write these rules and behaviours in a manner that allows for their formal verification, can be a complex task.

Even if complex, having the capability to formally prove properties about a critical interface is a relevant part of their development. So, it is important to have a way for domain experts to write and create these formal models. Writing such formal models, will allow their creators to argue that the interface they modelled satisfies a key set of requirements, as early as in the design phase of the software development life cycle [19]. This part of the development aims to lower the risk of system failures and vulnerabilities of critical systems.

The Ivy Workbench [23] is a tool that allows for these models to be written in MAL, and provides analysis functionalities to the users. The Ivy Workbench was developed in Java, using a plugin-based architecture which has been evolving throughout the years. While the technical solution was sensible at the time, it has become increasingly hard to maintain and to add new functionalities. At the same time, new development environments have emerged that might be a better solution. The goal of this thesis is to find an alternative to develop a new code editor that would facilitate the interfaces modeling process.

1.1 Motivations

Modeling software is an important part of the development process, especially for critical software. It allows for an abstract representation of a software solution that can be discussed and formally analyzed by tools such as Ivy Workbench, without the need of a concrete implementation. This way, changes can be made to the design in a simpler manner and the developers can ponder over different design solutions before deciding on a final one.

Even though this is applicable to all kinds of software, we will be focusing on the modeling of user interfaces. In this field, modeling can be even more important, since user interfaces are the first thing the users see and interact with. We need to make sure they act as we want them to, so they do not end up in any state that would offer a negative experience to the the users or a complete system failure.

When trying to prove properties of user interfaces, the IVY Workbench can be useful. The tool was developed to provide a way to formally analyze and find possible errors in User Interface (UI) models. However, the current modeling process is complicated to use for people without previous background of the MAL language. Aside from that, the technology used to develop Ivy is based on legacy solutions, and so the software must evolve to match current demand and standards, or it will inevitably become less useful [47]. To improve this modeling phase we need to improve the code editor of the Ivy Workbench so that new users can write and understand MAL, in order to be more productive and end up with safer software.

Hence, it becomes clear that building a better editor for the IVY Workbench would help all people interested in modeling their UI ideas.

1.2 Objectives

The main objective of this master's thesis is to develop an up-to-date code editor that can be maintained more easily and find a way to facilitate the interaction between users and Ivy's workbench editor, through an improved editor or an extension for an existing and commonly used code editor, like VS Code. These objectives can be separated as followed:

- Identify Ivy Workbench current editor's problems and present possible alternatives to solve them.
- Research about different alternatives for building the new editor and present a solution.
- Develop and implement the new solution for the code editor.
- Evaluate the developed solution in comparison to the old one.

1.3 Document Structure

This dissertation document is separated in six different chapters:

- **Chapter 1 - Introduction**

  This chapter was responsible for giving an introduction about the theme of the thesis. As well as explaining the problem that we are addressing, along with the motivations and the objectives for the writing of the thesis.

- **Chapter 2 - Tools for formal verification of user interfaces**

  Chapter 2 will present the state of the art to the reader, by providing information about current existing solutions to verify user interfaces, in order to explain the different approaches these solutions take to allow their users to build the models of the UI.

- **Chapter 3 - Ivy Workbench new editor**

  In Chapter 3, we make the decision about how the new Ivy Workbench editor should be developed, while also acknowledging the state of the current editor to find requirements that should be set for the new one.

- **Chapter 4 - Development of a VS Code Extension**

  In Chapter 4, the steps required for developing a VS Code Extension are presented to the reader, along with the explanation and detailing of files already present in the extension built, as well as how functionalities work.

- **Chapter 5 - Usability Testing**

  Chapter 5 is where we will explain the usability test performed on real users, in order to have feedback about the new develop solution, as well as, establishing a comparison with Ivy's current editor.

- **Chapter 6 - Conclusion**

  This chapter will be the one where we evaluate the work done and propose ideas for future work that can be done to further enhance the new model's editor.

Chapter 2

Tools for the formal verification of user interfaces

The idea of developing a tool that would support developers in creating models for their interfaces, is not new. There are already tools developed for that purpose. In this chapter, we describe in further detail the integration of the model design process into the development phase of software, and some of the different tools that were developed to support it.

2.1 Formal verification of user interfaces

As seen in Figure 1, integrating formal modelling and verification into interactive systems development progresses through five phases [4]:

- **Identifying the artifact**

  The first step in this process is to decide which part of the system we want to model, so that the focus can be set to a specific area of concern. Establishing requirements can help in the decision of what needs to be modeled.

  To describe these requirements, a clear list of goals should be provided, since knowing precisely what the user interface aims to achieve also facilitates the development of a design solution for such specific goals.

  The priorities of these requirements should also be settled so that the most critic ones are dealt with first, in case there is a time limit to the design phase.

  It is also important to have some criteria that can be used in order to quantify the quality of the solution built to the requirements [22].

- **Building the model**

  Building a formal model is the next step to take into consideration. This model should take into consideration the previously established requirements, so that a set of properties can be proved around those requirements and the initial model itself [20].

  In order to design an interactive system, the developer needs to know the characteristics of the users they are expecting, like experience, skill, age and common preferences, so that they can build the most fitting solution for their users. It is also important to determine which task the users are going to do, as well as the frequency and duration of such tasks, so that if a task is done multiple times the user will have a way of doing it as fast as possible and without much effort [22]. Where and how the users will interact with the interface is also important, since that determines how the user will have to interact with it. This design must then be expressed as a model, to support analysis.

  One problem with the modeling step, is the knowledge about formal methods that the developers must have, in order to build an accurate enough model. In many cases, domain experts, and even developers, interested in developing verified, safe and robust software, will not have the required formal methods expertise.

- **Verify Properties**

  This is the step where properties are specified and thus verified, typically using a model checker or a theorem prover.

- **Analyze Results**

  Different types of tools will provide different feedback, in case of failure. In general, to analyze the results given by the verification, a certain level of knowledge is required. The messages the tool sends are expressed in technical terms, and are not that easy to interpret for people without proper training.

  Assuming that developers can understand the output of the verification, they can then proceed to verify if the properties held true or if they failed. In the later case, the developer can then change the model accordingly so that the property can be verified.

- **System Design**

  This step is where the developers actually change the designed model in order to fix the properties that did not hold true, or for adding or removing some new functionality from the interface.

  After those changes, the process must be iterated again, and these reiterations will be done until a desired design is reached, for both clients and developers. Sometimes trying to understand what is wrong with the model can take several iterations of this process. However, the aim is that it will be less costly than modifying the whole software later on.

As we can see above, the process can be complex, however it assures the quality and safety of the design. In this thesis, we will be focusing on the modeling phase of the process.

The goal is to build an alternative to the current Ivy Workbench's code editor so that modelers can more easily understand and express their models, in a better UI which ultimately leads to a better User Experience (UX). The goal is to achieve this even for those without extensive previous knowledge of formal modeling, while also responding to the need for updating the current Ivy software.

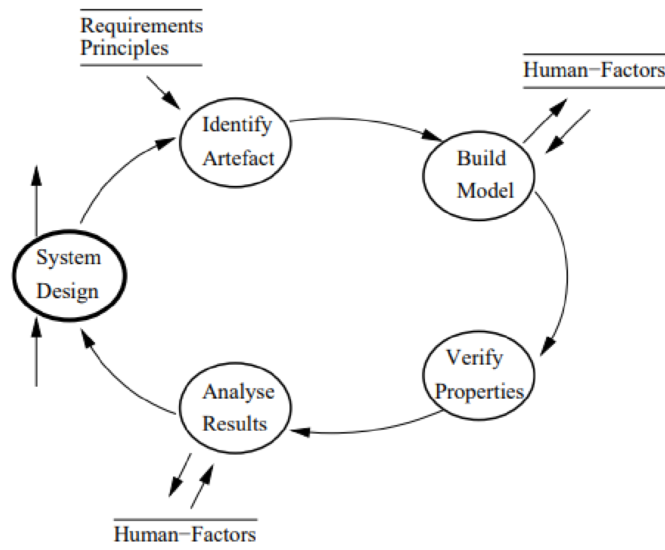


Figure 1: Integration of modeling in development as in [4]

The process of interface development is often seen as something that can be done after all the business logic of the software is over, and in doing so, completely separate the development of the business logic from the user interface. As said in [5], the interface is perceived by many as "simply a passive information transmission layer between the user and the 'application'", this approach more often than not leads to bad user experiences since the software was not developed thinking about how users would interact with it.

The user interface analysis phase is crucial to give the software the robustness and some correctness needed in critical environments, because errors occur more often than not, especially if we are introducing interaction between humans and machines. If it is possible to know for certain how a machine will respond to certain inputs, the same cannot be said about how humans will interact with a given user interface.

Even though it is possible to make some predictions about the behavior of the users, these are not certain and scenarios where the UI breaks may happen. For these reasons it is important to find a way to verify a given set of properties that need to stay true throughout the entire user experience with the user interface. For example, proving that some undesirable state of the UI cannot be reached, or that user actions can always be undone.

In specific, formal analysis can process the models developed in order to find problems that can disturb

the normal functionality of the software. The problems are found if some property that the developer has established for that interface proves itself to not hold true in a given scenario. That scenario is then shown to the developers, so that, with the information about the situation that lead to the error, they can more easily identify the source of the problem.

In some cases, software testing will not be enough, because it can not prove that the developed software does not have errors, and for those cases formal verification can be used.

Formal Verification allows for a mathematical proof of properties. These properties can be written using a logical formula in an appropriate language that supports formal verification, then the verification tool checks if the model built to represent the interfaces obeys to those properties and if so it can be said that the property is true beyond doubt [34], the recognition of formal verification as a reliable way of evaluating software is growing, as seen in [33], where the DO-178B standard allows for aircraft software to be certified using this method of evaluation. It also allows for automatic proof of properties, without the need of sample inputs that can be hard to find. Most times this formal verification is faster (for the same level of coverage) than that done by testing, since verification guarantees full coverage of the system behaviour.

The Ivy Workbench is a tool that allows for the formal verification of UI. Experience has shown that utilizing the tool can be useful [21], but its use requires expertise in formal modeling and verification. Making the tool more accessible would allow its use by domain experts and increase its impact. The model editor was identified as a bottleneck, but the current implementation is not easy to update.

In order to have a better idea of alternatives and what already exists, the next sections focus on the exploration of tools that have almost the same goals as Ivy does. The tools in these sections have been developed to explore how to formally characterize and analyze interfaces. Such tools aim to build safer and more intuitive interfaces that, in the end, can lead to a better UX. In order to better understand the existing tools, an example will be used throughout this chapter, and represented in each tool.

2.2 The example

The example will be a simple one. It will revolve around a gate with two sensors and one remote control. The idea is that, if the gate is closed and the remote is pressed, then the gate will start to open until it triggers the opened sensor. The same applies if the gate starts opened, but in the other direction. If the remote is pressed between the opened and closed states, then the gate will stop at the point it is at. Once the remote is pressed again, the gate will invert its direction. The state machine of this example can be seen in Figure 2.

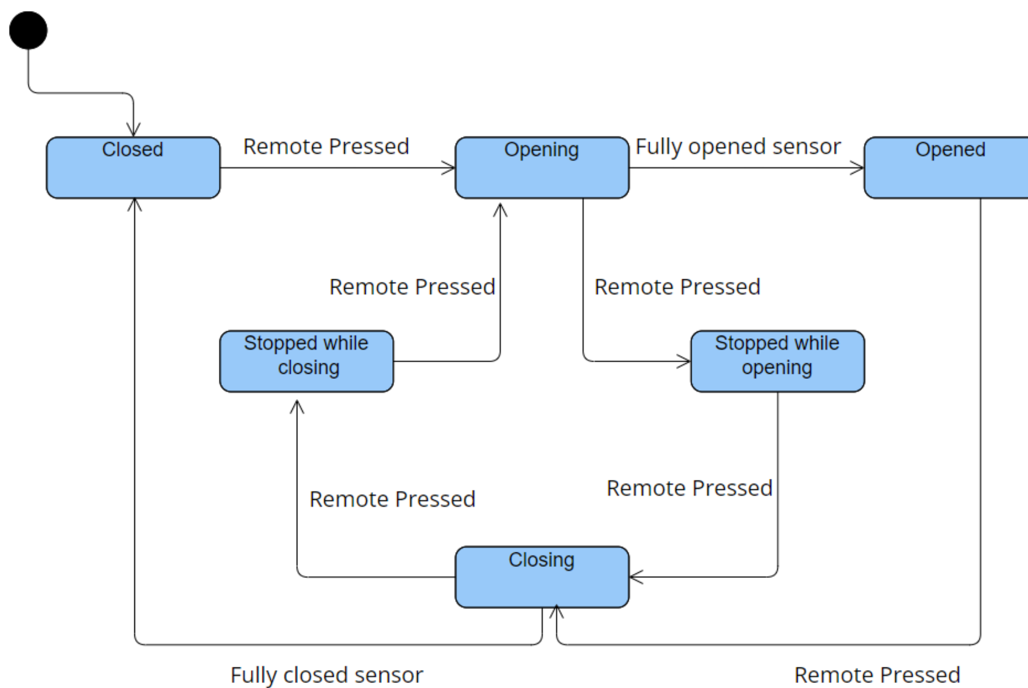


Figure 2: State machine diagram

2.3 ADEPT

ADEPT [17] is a tool developed by NASA Ames, and it aims to find possible miscommunication errors in the interaction between the user and the interface, providing a tool to integrate testing and analysis of such interactions. ADEPT was developed so that it could be used by a large range of users, from those with a vast knowledge of programming to those with little to no experience. This was done in order to give developer teams a link that would facilitate the communication between all sectors to achieve the best results possible. By combining a User Interface Editor with a Logic Editor it is possible to achieve a reliable prototype.

The User Interface Editor allows the developer to create a prototype like a graphical application would and make it interactive, the components that construct this UI are built as Java graphical widgets so that, after implementing the logic in the Logic Editor, ADEPT builds a model compiled into Java to behave like previously defined, and allows for interaction with the interface built [3].The Logic Editor is the part of ADEPT that separates it from any other design application.

The operational Procedure Table method [43] is used so that the developer can describe the expected behavior of the interface, and thus define how the state of the software should act upon certain inputs. A table in ADEPT is represented by a matrix where the columns are the possible states of the UI and the rows are separated into the input and output of a given state. The input and output can be further detailed to contain actions and variables that impact the UI.

A table that could represent the gate example in ADEPT is presented in Table 1. The idea of this matrix

is to describe the relations between inputs and outputs in a given state, these states are represented by numbered columns, starting at 0 (zero). For example, in Table 1 we can see that in state 0 (initial state) the gate starts as closed, and the available action is *remote_pressed*. If such action occurs in state 0, it is possible to see, in the output rows, that the state will become *opening* and the *is_opening* variable is set to true, thus changing the state of the UI.

Table 1: ADEPT example

		0	1	2	3	4	5	6	7
Inputs									
gate_states									
	opening		●					●	
	closing					●			●
	opened						●		
	closed	●							
	stopped			●	●				
is_opening									
	true			●					
	false				●				
Actions									
	remote_pressed	●	●	●	●	●	●		
	fully_opened_sensor							●	
	fully_closed_sensor								●
Outputs									
gate_states									
	opening	●			●				
	closing			●			●		
	opened							●	
	closed								●
	stopped		●			●			
is_opening									
	true	●	●		●				●
	false			●		●	●	●	

Although this tabular language is simple, this is achieved at the cost of expressiveness. For example, all state attributes in Table 1 are Boolean, and complex logical expressions involving conjunctions, disjunctions, implication, etc. are not easy to express.

2.4 CIRCUS

Computer-aided-design of Interactive, Resilient, Critical and Usable Systems (CIRCUS) [2] is a development environment meant to be used by technical people, like engineers and designers. The main goal of CIRCUS is similar to all other interactive testing and analysis tools: to help developers in both the design and development of an interactive system. It achieves this goal by separating itself into three components:

- **Formal verification** - In what regards formal verification, CIRCUS offers the Petri Net Workshop (PetShop) tool, which allows for development in all matters related to the system model. It does so by using the ICO notation. Since ICO is not the focus of this thesis, it can be resumed as a notation that allows for the specification of interactive systems based on object-oriented programming and Petri nets [38]. Figure 3 show the example from Section 2.2 in the ICO language. Each possible state is represented using an ellipse and the actions are inside rectangles. It is possible to see, for example, that when the current state is closing, then there are two possible next states for the gate, depending on the next action. If the remote is pressed the gate stops closing, but if *sig1* (fully closed sensor) is emitted then the gate is closed.

- **Conformity of users' tasks and interactive systems** - In order to get coherence between how the users behave to achieve their goals and the interface provided to them to do so, a notation is needed to formally describe the users' tasks. These can be described, in a hierarchical task model, as a main goal that can be further split into sub-goals. Sub-goals can be combined by expressing temporal relations between them. This can be done by using the Human-centered Assessment and Modeling to Support Task Engineering for Resilient Systems (HAMSTERS) notation provided by CIRCUS.

- **Co-execution and conformity of models** - Synergistic Workshop for Articulating Notations (SWAN) is a tool that allows for the co-execution of the HAMSTERS tool and the ICO system model by editing the relations between them, allowing for an analysis of the conformity between the system and the task. The co-execution can be done manually or partially automated further augmenting productivity.

2.5 PVSio-Web

PVSio-web [36] is, as the name suggests, a web-based tool that allows for the creation of prototypes based on PVS specificiations. Both storyboard-based prototypes and high-fidelity prototypes are supported. The specification of the properties is done using the specification language of PVS, which can be written manually or generated from an Emucharts Diagram, as seen in Figure 4.

In an Emucharts Diagram, transitions are labeled with events, pre-conditions and effects. Pre-conditions are represented between square brackets, and must hold for the transition to fire in response to the event.

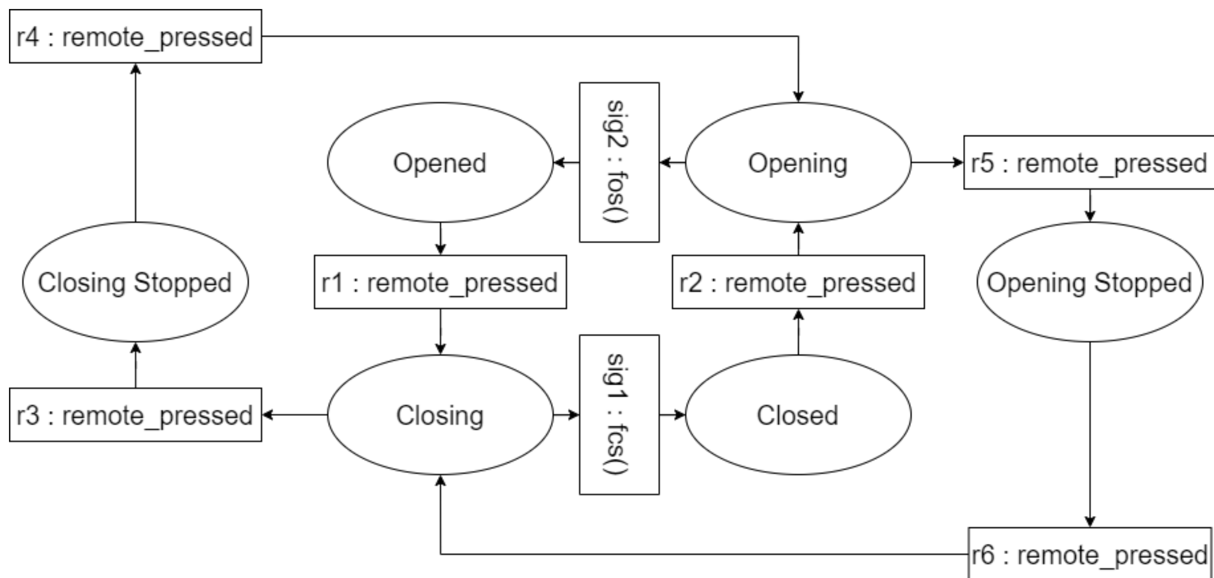


Figure 3: ICO like language diagram

Effects, the executable part of the transitions, are written between curly braces. To each transition there is an associated interface event which comes before both the preconditions and the effects.

In Figure 4 it is possible to see an Emucharts Diagram for the example proposed earlier. Each of the states of the Emucharts Diagram corresponds to a state of the gate, and the transitions correspond to the possible events that can occur in any given state. For example, if the gate is in the Opening state, then it has two possible next states: Opened, through the *fully_opened_sensor* event, which also sets the *isOpening* variable to false; and *Stopped*, which can be reached via the *remote_pressed* event, while also setting *isOpening* to true (see effect).

PVSio-Web also provides a way to build high-fidelity prototypes by making use of a set of widgets that are provided to facilitate their creation, such as buttons or touchscreen elements [20] which can then be linked to events described in the PVS. In this case, however, some degree of JavaScript programming can be required.

2.6 Action Simulator

Action Simulator [32], is an earlier tool that was developed to dynamically simulate the design without the need to explain low level details in the UI. This approach is mostly inspired by Propositional Production System (PPS) dialogue models [37] that consist in if-then blocks containing both pre- and post- conditions as well as the side effects that occur when the action is executed. In Action Simulator a tabular form is used to describe these conditions. In order to simplify the computation and the writing of this tables, Action Simulator does not register the side effects (i.e. the behaviour of any underlying system), so these must be recorded in another way if needed.

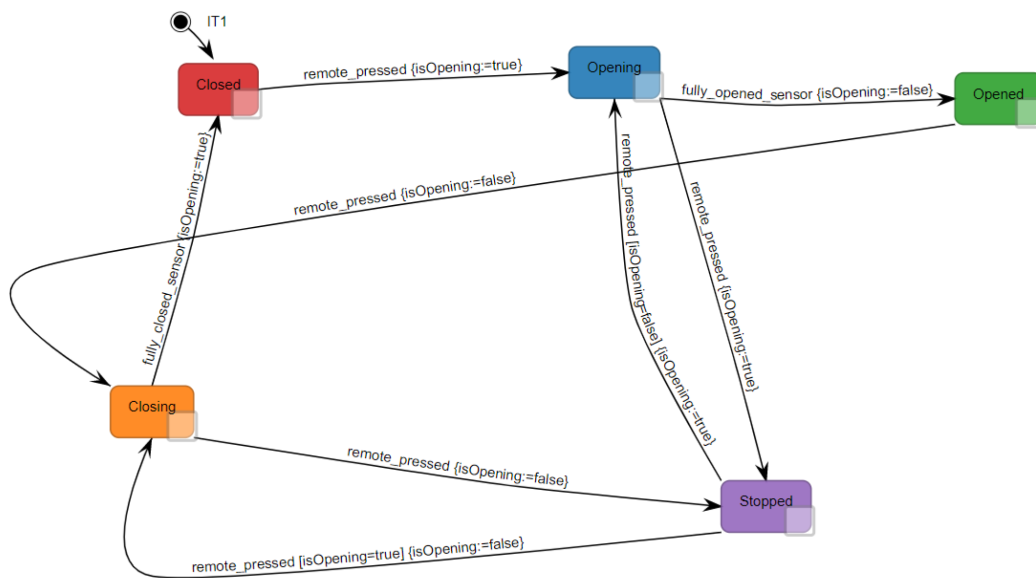


Figure 4: Emucharts Diagram example

This tabular form contains the variables of the state as columns and the possible actions as rows. The idea is that the first row contains the current state of the UI, and for each other row, the variables that affect that action are described in two lines, the first one represents the value that the variable has to have in order for that action to be executable; as for the second line, it states the value of the variable after the action was executed.

This table can be represented in an Excel sheet, where the actions are set as available (if their preconditions are met) with a row of (*) beneath their name. This makes it easier for developers to navigate through all the possible paths of the interface by running the 'Do Action' macro. By being able to navigate through the user interface states, developers can comprehend better the behaviour of the application and theoretically verify if all the possible and impossible paths they previously defined are being followed correctly.

In Figure 5 it is possible to see the representation of the example previously proposed in Action Simulator.

2.7 Ivy Workbench

The Ivy Workbench [23] is the tool we will be focusing on during the rest of this thesis, it uses the MAL interactors language to describe the UI, and Computer Tree Logic (CTL) to describe the properties we want to prove of a given UI. It is separated into four different plugins:

- **MAL editor**

	Opening	Closing	Opened	Closed	Stoped
State	FALSE	FALSE	FALSE	TRUE	FALSE
Start opening gate	FALSE			TRUE	
***************	TRUE			FALSE	
Start closing gate		FALSE	TRUE		
		TRUE	FALSE		
Stop opening gate	TRUE				FALSE
					TRUE
Stop closing gate		TRUE			FALSE
					TRUE
Re-start opening gate	FALSE	TRUE			TRUE
	TRUE	FALSE			FALSE
Re-start closing gate	TRUE	FALSE			TRUE
	FALSE	TRUE			FALSE
Fully opened gate	TRUE		FALSE		
	FALSE		TRUE		
Fully closed gate		TRUE		FALSE	
		FALSE		TRUE	

Figure 5: Action Simulator table example

The MAL editor supports tree editing of MAL interactors models. In order to better understand them, it is useful to think of them, as described by the York group [12], as a way to represent an object that interacts with its environment through some actions, and in doing so makes changes to the state. These changes can then be shown through some form of understandable presentation method so that the user can see what is happening.

These interactors, in Ivy Workbench, are composed of three main parts:

- The state

  The state is defined using structured MAL that allows us to describe it through a set of attributes. Some of them are visible to the user but some are internal. The difference between visible and internal attributes (see rendering below) can be understood by thinking of an interactor representing a coffee vending machine. The user can see the available coffee flavors, but not how much water is left in the machine, and both of these attributes are important for the state, in this case, our interactor.

- The behavior

  A set of actions is also available in an interactor to change the state itself. Again, some of the actions can be available to the user, and some are internal to the system. MAL is used to write the axioms that specify how actions affect the state, thus making it possible to perform formal verification.

- The rendering

> The Ivy tool provides a way to express the visibility of both variables and actions to users through the *[vis]* tag that can be attributed to something in the UI we want the user to be able to interact with.

MAL axioms have three main variants. The first variant, are the axioms that represent the initial state of the interactor, these axioms are represented with the empty square brackets at the start of the expression and then define initial values for the attributes. See the axiom marked with the comment *initial state axiom* in the code below.

The second variant of axioms are the ones responsible for describing how the state of the interactor evolves in response to actions. This is done by setting the action we want to describe inside square brackets. After the action, we describe the value of the attributes after the action has taken place. This is done using the **'** symbol after the attribute name, to make it clear that we are referring to the value of the attribute in the following state. Optionally, we can define a condition before the action. This type of axiom can be seen in the interactor below with the comment *next state axiom*.

The final variation allows us to restrict the use of actions. This is done by using the ***per*** keyword to make sure the action can only be executed when the following condition is met, reinforcing the control over the development of the state. It can be seen in the example below marked with the comment *action permission axiom*.

The following code is an excerpt of a MAL interactor developed to represent the gate example from Section 2.2:

```
types
  States = {opening, closing, opened, closed, stoped}
interactor main
 attributes
  [vis] currentState: States
  isOpening: boolean
 actions
  Rc
  Fo
 axioms
  [] currentState = closed & isOpening # initial state axiom
  currentState = closed -> [Rc]
    (currentState'= opening & isOpening') # next state axiom
  ...
  per(Fo) -> currentState = opening # action permission axiom
  ...
```

- **Property Editor**

  This module is responsible for the formulation of properties of the model, previously defined using the MAL editor. These properties are expressed in CTL. In this plugin, some assistance is given, for those who are not so familiar with notation, using patterns for commonly checked properties. These patterns are used by instantiating them with attributes and actions from the MAL model, in order to generate the properties for verification [7].

- **Trace Visualizer**

  The trace visualizer is a plugin that allows for the visualization of counter-examples. If a property can not be proven true, then this module is responsible for helping the analyst visualize what went wrong, by providing a counter-example visualization.

  To visualize these counter-examples Ivy provides two main alternatives, a matrix notation showing the values of all attributes throughout the executions of the actions, and a variant of an Unified Modeling Language (UML) [46] activity diagram.

- **Trace simulator**

  The simulator allows for the exploration of different routes along the possible behaviours of the model, thus enabling the analysis of various scenarios.

Although Ivy is already a very useful tool, the MAL editor presents some flaws that are further detailed in section 3.1.

2.8 Discussion

When talking about UI modeling, there are several possible approaches. Like previously discussed, some of the existing tools use a tabular approach, that allows for a more visual experience, which can be perceived as better for not so experienced modelers. Others opt for a more textual interface. It can be said that this approach potentially has a steeper learning curve, but after such a learning curve it allows modelers to have much more expressive power available, and to specify the software to a point where most tabular interfaces cannot.

Even if these are the two most used interfaces styles for model editing found in the review, other styles can be used. Other tools use more graphical approaches, such as the state machines based approach seen in Figure 4.

Although the modeling approach may vary, the essence of the tools does not change. They are mostly based on an action-reaction style, where the developer describes one or multiple states, and the transitions between them through a set of actions, giving all tools a common ground.

The selection of which approach to use will be done in Section 3.4.

Chapter 3

Ivy Workbench new editor

3.1 Current state of Ivy Workbench editor

Ivy's code editor has some functionalities in it already, such as basic syntax highlighting; history, so we can undo and redo; and a very basic system that allows for code completion [6]. The editor also offers a tree vision of the model to help with the navigation within it. All these features are important in a text editor, but they are not enough, especially for newer users trying to write in a language they are not familiar with.

As said in [14], "In the case of business software products and services, user experience is one of the most important determining factors. User experience determines how gladly and voluntarily an employee uses an application. This in turn can lead to many different positive financial effects, like increased efficiency and profit for companies.", so we understand how important it is that new users find the tool easy to use and productive, in order to be efficient and ultimately lead to better projects.

We can see in the Figure 6 an example of how the current MAL editor looks, and it is possible to understand that it does not provide much information to the user, which is obviously not ideal for users without the proper experience in the tool. The editor does not look like the modern editors we use today, and some of the buttons are confusing, for example, the find and replace component fills most of the toolbar, even if the user does not want to use it.

Some of Ivy's current editor problems are:

- Code completion — we need to go to the tree view presented by the editor, open the tree (which closes when the model is edited), and click the attribute we want to write, in order to insert it in the code. This is clearly not good enough, since it can take longer to auto-complete the code than writing it.

- Lack of hints — new users without a background on MAL will not be able to be efficient since Ivy

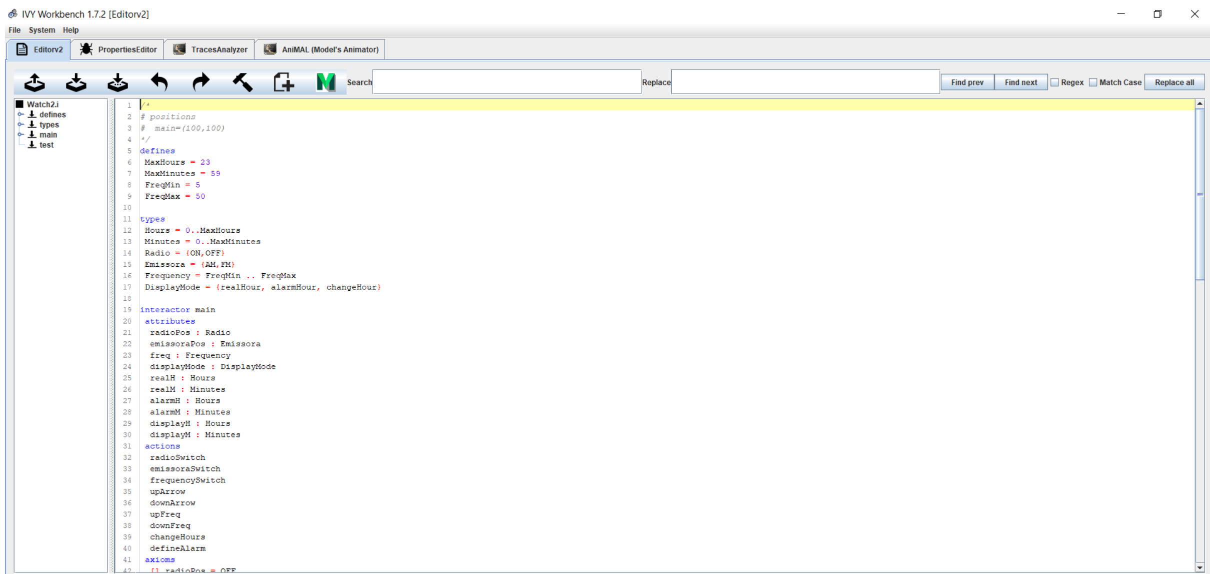


Figure 6: Ivy's Workbench current code editor

does not guide them through the process of building an interactor, nor does it explain what already written code means.

- The MAL editor is a built in plugin in Ivy, and trying to expand it can be challenging — most of the code used to develop Ivy is now legacy code, that went through several iterations. Thus, it can be more complex than alternatives, such as developing an extension.

- Ivy does not provide a simpler way for new users to introduce their own rules into the models directly from the code editor — this could be achieved through the identification of patterns, in order to allow those without experience to write their own properties by just filling in the missing information in a given pattern, instead of having the need to have a vast knowledge about the MAL language.

For all these reasons, it becomes clear that we must find a way to update this editor and make it more efficient to use (by non-experts), while also assuring the evolution of the software, enabling for better development in the future.

3.2 Requirements

It has become clear that there are two main alternatives as to how the software discussed earlier supports modeling, either a visual or a textual interface. As previously discussed in Section 2.8, both approaches have strong and weak points.

Before starting to implement the solution itself, it is important to determine the requirements, so that we have a guide and a set number of goals to achieve. These requirements can be split into three categories: interpreting, editing and fixing the models.

- **Interpreting the model**
  - **Syntax Highlight**

    Functionality that allows the users to distinguish between distinct parts of the model, such as actions, attributes or keywords.
  - **Semantic Highlighting**

    Functionality that changes the colors pre-established by the syntax highlight to better fit the context of the model. For example, after syntax highlight, all attributes present in the interactors will have the same color, regardless of if they are defined or not. Through semantic highlighting, we can change the color of an attribute depending on whether the attribute was already defined or not.
  - **Hover Information**

    Functionality that allows the users to find more about the specifics of MAL, such as what specific keywords mean, or even, what is the type of a certain attribute. This is important so the users can obtain the information about how to write the language inside the editor itself, and thus be more productive and have a better understanding of how the language is used.
  - **Go to the definition**

    Allow the users to go to the part of the model where some attribute or action is defined by hovering over it. It improves the navigation through the interactors present in the file, and provides a more intuitive way of finding specific parts of the model.
- **Editing the model**
  - **Ability to undo, redo, select, copy, paste or find and replace**

    To facilitate the usage of the tool, some common features should be implemented. These features should be implemented in a way that makes it familiar to the user to use any of them.
  - **Snippets**

    Provides a way for users to write models more economically, increasing productivity, as well as, removing the need of knowing every aspect of the language. For example, when creating an interactor, the user has to write the interactor name, type the attributes keyword where the attributes start, and the same for all the other parts that constitute an interactor. However, with snippets, it is possible to only write some specific pattern, and with that information, obtain a foundation for the rest of the expression.
  - **Code completion**

    Functionality that provides alternatives for completing the expression the user is writing in real-time, allowing for faster development as well as providing help remembering specific

possibilities. The code completion functionality also takes into consideration the part of the model the user is in, so that it can provide the best match possible.

- **Fixing the model**
  - **Diagnostics**

    Diagnostics are a vital tool to help users correct their mistakes, by showing relevant information over the error. That way, the user can more easily identify where the errors are occurring.
  - **Quick Fixes**

    This functionality is very tied to diagnostics because it provides solutions to a set number of common errors the users make. For example, not attributing the correct type to an attribute and then using it as if the type was correct. In this case, quick fixes will find the type that the attribute was supposed to have and propose to the user the possibility of changing it accordingly, with only two or three clicks.

3.3 Possible implementation approaches

In order to implement either a visual or a textual editor, a variety of technologies can be used. However, some technologies are better suited for a certain type of interface than another. In this section, three possible technologies are discussed as possibilities to implement the editors analyzed previously, these being:

- Build a web based editor, avoiding the need to install software.
- Create an extension in a modern IDE, and help the users while keeping their normal tools.
- Build an entirely new editor from scratch, and plug it into the Ivy Workbench directly.

3.3.1 Building a web based solution

3.3.1.1 Textual editor

In the first method, the possibilities are endless, there are a lot of pre-prepared APIs like **QUILL**, **Editor.js**, or **CKEditor** that help us in creating our own What you see is what you get (WYSIWYG) online editor. These editors can be accessed by everyone with an internet connection, without having to download and install new software, and for this reason are easy to use.

In all the cases above, the APIs can be very useful if you are creating an editor to write plain text or a formatted text document, because they present tools like making the text bold or italic, changing the font and the font size, inserting images or text from other sources, or even inserting code blocks using libraries like **highlight.js**.

Diving further into one of the existing WYSIWYG APIs, Quill, it is possible to see that it presents a vast range of utilities in order to manipulate text. In [39] the whole API is described in detail. Some of the most important of these utilities are features such as the ability to change the text entered by the user, changing properties such as the color or font weight for certain words in an automatic way. The color changing ability is especially important to build a modern code text editor, since it allows users to better understand the language and its reserved words and special notation.

However, this process can over-complicate the implementation phase of the project since it would require a lot of effort to implement features such as auto-complete or hints for the code itself. This is because most of the available editors and frameworks do not present those kinds of functionalities, at least in a direct way.

In summary, when trying to build a web based code editor the process can end up being very rewarding, since it is possible to change virtually every aspect of the editor, and build it the way we intend it to. But that also leads to a much more complex implementation.

3.3.1.2 Visual method

If we are trying to build a more visual editor, then the web based approach is possibly a good one. It allows for the construction of UIs that could react to user input. With all the modern JavaScript front-end frameworks like React, Vue.js or Angular, it is possible to build a personalized, almost no code, editor that low experienced users could use.

Even if we aim for a more tabular method, there are several already built components that could be used to achieve it, with a tool similar to AG Grid [40]. AG Grid is a JavaScript grid component that can be plugged into all mainstream JavaScript frameworks and allows for treatment of data similar to Microsoft's Excel, facilitating the construction of tables and grids.

However, like previously stated, these visual approaches can be limited in terms of expressive power. This limitation will mean that developers will face challenges to precisely describe their user interfaces, leading to potentially not so accurate analysis. Aside from that, web solutions present the difficulty of maintaining the code up to date, while also managing compatibility issues between the different components of the web application.

3.3.2 Extension method

Building an extension to an existing IDE can take several steps, the first of all of them being selecting the IDE we want to build an extension for. In this subsection, we will consider three main IDE, these being IntelliJ, Eclipse and VS Code, and consider the tools they provide for building new extensions that could support the Ivy Workbench's needs.

3.3.2.1 Textual editor

- **IntelliJ IDEA**

  IntelliJ IDEA[1] is an IDE that was originally meant to be used for writing and developing Java software.

  This IDE is owned by JetBrains and although IntelliJ is focused on Java it also supports a range of other languages like Groovy, Kotlin, JavaScript, TypeScript or SQL. On top of all that, it also allows for the creation of custom language support plugins, which fits the goal of this thesis.

  The steps needed to develop a plugin are described in [10], but the main features this IDE provides are the ability to construct a custom language support plugin that, in theory, can be as complex as we want it to be. It provides features such as: Completion Contributor that would allow for the code completion requirement; a Structure View Factory, which would be useful when building the tree view of the project and even inside individual files; the Annotator that helps the user by providing hints for their mistakes; and even other functionalities such as Code Formatter and Reference Contributor to better organize and make more visually appealing code.

  Even with all these tools that IntelliJ provides, it also has some problems. Most of the times, IntelliJ is seen as a Java code editor and most users only use it for that. This creates a problem for users that normally do not write Java, and so have no point in having IntelliJ in their systems. Those users would have to install IntelliJ specifically for writing in MAL. Although this is not a problem exclusive to IntelliJ, it is certainly a problem.

  Another factor to take in consideration is that, based on a survey from Stack Overflow [45], IntelliJ IDEA is just the fourth most commonly used IDE, which makes it less likely to be used by most developers.

- **Eclipse**

  Eclipse[2] is an open source IDE that provides an extensible development environment for writing code. It includes support for modeling, Java and C/C++, amongst other languages [13]. As in IntelliJ, Eclipse also presents a way for supporting custom languages, which means it is possible to create a plugin that supports MAL.

  In [16] information is provided about how to create such extension. However, the description is more vague and a lot less informative than the guide which is available from IntelliJ. Developing the extension in Eclipse could be a bigger implementation challenge, since there are fewer resources to learn from.

  Eclipse extension tools allow for the development of features such as setting rules based on specific keywords, allowing for syntax highlight, and a content assistant that can provide code completion.

[1] https://www.jetbrains.com/idea/
[2] https://www.eclipse.org/

This IDE presents a problem already present in IntelliJ, as seen in [45] Eclipse is only the ninth most used editor, with only nearly fifteen percent of the respondents of the survey choosing it as their primary development environment.

- **Visual Studio Code**

  VS Code[3] is a code editor that was specifically built with extension in mind, this meaning that almost every aspect of VS Code can be customized. It presents an Extension API so developers can create their extensions with ease, and even some core functionalities of VS Code are built as extensions using this Extension API [27].

  VS Code offers several features that will help to satisfy the previously settled requirements. It is possible to use the language extension API from VS Code [28], which provides ways to implement code completion, go to definition, hover effects and more. This can be achieved by either using the languages API directly, or, in this case, creating a language server that can support MAL.

  Through the VS Code name space API [30], all features and required tools to build and develop an extension are available for the developers, such as end-points to add diagnostics and their respective fixes directly in the editor, or add hover information to help the user understand specific parts of the code.

  It is also possible to describe a grammar for a custom language, using a *TextMate Grammar*, which is described using JavaScript Object Notation (JSON), the regular expressions that describe the language, and the respective tokens that should be associated with such expressions. This allows for simple syntax highlight.

  Another point in favor of VS Code is that, according to [45], it is the most commonly used IDE with over seventy percent of the respondents saying they use VS Code over any other IDE, which makes it a good candidate for holding our extension.

3.3.2.2 Visual method

While it is possible to create some form of visual extension for an IDE like VS Code, the whole idea of developing an extension for a code editor revolves around the idea of a more textual approach. However, if we were to create a more visual one, it would be possible to use something similar to, or even use directly, the VS Code extension *Draw.io* which allows for the construction of diagrams directly in the editor, and for the visualization of the Extensible Markup Language (XML) it produces [11]. Theoretically, it would be possible to create a parser for this XML that would translate the information to MAL and allow the development of the models visually.

This approach would have some problems, possibly the most relevant one being the lack of a visual language to express MAL interactors at present.

[3] https://code.visualstudio.com/

3.3.3 Building a new editor plugin

If we were to develop a plugin to integrate directly into the Ivy Workbench, it is important to know that Ivy 2 is built using the Java 9 modular architecture [26]. A mechanism was created to help support this modularity. The basic file structure of these plugins starts with two files, *module-info.java* and *pom.xml*. The *pom.xml* file defines the information about the plugin, such as name, build and dependencies. Some of dependencies that should be in said *pom.xml* are ivy-core, which is the main core of Ivy, and ivy-messaging, used for the messaging mechanism. As for the *module-info.java*, we would need a Java Module to be created. In that module, the messaging and core dependencies should be required, as well as the Java Desktop, which provides the swing related libraries for the plugin.

To make the new plugin work together with Ivy Workbench we would have to implement the required APIs to match Ivy's plugins and services requirements [9].

This includes implementing the Plugin Service API and the Messaging Service API. Some of the methods that require implementation, are ***getGUI*** and ***getMenuItems***, amongst others. For example, the ***getGUI*** method is the one that should return a JPanel to be used in the main GUI of Ivy Workbench, and ***getMenuItems*** is the method responsible for integrating new menu items into the existing Ivy menu. It does so by returning a list of JMenu objects.

After the module is complete, it needs to be added to the modules tab in the main Ivy *pom.xml* file, so that it can be included and thus compiled into the main program.

This way of building the new editor presents both advantages and issues. For advantages, we can think of features such as full control about how the editor should behave, full control over the text area, and the possibility to implement virtually anything that can be made using Java and Swing for the UI. The main issues are somewhat correlated to the advantages. While we have full control, that also means that nothing is implemented yet, and so, functionalities like undo, find and replace, project navigation, and other basic aspects of an editor, need to be implemented manually or through the use of third party libraries.

Another possible problem, for some users, might be that they can not use their most commonly used text editor to code MAL. However, the files would already be integrated into Ivy Workbench and thus easier to compile and run together with the other Ivy tools.

3.4 Decision

As seen in the subsection above, there are many possible alternatives to build the new Ivy MAL editor. When considering the complexity, the amount of possible users as well as the scalability of the solution, it is possible to verify that developing a VS Code extension is the right choice among the presented alternatives.

VS Code and Microsoft offer a vast documentation and support for the development of extensions for this IDE, which is always important when selecting the development environment for a project like the one proposed on this thesis. Another point in favor of VS Code is the fact that their extensions are written in

JavaScript or TypeScript, meaning they are almost unlimited in terms of the possible features they can provide.

Chapter 4

Development of the VS Code Extension

The VS Code website provides a full guide on how to start the development of an extension [31]. The process is supported by the VS Code extension generator. Based on the type of the extension that is intended, the development language, and other details such as extension name, git repository initialization, and web pack integration, the skeleton of a basic first extension is generated.

4.1 Overview of developing a Language Support Extension

The type of extension that would best suite this thesis is the language support extension. This type of extension can control the behavior of the editor, making it format parts of text based on names given to each of the parts, named tokens. To configure a language support extension, the developer must choose a language id (such as 'php' or 'javascript'). In the case of this thesis, the 'mal' id will be used. The language name must also be defined. This name is used to show the user which language is present in the file that is currently open, once again the name chosen was 'MAL'. With the language id and name decided, it remains to be decided the file extension to be used with the language. In this case, we want to process the files with '.i' as an extension. With this information a basic structure of the extension is built automatically and can be seen in the following subsections.

4.1.1 Package.json

This file is probably the most complex one when an extension is created. It contains the name of the extension, the display name, the description to be shown in the VS Code Marketplace and the version. For the more technical attributes that should be defined in this file, one has the **activationEvents**, which defines in what situation the extension should start. In the case of the MAL support extension, the

onLanguage tag is used with the MAL value, so that every time VS Code detects that the file opened is written in the MAL language then the extension can run.

Another technical attribute is the **main** attribute which defines the main JavaScript file that represents the extension. However, the main file can be written in TypeScript, due to the fact that the Typescript file is later converted into JavaScript by the VS Code bundle. That JavaScript file is then placed into the `dist` folder, and so, the extension main file should be `./dist/extension.js`. The TypeScript file written for this extension, can be seen in Annex I.

However, when the language support type of extension is chosen, the main file attribute is not present in the **package.json** file, created by the extension generator, and needs to be added manually, if needed. In this thesis, we use this JavaScript file in order to integrate more complex editor features.

The **contributes** attribute holds most of the more specific information about the extension, and is represented as a JSON object. That object can contain various attributes such as **languages**, where we define an array of languages that we want VS Code to identify by their file extension, giving them an id, aliases and the respective *language-configuration.json* file, described in Section 4.1.3. The **grammars** attribute, where we identify the languages by their id, and give them their respective grammar. In this case, the grammar defined in the TextMate Grammar file in the syntaxes folder, explained in Section 4.1.2. The **commands** array is also defined inside contributes, and should contain the id of the command to be executed per request of the user through the command pallete of VS Code as well as the title of such command. No commands were developed in this extension.

Once the contributes attribute is defined, it is possible to define more custom attributes, through the **configurationDefaults** attribute. For example, in this extension semantic highlighting must be set to true in this attribute. Semantic highlighting is explained further in Section 4.2.2.

4.1.2 Syntaxes folder

The syntaxes folder contains the JSON file, or files, responsible for describing the syntax of the language, with TextMate Grammar. This file itself is separated into parts, starting with a **patterns** entry, which should contain all token types that make up the language, such as the comments, the strings or, more specifically, MAL tokens such as the defines keyword, used to define constants and aliases in an interactor.

The **repository** section follows. This section will contain one entry per each of the patterns described in the main patterns entry at the start of the file. Then, each of this entries will contain the actual pattern describing the token. This pattern defines the regular expression that should be matched, and a name that can later be processed by the VS Code Theme in order to attribute the right color highlight to that part of the code.

An example for how to parse comments can be seen in Listing 4.1, and the complete grammar can be found in Annex II. In Listing 4.1, we can see (in the repository) that the comments token is defined by a JSON object containing the name and the regular expression that defines a comment. The name should follow the naming conventions defined in [24], so that VS Code can use them to attribute the correct

highlight to the different parts of the text. In the example, we can see that the name starts with comment, and this will tell VS Code that this section should be highlighted as a comment, the same way it would in any other language. The regular expression defines as a comment any text starting with a hash character.

Listing 4.1: Comment pattern matching example inside the grammar file.

```
1  {
2      "patterns": [
3              {
4                  "include" : "#comments"
5              }
6          ],
7      "repository":{
8              "comments":{
9                  "patterns":[
10                     {
11                         "name":"comment.line.number-sign.mal",
12                         "match": "#.*"
13                     }
14                 ]
15             }
16     }
17 }
```

4.1.3 Language Configuration

The *language-configuration.json* file is responsible for the definition of symbols in the language such as the ones that should be considered for comments (either block or line ones), as well as brackets, auto-closing pairs, and surrounding pairs. These options are useful for the VS Code to be able to provide some functionalities such as the comment selection keyboard shortcut or embrace selected text in some surrounding pair, for example, curly braces.

For example, to define that line comments in MAL start with the hash character, the following *language-configuration.json* entry would be used:

Listing 4.2: Comments entry in the language configuration file.

```
1 {
2     "comments": {
3         "lineComment": "#"
4     }
5 }
```

The *language-configuration.json* file created for the MAL language in this thesis, is in Annex III.

4.2 Technology and implementation

In this section, all the technology and procedures for each requirement will be explained, to provide information about how the extension works.

4.2.1 Syntax Highlight

As previously described in Section 4.1.2, semantic highlighting is achieved in VS Code using TextMate Grammar files. These files are customizable JSON files that allow for the creation of patterns that we want to match with a specific part of code in our language.

The patterns need to be written as regular expressions. However, it is possible to add begin and end regular expressions that define the range of text that should be compared to a given set of patterns. For example, a valid variable name that is written in the attributes section of an interactor should have a different color than one written in the actions section. However, the regular expression that identifies valid names is the same in both sections, and so, this mechanism of defining both the begin and end regular expressions that surround a part of the code, allows us to distinguish between both possibilities, even though the regular expression is the same. In this example, the regular expression used to match possible names for actions and attributes is:

```
[a-zA-Z]+[a-zA-Z0-9\\_]*
```

This regular expression will match all patterns that start with a letter and are followed by any arrangement of letters, numbers and the underscore character. So, to separate the possibility of it being an action or an attribute we separate them with sections. The section for attributes will begin with the regular expression `attributes` and end with:

```
(?=\\b((actions)|(axioms)|(test)|(interactor)|
(aggregates)|(importing))\\b)
```

This regular expression matches all possible other keywords that start a new section. For actions, the begin regular expression is `actions` and the end is:

```
(?=\\b((attributes)|(axioms)|(test)|(interactor)|
(aggregates)|(importing))\\b)
```

For the same reason as for the end of the attributes section.

Aside from the regular expression we want to match, we need to define the syntax name for the pattern found. The name that we give to the pattern will change the color in which the matched expression is represented in the editor. The different alternatives for pattern names, as well as more detail about the TextMate Grammar files, can be found in [24].

4.2.2 Semantic Highlight

The logic to make Semantic Highlighting function correctly is the most complex feature built in the extension. It requires processing every line of the document, because it will need to know information about the whole text in order to highlight the different parts accordingly to the role they play in the model.

The process of separating the entire text into tokens is not straightforward, since it requires the separation of the information into different data structures that can be used to provide more accurate information to the users about how their code is written. The most relevant of those data structures are:

- A sections map that contains all possible sections that are present in an interactor (these being the attributes, the types, the defines, the interactor declaration itself, the importing, the actions, the axioms, the tests, and aggregates) as keys, and a Boolean that is true for the section that the parser is currently analyzing and false for all the other ones.

- A map that contains as keys the name of all interactors present in the file, and as values, another map containing the name of each attribute defined inside a given interactor as a key. The value of the second map is an object containing information regarding if the attribute was used at least once after its declaration, the attribute type, a number representing the line in which the attribute was defined, and a Boolean that represents if the attribute is alone in the line or if there are more attributes declared with the same type in that line.

- Similar maps to the above for actions, the constants present in defines and the enumerations, ranges, and arrays present in the types section.

Other structures, such as an array that holds the information about which attributes were defined in a given action, were built to achieve specific functionalities, and will be described when those functionalities are explained further in this thesis.

For semantic highlighting to work, we need to register a document semantic token provider, through the `vscode.languages` API, using **registerDocumentSemanticTokenProvider**. This registration takes in three parameters: a configurations object, that should include the language which the token provider is associated with, an actual document semantic token provider instance, and a legend that will associate the tokens provided by the provider with actual semantic tokens that will be used in the editor.

Both the configuration object, as well as the legend are simple arguments. A configuration object is just an object containing a language field with ”mal”, so it activates once the MAL language is detected in a file. The legend takes the tokens' possible types and creates a new semantic tokens legend that can be used by the editor to highlight code, this transition between the tokens legend and the highlight color is done in the background by VS Code that matches the legend to the current theme of the editor and gives that token the respective color [29].

The document semantic tokens provider instance is the most complex argument. A tokens' provider is responsible for delivering to the VS Code editor a set of objects that represent different tokens that the editor uses to format the text. For example, if an attribute that has not been defined in the attributes section gets used in the axioms, this tokens' provider should send an object to VS Code saying the line where it occurs, the starting character, the length and the respective token type. The editor will then highlight the attribute to show the error. The possible token types can be seen in [29].

In this extension, we created a class **DocumentSemanticTokensProvider** which implements the corresponding VS Code version of the tokens' provider. We created the method **provideDocumentSemanticTokens**, which will separate the entire text written in the editor into tokens that can later be read by a Semantic Tokens Builder and passed to the editor. This token builder is responsible for transmitting the information of each token to the editor.

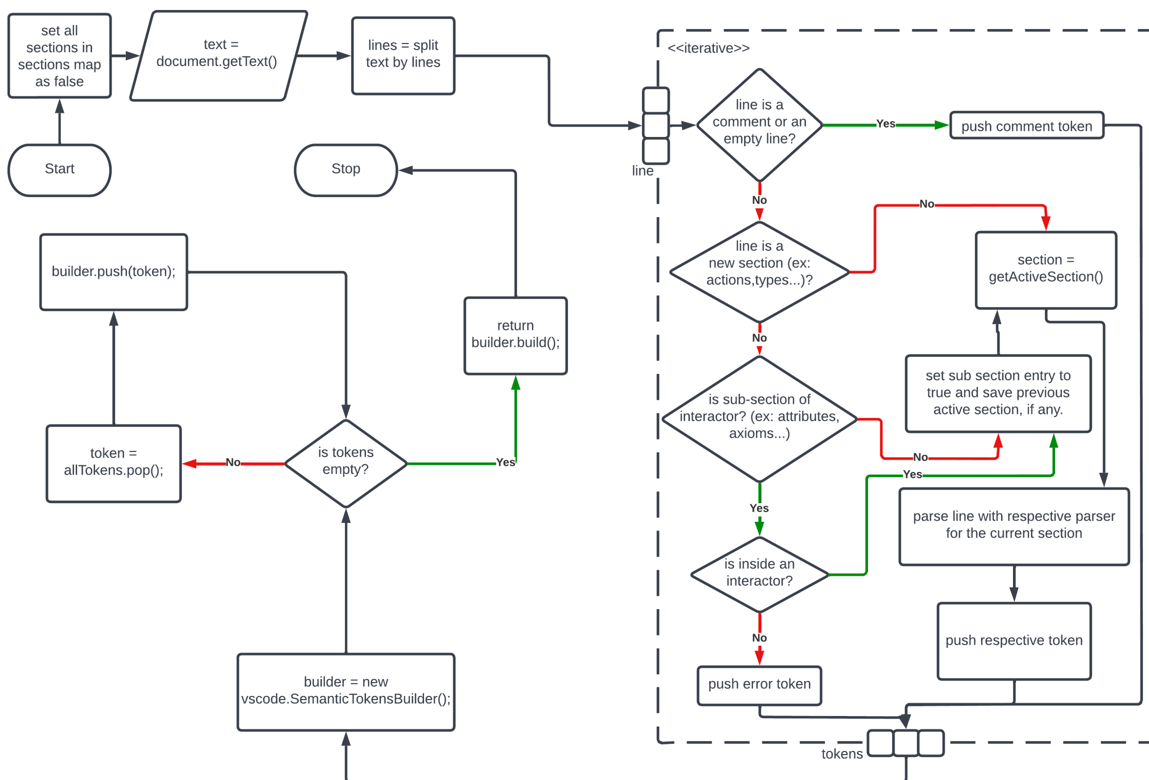


Figure 7: Process of building the tokens.

As seen in Figure 7, one of the first steps to filling the data structures and starting to find the tokens in the text is being able to identify in which part of the code we are currently in. And for that, we check if the line we are parsing at the moment corresponds to any keyword that represents the start of a new section inside the model. Once such keyword is reached, we set the Boolean that represents which section is active to true and forward the following lines to the corresponding parser, until we reach a new section. These smaller parsers were built to better separate the code, and facilitate the development of each parser more independently.

Since all parsers share logic and behaviour between them, we created a class that could be used by all parsers, this class is the **ParseSection** class. It contains three main class variables: the **findTokens**, the **separationSymbols**, and the **tokenTypeCondition**. **findTokens** represents a regular expression that should be compared to the line that is currently being parsed. If the pattern present in **findTokens** is in the line we can assume that the line contains the tokens we are looking for in that parser. After the line is compared with **findTokens**, and assuming that some pattern was found, that matching pattern is split by the **separationSymbols** regular expression also present in the class. This way, we just need to set these different variables to the specific section we want to parse.

For example, when parsing the declaration of the attributes inside an interactor, it is possible to define the attributes as:

```
att1,att2: boolean
```

In the case of attribute names, the **findTokens** regular expression is:

```
(\s*[A-Za-z]+\w*\s*(\,|(?=\:)))+
```

This regular expression will match every word that either has a comma or a semicolon following it. Which means that it would match with the two attributes, leaving us with `att1, att2` as a token. However, with the **separationSymbols** being described by a regular expression such as:

```
(\,|\s)
```

After matching the original string with both regular expressions, we will be left with `att1` and `att2` separately, as different tokens.

To provide all the information needed to create a semantic token, we also need to know the exact character where each token starts in the line. That is done by splitting the matched expression, using the **separationSymbols**, by registering the offset of each token, as well as its length.

The decision of what token type should be associated with the token is done through the last class variable, the **tokenTypeCondition**, which takes in both the value of the token as well as the offset, and returns a string. The string returned by the **tokenTypeCondition** method should match one of the possibilities of the token types described in [29], which are the currently supported token types by VS

Code. For example, if an attribute declaration is found that has already been declared, the method will return ”regexp”, otherwise it will return ”variable”.

This way, the user can notice more easily, through the change of color, that the attribute has already been defined, along with a diagnostic that will inform the user of the concrete error, and provide a quick fix with a possible solution for this kind of problem, these functionalities are explained in more detailed in Section 4.2.4.

All these tokens, along with the corresponding line number, start character, length, and token type are then returned by the **getTokens** method inside of the **ParseSection** class, so they can be used inside the semantic tokens builder.

Once the parse section class was completed, the writing of the smaller parsers was more intuitive. For example, when parsing the attributes present in an interactor, we use the **parseAttributes** method explained previously, along with two smaller parsers, the **parseVis**, and the **parseType**, all of which make use of the **ParseSection** class to form the **_parseAttributes** parser, which will use all three parsers to parse the whole section of attributes.

The **parseVis** method utilizes the regular expression:

```
^\s*\[\s*vis\s*\]
```

And the separation tokens:

```
(\[|\]|\s)
```

This regular expressions will allow us to get the vis keyword between square brackets in the start of each line. This keyword can be present in front of every attribute to define it as visible to the user. As for the **tokenTypeCondition**, we always return “keyword” because this method in specific is only looking for the presence of a word, and thus if a pattern is found we know that it is the “vis” keyword.

The **parseType** method is built similarly.

Returning to the example above:

```
att1,att2: boolean
```

The **parseVis** parser would not match anything, the **parseAttribute** would match both **att1** and **att2**, and the **parseType** would match boolean.

If no error is found and the type is valid (which it is in this case), we iterate over the auxiliary data structure **attributesInLine** and add each element to the attributes map with the correct type, along with the rest of the information needed, such as the used Boolean set to false, the line number, and the alone Boolean, which is calculated by determining if **attributesInLine** as a length bigger than one.

After all these methods run, and if a pattern is found, all the tokens found by each method are returned to the **parseText** method which forwards the tokens to the semantic token builder.

For all other sections, similar parsers are used. However, for more complex expression, such as model axioms, several levels of nesting of pattern matching are required.

4.2.3 Diagnostics

The diagnostics functionality is present in the extension through the use of another VS Code method, **createDiagnosticCollection**. This method creates a space where the diagnostics can be stored, meaning that if an error is found in the code it can be shown to the user in the form of a diagnostic.

To add a diagnostic to the collection, a method called **addDiagnostic** was created. It takes the line number and the initial character where the error starts, along with the string containing the error. This method also receives, as arguments, the information needed to create a readable message to the user about the error: the diagnostic message, which is shown on hovering the error or in the console inside VS Code; and the severity of the diagnostic, which can be "error", "warning", "info" or "hint". All of these options will result in different colors for the diagnostic, providing the user with information about the specific problem diagnosed in the model. Another variable can be passed to the **addDiagnostic** method, the code of the diagnostic, which is later used to provide quick fixes, more information is provided in Section 4.2.4.

The diagnostic is then created with a new **vscode.Diagnostic**, which represents a diagnostic that the editor can process, and add to the diagnostic collection explained earlier.

4.2.4 Quick Fixes

```
types
Color = {red,blue,green}

interactor main
attributes

shouldBeANumber:boolean

shouldBeAColor:boolean

isDefinedTwice:Color

isDefinedTwice:Color

actions

axioms
[thisActionDoesNotExist] shouldBeANumber = 3+4
    & shouldBeAColor = red & willbe=false
        & isDefinedTwice in {red,white,blue}
```

Figure 8: Example of errors fixable through quick fixes

To create the quick fixes functionality, we needed to create a **CodeActionProvider**, and so the class **IvyCodeActionProvider** was created, implementing the **CodeActionProvider** interface.

In **IvyCodeActionProvider**, we needed to implement the **provideCodeActions** method, which receives the document, the range of the code that will trigger the action (i.e. The position at which the token starts and its length) and the context of the extension. To add quick fixes to each diagnostic, we took advantage of the code field.

To allow for standardized communication between the code action provider and the diagnostics, we created a protocol to facilitate information exchange between the two functionalities.While the diagnostics are beings created, a code field is added to each one, corresponding to the kind of solution for the specific error that generated the diagnostic. The format is simple, we use a string separated by colons, where the first field always corresponds to the type of solution, and the following fields depend on the information needed to implement the solution.

A set of quick fixes were implemented, including: **CREATE_CHANGE_NUMBER**, **DECLARE_ACTION**, **CHANGE_TYPE**, **ALREADY_DEFINED**, **DEFINE_ATTRIBUTE**, and **ADD_TO_ENUM**:

```
types
Number= 0..17
Color = {red,blue,green}

interactor main
attributes

shouldBeANumber:Number

shouldBeAColor:boolean

isDefinedTwice:Color

isDefinedTwice:Color

actions

axioms
[thisActionDoesNotExist] shouldBeANumber = 3+4
    & shouldBeAColor = red & willbe=false
        & isDefinedTwice in {red,white,blue}
```

Figure 9: Example after the fix for the wrong type, when the type is a number.

- **CREATE_CHANGE_NUMBER** is a quick fix that allows the user to declare an attribute with a numeric type when no numeric types are available. For example, in the example present in Figure 8, the **shouldBeANumber** attribute is defined as Boolean when it should be numeric (see axiom), and there are no numeric types to which we could change the **shouldBeANumber** type. So, to fix the error, a quick fix is provided that generates a Number type with a range correspondent to the value of the expression we want to match the attribute with, plus and minus ten. To check the type

of the attribute, the extension checks the attributes data structure that contains the type of each attribute defined in the file, and to verify that there are no numeric types, it checks the ranges data structure. These data structures were populated during the build of the semantic tokens. If the attribute does not have a numeric type and no range is defined, then the extension will add a type Number to the types section of the model and assign that type to the attribute. So, after the quick fix the model would be the one present in Figure 9.

```
types
Number= 0..17
Color = {red,blue,green}

interactor main
attributes

shouldBeANumber:Number

shouldBeAColor:boolean

isDefinedTwice:Color

isDefinedTwice:Color

actions
thisActionDoesNotExist

axioms
[thisActionDoesNotExist] shouldBeANumber = 3+4
    & shouldBeAColor = red & willbe=false
        & isDefinedTwice in {red,white,blue}
```

Figure 10: Example after the fix for a non declared action.

- **DECLARE_ACTION** allows for the declaration of actions that are not yet defined inside an interactor. In Figure 8, we can see that the interactor does not contain any actions, and so the action named **thisActionDoesNotExist** is not defined. To fix it, since no actions exist, we need to insert a line where the actions section starts, with the action itself. To do so, we verify if the action is present in the actions data structure for the current interactor we are in. The fixed model would be the one present in Figure 10.

```
types
Number= 0..17
Color = {red,blue,green}

interactor main
attributes

shouldBeANumber:Number

shouldBeAColor:Color

isDefinedTwice:Color

isDefinedTwice:Color

actions
thisActionDoesNotExist

axioms
[thisActionDoesNotExist] shouldBeANumber = 3+4
    & shouldBeAColor = red & willbe=false
        & isDefinedTwice in {red,white,blue}
```

Figure 11: Example after the fix to change an incorrect type.

- **CHANGE_TYPE** quick fix is self-explanatory. It checks if the attribute and the value we are trying to match in a relation have the same type, and in case they are not, a diagnostic with the code **change type** is created. In the example, **shouldBeAColor** is a Boolean, however, we are comparing it to a color, and thus the fix would be to convert **shouldBeAColor** to a Color type. To do so, the extension checks the value of the left side of the relation, in this case, it verifies that the red keyword is a member of the Color type and thus this is the type which the attributes will be assigned to. The fixed model would be the one present in Figure 11.

- **ALREADY_DEFINED** is a quick fix that is used in all possible definitions inside a file: attributes, actions, defines, and types. It tells the user that there is already one definition with that name, and thus one of them must be removed. We know if the attribute was already defined or not by checking the attributes data structure and verifying if the attribute we are declaring is not yet present in it. It is, the code that is sent with error will contain the line in which the duplicate is present so it can be erased. After the quick fix is ran in the model above, the fixed version would be the one present in Figure 12.

- **DEFINE_ATTRIBUTE** is very similar to the define actions quick fix, however it takes into consideration the use of the undeclared attribute to define its type. In the example, **willbe** is not defined and it is being compared to a Boolean, thus the quick fix should insert an attribute with the name **willbe** and type Boolean in the interactor. To know that the attribute is not defined it checks the attributes data structure, and once the extension knows that the attribute is not present it will try to

```
types
Number= 0..17
Color = {red,blue,green}

interactor main
attributes

shouldBeANumber:Number

shouldBeAColor:Color

isDefinedTwice:Color

actions
thisActionDoesNotExist

axioms
[thisActionDoesNotExist] shouldBeANumber = 3+4
    & shouldBeAColor = red & willbe=false
        & isDefinedTwice in {red,white,blue}
```

Figure 12: Example after the fix to a duplicate declaration.

```
types
Number= 0..17
Color = {red,blue,green}

interactor main
attributes
willbe : boolean

shouldBeANumber:Number

shouldBeAColor:Color

isDefinedTwice:Color

actions
thisActionDoesNotExist

axioms
[thisActionDoesNotExist] shouldBeANumber = 3+4
    & shouldBeAColor = red & willbe=false
        & isDefinedTwice in {red,white,blue}
```

Figure 13: Example after the fix for a non declared attribute.

find the type of the value to which the attribute is being compared to. Since the value is the keyword false, the extension knows it should be a Boolean. The fixed model is present in Figure 13.

- **ADD_TO_ENUM** allows users to add a value to an existing enumeration. In Figure 8, we see that the enumeration Color contains three colors: red, green, and blue. However, in the axiom, we are checking if **white** is present in the Color enumeration. Thus, the extension detects that the value

```
types
Number= 0..17
Color = {white, red,blue,green}

interactor main
attributes
willbe : boolean

shouldBeANumber:Number

shouldBeAColor:Color

isDefinedTwice:Color

actions
thisActionDoesNotExist

axioms
[thisActionDoesNotExist] shouldBeANumber = 3+4
    & shouldBeAColor = red & willbe=false
        & isDefinedTwice in {red,white,blue}
```

Figure 14: Example after the fix for a member not present in an enumeration.

is not defined and provides a quick fix to the user that adds the value to the correct enumeration. In this case, it adds white to the Color enumeration, as seen in Figure 14.

4.2.5 Hover Information

Providing information to the users when hovering over some part of the code is important to help the user better understand the meaning of what is written. This functionality is provided by the registration of a **HoverProvider**, and so, we created a class **IvyHoverProvider** that implements the **HoverProvider** interface. This class needs to implement the **provideHover** method, which receives the opened document, the position of the hover, and a cancellation token. To know which word is being hovered, we use the **getWordRangeAtPosition** method provided by the document, this method gives us the word (delimited by non-alpha-numeric values) in the position of the hover.

Once we know the word, we can provide information about what is being hovered to the user. For example, when the user hovers the mouse on the keyword per, the hover provider will create a message that states that: "per is a keyword of MAL that allows the user to specify in what cases an action is valid. The expression 'per(ac)->a=b' means that the action ac can only happen when the attribute a equals b". This is relevant information that an inexperienced user might not know about this specific keyword in the language. An example can be seen in Figure 15.

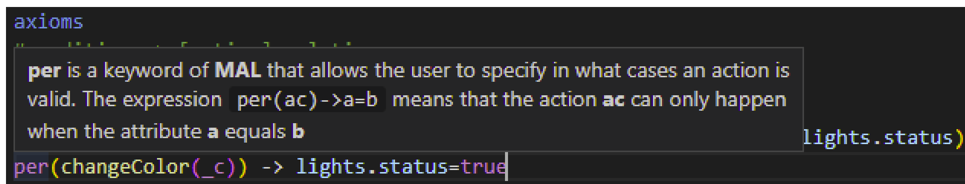


Figure 15: Example of the extension present when **per** is hovered.

4.2.6 Go to definition

The functionality is implemented by registering a **DefinitionProvider**. The approach was the same as for the hover provider. We created a class that implemented **DefinitionProvider** and a method **provideDefinition** that receives a document, the position, and the cancellation token. For detecting which word is the one that the user wants to know the definition of, we also used the **getWordRangeAtPosition**. After knowing the word, we iterate over the attributes and the actions to find a match for the word, and, in case there is a match, we return a location with the position where the variable was defined because that information is stored in each of the data structures that hold both attributes and actions. This feature enables the user to travel through the file without having to scroll it.

4.2.7 Snippets

To add snippets to the file we had to change the **package.json** present in the extension, by adding a snippets field to the contributes. This field allows us to tell the extension where the snippet file is located, as well as the language for which the snippet is for.

The snippets themselves are defined in the **snippets.json** file. The syntax for defining a snippet is simple, we create a JSON object containing all the different snippets we want to add to the extension. After that, each snippet is itself a JSON object (see Listing 4.3), where the key is the snippet name and the value is an object containing the prefix, the body, and the description. The prefix is the string that the extension will try to match to trigger the snippet, and so must be a memorable string so the users can use it without effort, instead of having to write the whole block of code. The body is where we define which lines are to be added to the file, with the possibility of determining spaces to be filled with custom information, as seen in Listing 4.3, where we define the interactor_name as an input field with the default value interactor_name.

Listing 4.3: Snippet for creating a new interactor

```
1 "Build Interactor": {
2     "prefix": "interactor",
3     "body": [
4       "interactor ${0:interactor_name}",
5       "attributes",
6       "",
```

```
7          "actions",
8          "",
9          "axioms",
10         "#condition -> [action] relations",
11         "#example:",
12         "#a=3 -> [action1] b'=3 & keep(a,c)"
13       ],
14       "description": "Boiler plate for interactor"
15     }
```

4.2.8 Code Completion

The code completion functionality is obtained by registering **completionItemProviders**, through the **vscode.languages.registerCompletionItemProvider** method. This method receives as arguments: the language for the code completion item; a method that returns an array of **CompletionItems** and the trigger strings that will trigger the provider. The language for each provider was always set to MAL since this is the language, we are focusing on. For the method, we implemented one that receives the document, and the position of the text the user is currently writing. This method varied with each provider, and there are five main providers, each of them implements the method for the code completion respectively.

The first provider is the one that is always active, since it does not have any trigger string, and the method returns all the possible attributes, actions, defines, and aggregates possible in an interactor.

The second provider is triggered with either an equal sign or an open bracket. When the trigger is an open bracket, the provider will present the user with all the possible actions available, since the most likely piece of code to be written inside brackets are actions. If the trigger is instead the equal sign, this provider will check if the left side of the equal sign is an attribute that has an enumeration type, and if so, it will provide the different elements of the enumeration as alternatives to the user.

The third provider is responsible for providing alternatives when the user is using aggregated interactors, meaning that we need to look for the information in various interactors. To do that, we use the dot as a trigger character, and once a dot is written, we verify if the word behind it is an aggregated value of the interactor, we are currently in. If this is the case, we provide the user with the information present inside the aggregated interactor, this is the attributes, actions, and even the aggregations present in the other interactor.

The fourth provider gives the user completions for the types, meaning that the triggers are the colon and the word *of*. Once one of these strings is typed by the user, this provider will loop over all the enumeration, ranges and array types declared to provide the user with their names, along with the native type Boolean.

The fifth provider is "keep non defined", which is a code completion command that adds a keep with all the attributes of the interactor that are not yet defined, thus preventing non-deterministic behavior. To

find which attributes are not defined, we first need to know the ones that are. To find the defined attributes, we separate the axiom where the keep snippet is called, into the different words. Once the line is split, we iterate over the different elements and find those that are attributes in the current interactor. After we have a set of attributes defined in the line, we just compare them to the attributes defined in the interactor, and write each of the non-defined ones after the *keep* keyword.

4.3 Additional Functionalities

Two additional functionalities were added to enhance the extension. The first one, **Axioms Analysis**, is a tool that will allow the modelers to check which attributes are defined in the next state by each of the actions. This is relevant because it will provide information about which attributes will behave non deterministically, according to the axiom, since no specific value was set, which can lead to unexpected behaviour. By having a solution to show this information quickly we can make sure that the modeler will know such situation exist. The modeller can then fix the situation, if needed, or leave it if that is the intention. This functionality is explained in more detail in Section 4.3.2.

The second functionality, **Properties Creator**, will provide an alternative to writing the properties of the interactors, in a more intuitive way and without the need of previous knowledge of temporal logic. This is explained in Section 4.3.3.

4.3.1 Web Views

Before diving further into how **Axioms Analysis** and **Properties Creator** functionalities were implemented, we need to understand what a web view is, and how it is implemented in a VS Code extension.

The idea to use a web view came from the need to have a way to display information to the user in a practical way, while also being able to receive input. For example, being able to display to the user that a certain variable was not defined in an axiom probably could have been done using a diagnostic message. However, if the user did not write any attribute in the axiom the diagnostic message could become large and too difficult to read inside a diagnostic message box.

We tried multiple alternatives, such as dialog boxes, or information text pop-ups provided by VS Code. However, none of these alternatives perfectly matched the idea for what we are trying to design, because the problem of providing the users with an intuitive UI was not possible due to the lack of customization these alternatives provided.

Due to that, we looked for alternatives that would allow us to fully customize a UI inside VS Code, and web views are special views that work similarly to a website. VS Code displays the web view as a different editor, allowing the web view to be used while using the normal text editor. However, web views present a problem, since they can execute arbitrary code, they are isolated from the extension, and thus a system to allow communication between them needs to be built.

A web view is defined using HTML and can load JavaScript and CSS, meaning it can be built as a normal web application [18]. To implement these web views, the first step was to establish a view container that would hold both web views (Axiom Analysis and Properties Creator) created in the extension. So, we needed to change the **package.json** file and declare a view container, giving it an id, a title, and an icon file, as seen in Listing 4.4.

Listing 4.4: View Containers

```
"viewsContainers": {
      "activitybar": [
        {
          "id": "mal-explorer",
          "title": "MAL Functionalities",
          "icon": "resources/info.svg"
        }
      ]
}
```

Once the view container is defined, we define each view inside the view container. Each view has a type that will be set to the web view, an id, and a name, like in Figure 4.5.

Listing 4.5: View inside the container

```
"views": {
      "mal-explorer": [
        {
          "type": "webview",
          "id": "mal-deterministic",
          "name": "Axioms analysis"
        },
        {
          "type": "webview",
          "id": "mal-properties",
          "name": "Properties Creator"
        }
      ]
}
```

Both the views containers and the views must be inside the contributes object of the **package.json**. To allow the views to have content, we then need to register a **WebviewViewProvider** for each. We do so by using the **vscode.window.registerWebviewViewProvider** method, that receives the view id, and

an instance of a **WebviewViewProvider**. We created separate files for each **WebviewViewProvider**, and on each file, we declared a class that implemented the **vscode.WebviewViewProvider** interface. This class contained a **viewType** variable that holds the name of the view the class is responsible for, as well as two main methods the **_getHtmlForWebview** and the **resolveWebviewView**. The **_getHtmlForWebview** method is responsible for returning the HTML that will be present in the webview, and so it returns a string containing that HTML. As for the **resolveWebviewView**, this method is where we set the information about the webview because it is the method that is called on creation. To use scripts inside the HTML, we need to define the **enableScripts** option of the webview as true, and that is done by using the webview object passed to **resolveWebviewView** as an argument and changing the options.

The communication between the web view and the extension, is done by using the **onDidReceiveMessage** method present in the web view object. This method allows us to decide what needs to be done in the extension once it gets a message from the web view. We do that by establishing a switch where the cases are the different data types a message could have in the environment of the extension.

For example, one of the reasons why a set of messages could be exchanged is when the web view asks the extension which interactors are defined in the file. For that, the web view sends a message to the extension with the type "get-interactors". Once the message reaches the extension, it enters the switch inside **onDidReceiveMessage** and will trigger another method **postMessage**, that allows the extension to send messages to the web view. The message to be sent to the web view will also contain a type and a possibilities element composed of an array containing all the interactors present in the data structure of the extension. The type is added inside the message sent so that the web view can do similar processing of each message based on the kind of message received.

As mentioned above, the web view implements a similar mechanism for processing the messages. However, to tell the web view to listen to the messages, we need to add an event listener that will be triggered with the event "message", and so, each time the web view receives a message the data will be extracted, and the data type will be matched inside a switch to the corresponding operation.

The next problem that occurred when developing the web views was that writing complex UI, such as a tree view or a navigation system, just became too verbose and complex with plain JavaScript. Thus, we opted to use a framework that could produce the HTML with less complexity, and facilitate the writing of the web views, thus allowing for more complex interfaces. The framework chosen was **React.js** since it is one of the lightest JavaScript frameworks, producing the smallest production build most of the time [41].

After the framework was chosen, we needed to find a way to integrate it inside the extension and to do so we focused on the process described in [15]. We started by installing react and WebPack using npm inside the extension. Using WebPack to compile the extension allowed us to compile all the React files into a single JavaScript file that could be used by the extension to add as a script HTML tag to the HTML of the web view. With all the required tools to develop a UI that could support all our needs and provide useful tools to the users, we started developing the web views.

4.3.2 Axioms analysis

This web view aims to give the users a visual and intuitive alternative to check that the model axioms fully define the effect of actions in the next state (i.e. they define the next value of all attributes in each action). This functionality is important so users can prevent non-deterministic behavior for their model, because if an attribute's value is not defined, the verification engine will consider that it can have any value, and this may lead to undesirable behaviors.

To achieve it, we provide the users with a tree view interface, which allows the user to navigate through the different interactors present in the model. Initially, the web view provides a button for users to refresh their actions and reload the web view information. Once that button is pressed, a message is sent to the extension, asking for the information of each action inside the interactors. The message type is set to **receiveActions**, this way the extension knows what the data the web view wants.

Once that message is received, another message is sent, now from the extension to the web view as a response. The message will be an object of type **refreshActions**, and, for each interactor, the total number of attributes for that interactor and the attributes that are not defined using that action.

Once the message is received in the web view, we iterate over the elements inside the message to render a tree view interface containing the information of each interactor. The interface, after receiving the message, can be seen in Figure 16.

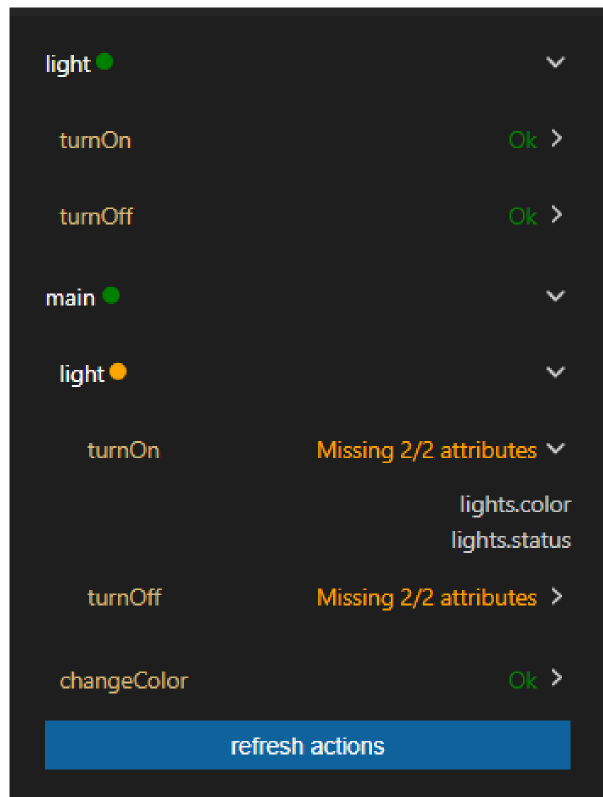


Figure 16: Web view interface for the action determinism functionality.

In Figure 16, we used a file containing only two interactors: **light** and **main**. In interactor **light**,

we can see two actions declared, **turnOn** and **turnOff**. And we can also see that both action have all attributes defined in the next state. In interactor **main**, which aggregates **light** via a variable named lights, we can see that there is a separation between **main**'s own actions and the actions that come from the **light** interactor. There, we can see that, inside the **main** interactor, the action **changeColor** defines all attributes, but the actions that come from the interactor **light** do not have any of the two attributes defined.

With this information the user can fix the axioms by defining the missing attributes, without the need to look for the undeclared and possible nondeterministic paths himself.

4.3.3 Properties Creator

This web view is meant to allow users to write their properties intuitively and without the need to know the extensive set of rules that compound a property. To achieve that, we started by studying the properties editor plugin inside IVY Workbench, which can be seen in Figure 17.

The idea is very similar. However, with the new tool being implemented inside of the editor, the users are able to develop their properties without having to change their development environment. This integration of the writing of properties inside of the editor enables the user to write the property while looking and navigating through the code, something that is not possible in the current functionality inside the IVY Workbench.

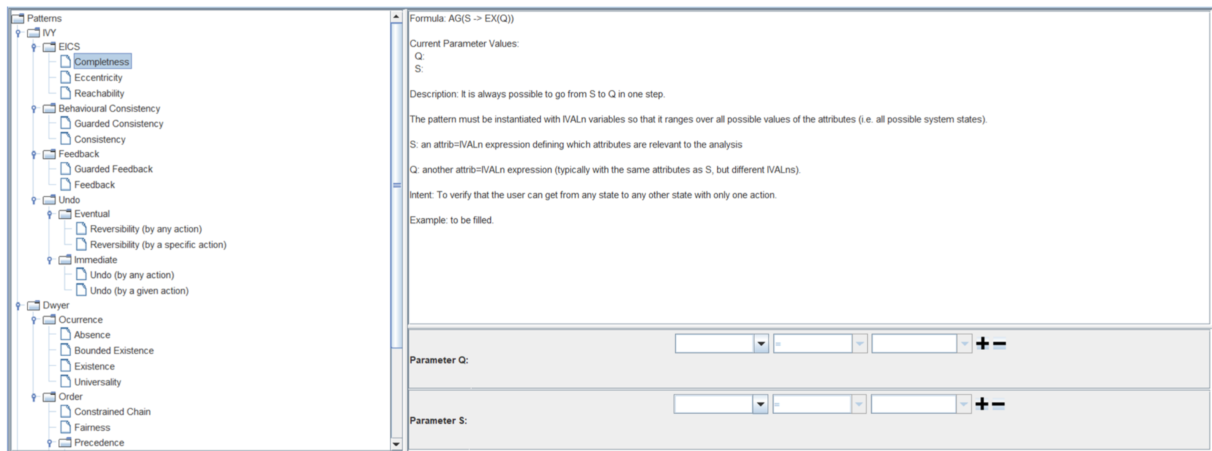


Figure 17: Ivy Workbench's property creator.

In Figures 17 and 18, we can see that both interfaces provide the possibility of creating patters. They allow the users to specify the arguments of the patterns (they are called parameters in Ivy) while also providing information about what each argument means. This information is important for inexperienced users that might not know how to write a specific property.

We made some quality-of-life changes to the interface, including closing and opening tabs so that users are not overwhelmed by a large amount of information, allowing for the selection of the content they want to see. Also, an argument can be instantiated with a conjunction of condition, however, in Ivy's

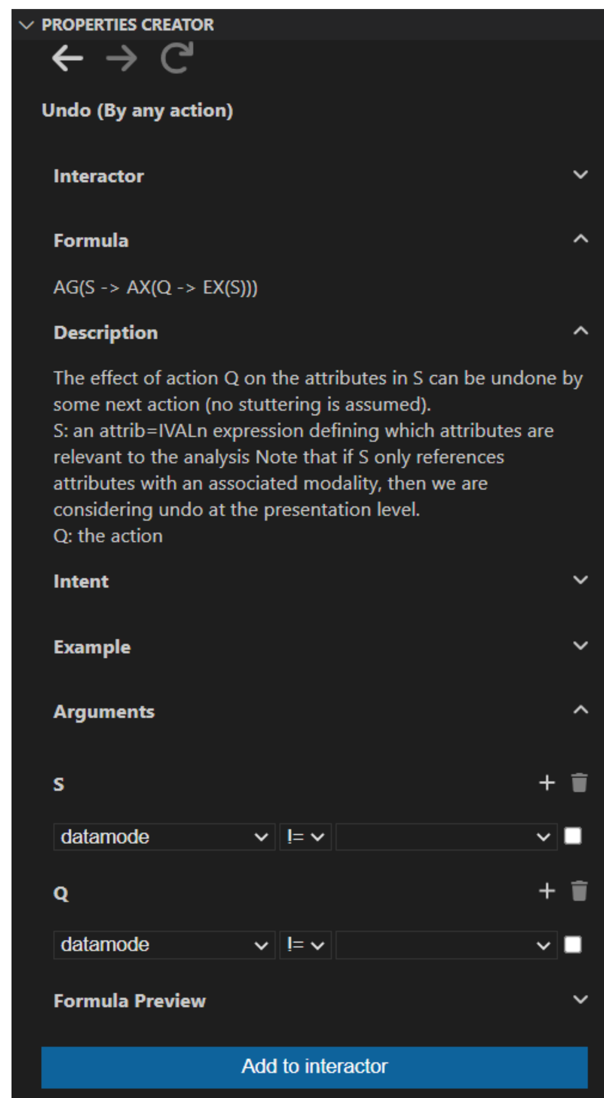


Figure 18: Extension's property creator.

interface, the users cannot remove a specific condition for a parameter. For example, if the user added ten conditions to the argument S, and then decided they wanted to remove the first one they wrote, they would have to delete all the conditions one by one. With the extension interface, the users can select exactly which condition they want to delete, and just press the bin icon after the selection is made, providing a better UX. Once the interface was developed, we needed to provide the correct information to the web view so the users could write properties based on the interactors they developed. To do that, we use the messaging system between the extension and the web view to provide the required information.

The information about the interactors is stored in a component called Pattern, using the **useState** hook inside React. The Pattern component is also responsible for providing the user with information about each specific pattern. For that, it takes as inputs the pattern information: the name, the formula, the description, the intent, the example, and the list of arguments that form the property's formula, as seen in Figure 18. Most of the information is only text that helps the user understand the pattern, and thus

a simple drop down interface with the information name and the information itself is enough. However, the arguments are inputs, which require some additional logic.

To render the arguments, we use the Argument component, which takes as inputs, the **onValueChange** method, the current active interactor name, the name of the argument, usually a letter, and the VS Code object so it can send and receive messages from the extension.

The **onValueChange** method is responsible for notifying the Pattern component that a change occurred inside an argument, so that the Pattern can store the value of each variable when a change happens. This storage of variable values is needed so that the web view can later produce a string representing the instantiated property.

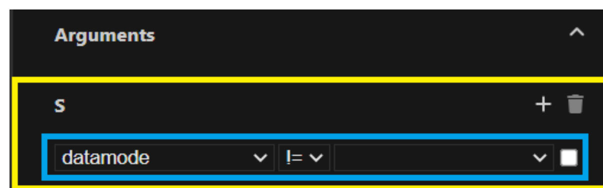


Figure 19: Choose interactor drop down.

The Argument component, which is represented in Figure 19 as a yellow rectangle, has an internal state that consists of all the identifiers of each possible input for a variable, all the identifiers of the selected inputs (which is important for the removal of such inputs), and the values the user chose for each input. This component also renders each argument's conditions using the **ArgumentInput** component.

ArgumentInput, represented in Figure 19 inside the blue rectangle, is where most of the complex communication happens. When this component is mounted, using the **useEffect** React hook, we send a message with the type interactor-info requesting the possible attributes that can be on the left of the condition inside **ArgumentInput**. Once the extension receives this message, it will build a response message, containing the attributes and actions that are present in the interactor the web view wants. These values are placed on the left side of the condition.

Aside from the interactor-info message, the **ArgumentInput** component also sends a message each time the left side value of the relation is altered, so that it can provide the user with correct set of possible values for the attribute chosen. This message will have the type get-valid-values. After the extension receives this message, it will check what is the type of the value received and check for variables with the same type inside the specified interactor.

For example, if the user defined an array of Boolean named **boolArray** with 3 elements, those 3 elements would be eligible values for the left side of the operator (**boolArray**[0],**boolArray**[1] and **boolArray**[2]), and when the user selects one of those values, the web view will send a message requesting information about **boolArray**[x]. The extension is then capable of detecting it as a member of an array of Booleans and will provide the user with the options TRUE, FALSE, $boolean or any other attributes that have the Boolean type so that the relation makes sense.

After the user has defined all the argument inputs for the property he can press the “Add to Interactor” button and the property will be immediately written in the file without the need for the user to know anything specific about the formal language used to develop the property.

4.4 Results

With the features explained in this Section, all the requirements set in Section 3.2 for the new editor are met. This way, we ensure that the functionalities we wanted to have in the new editor are present, while also creating software that uses more up-to-date technology, which can be subject to changes more easily in the future than the current Ivy editor.

Chapter 5

Usability testing

In order to assess the new editor against the current Ivy editor, we performed usability tests. In this tests, we asked users that were not proficient in MAL or the Ivy Workbench, to use both the extension and the editor to write a MAL interactors modal. This allowed us to get a comparative response from them, and with the collected data evaluate both solutions.

5.1 Research questions

The goal of the study was to find which editor gives the user the better UX and guidance throughout its usage, and also if the users utilized the new features and solutions provided in the new editor. For this we defined two main research questions.

- **RQ1:** Which of the two editors provides the best UX and guidance?

  By getting an answer to this question, we will be able to conclude which of the solutions was more satisfying for the users, and so, perform an evaluation and comparison of both tools.

- **RQ2:** How and when do the users utilize the features of the new editor?

  This question serves the purpose of knowing if the functionalities in the new editor are used or not. With this information we will be able to improve functionalities that are used less by facilitating their use to the user.

5.2 Procedure

To collect meaningful data from the users, we used a within-subjects test design [8] and we separated the test into two parts. The idea is that all users experiment both solutions separately, while also making sure

that half of the users start with one tool, and the other half starts with another, so that the results are not influenced by which tool the user uses first.

A questionnaire was applied at the end of the use of each tool, since questionnaires allow for an efficient quantitative measurement of a tool's features [25]. Another questionnaire was applied at the end of the session, to ask the participants about ways to improve the editor further.

The first questionnaire we used was the UEQ [42]. This questionnaire is a well know method to perform usability evaluation, and is suited to perform tests that compare two identical tools in order to know which one is better regarding a set of parameters. These parameters are attractiveness, perspicuity, efficiency, dependability, stimulation and novelty.

UEQ is composed of 26 questions. Each question is answered in a 7-point Likert scale, between two antonyms where the participant has to decide which point is closer to the balance between the adjectives that better suits the experience. Since these results are numerical, it becomes easier, through the use of a results analysis tool, to find results that give a concrete value on which editor is better in each of the parameters [44].

The second questionnaire consisted of two questions. The first question was "What did you think about each of the editors? Which did you prefer and why?", and the second question was "How could each of the editors be improved?". With these questions we aimed to get a clear point of view from the participants on which editor they enjoyed the most, as well as how they could be improved.

5.3 Participants

All participants needed to be over 18 years old, and have basic knowledge of programming, so that they could understand the requested tasks within the usability test, and thus provide better feedback over which of the tools they preferred.

5.4 Material

The test was performed remotely, through a video call with the participants. The required materials were:

- Computers for both the participant and the mediator, both with connection to the internet.
- A video conference tool, such as Zoom, so the participant could talk to the mediator as well as record the session. Zoom was also used to grant the participant the possibly of controlling the mediator's computer. This method avoided the process of installing required software in the participant's computer, thus reducing the complexity of performing a test for the participant.
- The mediator had Ivy Workbench and the VS Code extension installed, as well as Java to be able to run Ivy.

5.5 Description of the test

The test was designed to last 45 minutes, and it was separated into seven stages:

- **Welcome** – The participants were welcomed and asked to sign an informed consent to record the session as well as storing their answers. Duration: **5 minutes**.
- **Brief explanation of MAL** – The participants were explained the basics of MAL, so that they could understand the proceeding tasks. This explanation was done using the MAL example file present in Annex IV, which consists of an air conditioner that can be turned on or off, whose temperature can be increased or decreased by one, and where the user can enable or disable the temperature display. The participants were informed that they were not expected to understand every aspect of the language, and that was fine. Duration: **5 minutes**.
- **Use the first tool** - The participant were asked to write the example in Annex IV into the corresponding first tool, either the current Ivy's editor or the extension. However, this example contained an error. The error was the missing declaration of an attribute named **status**. After the participants finished typing in the example, they were asked to find the error present in the interactor and to try to fix it. Duration: **10 minutes**.
- **First Questionnaire** – The participants answered to the UEQ questionnaire regarding the experience with the first tool. Duration: **5 minutes**.
- **Use the second tool** – The participants were asked to write a similar example to the one present in Annex IV with the second tool. This example differed from the first one only in the error present. In this example, the error was missing a declaration of the action **toggleDisplay**. If the participants were capable of finding the error, they were asked to fix it. Duration: **10 minutes**.
- **Second Questionnaire** – The participants answered the UEQ questionnaire regarding the experience with the second tool. Duration: **5 minutes**.
- **Final Questionnaire** – The participants answered two questions. The first being which of the editor the participant would choose and the second one about more ways to improve both editors. Duration: **5 minutes**.

5.6 Data Collecting

In order to have access to more information about the tests, the mediator took notes during the course of the test. This notes included key moments during the test where the participants had difficulties or concluded a task in a non standard manner, as well as, registering if the participants were able to find the errors present in each of the files and how much time it took them.

The notes were taken manually by the mediator during the test and later copied to a spreadsheet in Google Sheets. The need for the notes to be taken manually comes from the fact that the participant had taking control over the mediator's computer, and thus, prevented the mediator from taking notes using it.

Along with the notes, a recorded video of the test was saved for posterior analysis, if necessary.

5.7 Demographic of participants

The test was performed by 12 participants, which was enough for UEQ to give reliable information. A minimum number of participants can not be set because it depends on the deviation of the answers. If the answers do not deviate between them, we can find reliable information with a lower amount of participants [42], which was the case.

All participants where between 19 and 23 years old, with 11 of them being male and 1 other participant without a specific gender. All the participants had some programming background so that they could understand the tasks proposed during the test.

5.8 Observations during the tests

While the participants were testing the two editors, a mediator was taking notes on the behaviour of the participants, in order to understand possible difficulties they had during this process.

During the testing of the Ivy editor, the participants had difficulties using the code completion mechanism, as well as not being able to easily identify typos due to lack of information from the editor. Also, the need to compile to see the errors in the model was mentioned several times as a flaw of the editor.

With the VS Code extension, some participants expressed that the code completion and the snippets mechanism were helpful and allowed for a more intuitive usage. However, there were also some problems. In particular, the participants sometimes did not use the code completion mechanism to its fullest.

Because they were copying a model, instead of writing one themselves, there were situations where the participants did not notice the suggestions made by the editor, since they were looking at the text they had to copy more than at the screen.

The quick fixes provided by the VS Code were viewed as a positive feature by the participants. However, some did not use them due to lack of visibility provided by the editor. The quick fix trigger is provided intrinsically by VS Code, and thus cannot be changed, so the usage of quick fixes seems to depend on the familiarity the users have with this functionality inside VS Code, which was the case for participants that used them in other programming languages.

5.9 Open questions' answers

In order to get a clear answer to which of the editors the participants preferred, they had to answer a question about which of the editors they were more likely to use if necessary and why. The result was clear, all the 12 participants chose VS Code as the editor they were more likely to use.

The reasons the participants gave for their choice were common amongst the answers:

- Good auto-complete functionality that allowed the participants to write their code more proficiently.
- The VS Code editor was able to find errors while writing the model, instead of having to compile or run the model like they had to for Ivy, thus reducing the feedback loop and improving the overall UX.
- The mistakes done by the users in the code were clear and visible, with the error message also being easily understandable.
- The possibility to check which attributes are set in the following state of each action, as explained in Section 4.3.2.

In the other open question, the participants were asked to answer about ways for both editors to improve. All participants mentioned that one way to improve Ivy would be to add auto completion like the one present in VS Code, since the code completion provided by Ivy was too difficult to use and made the participants frustrated.

Other answers also mentioned ways we could use to improve the VS Code extension, with features such as a preview of the whole model along side the code and also improving the visibility of the quick fix button on the dialogues.

5.10 User experience questionnaire results

The UEQ results were analysed by using the UEQ data analysis tool to compare two versions of a similar product [42]. The results provided are presented as a Likert scale [35], where the value ranges from -3 (very negative experience) to +3 (very positive experience). We can observe in Figure 20, that the VS Code editor had better results in all points evaluated in the UEQ.

In Figure 21, we can see more details regarding the answers given in the UEQ. We can see that the standard deviation of the answers regarding Ivy was higher than the one in VS Code. This difference can be explained by the fact that the answers of the participants fluctuated based on which editor they experienced first.

As we can see in Figures 22 and 23, the participants that started the tests with Ivy gave better reviews to Ivy than the ones that started with the VS Code extension. These results can be explained by the comparison made with the previous editor used. As one uses the VS Code editor first, then tends to

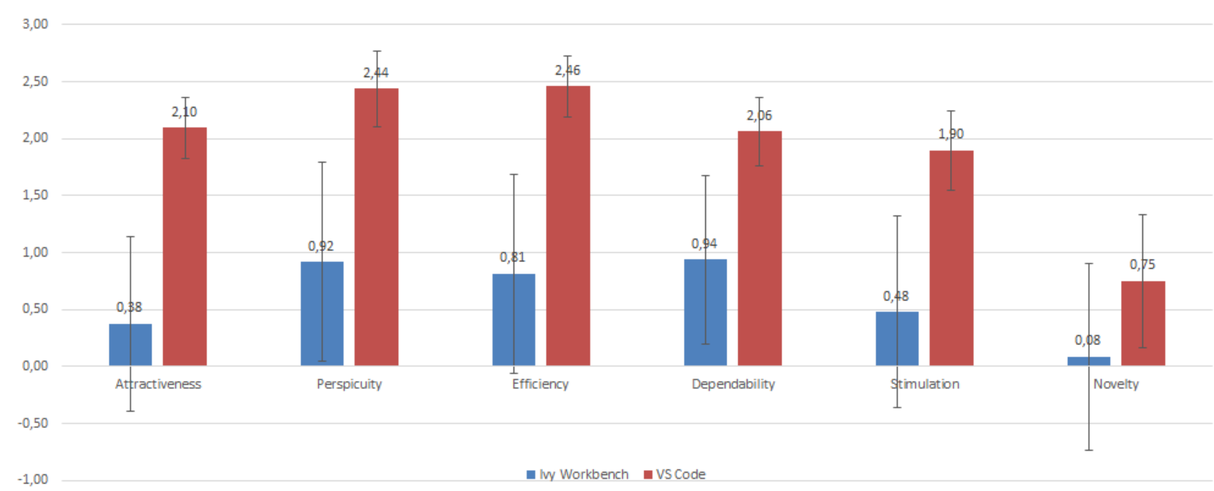


Figure 20: Results obtained via UEQ data analysis tool, where the blue represents the answers given to the Ivy editor and the red represents the answers given to the VS Code editor.

Scale	Ivy Workbench						VS Code					
	Mean	STD	N	Confidence	Confidence Interval		Mean	STD	N	Confidence	Confidence Interval	
Attractiveness	0,38	1,36	12	0,77	-0,39	1,14	2,10	0,47	12	0,27	1,83	2,37
Perspicuity	0,92	1,55	12	0,87	0,04	1,79	2,44	0,59	12	0,34	2,10	2,77
Efficiency	0,81	1,55	12	0,88	-0,06	1,69	2,46	0,47	12	0,27	2,19	2,73
Dependability	0,94	1,30	12	0,74	0,20	1,67	2,06	0,53	12	0,30	1,76	2,36
Stimulation	0,48	1,48	12	0,84	-0,36	1,32	1,90	0,61	12	0,34	1,55	2,24
Novelty	0,08	1,46	12	0,82	-0,74	0,91	0,75	1,03	12	0,58	0,17	1,33

Figure 21: Statistics from the UEQ analysis tool, where STD means standard deviation and N is the number of tests.

give lower scores to the editor used in the second place (Ivy Workbench). When the participants started using the Ivy Workbench editor, they gave higher responses to this editor, and then had a lower margin to increase the scores for the VS Code extension.

5.11 Result analysis

Based on the results, we can assume that the participants of the test enjoyed using VS Code more, as they gave better feedback to it in both the UEQ questionnaire and the open questions.

In Figure 26, which shows if the difference between results from the UEQ of each tool is significant or not, it is clear that the results obtained by the UEQ show a considerable difference in all fields, except novelty, between VS Code and the current Ivy editor. This leads us to conclude that the goal of updating the editor of Ivy to current standards, while also improving the UX, was overall successful.

5.12 Answers to research questions

In Section 5.1, we defined two research questions that were meant to be answered by these tests.

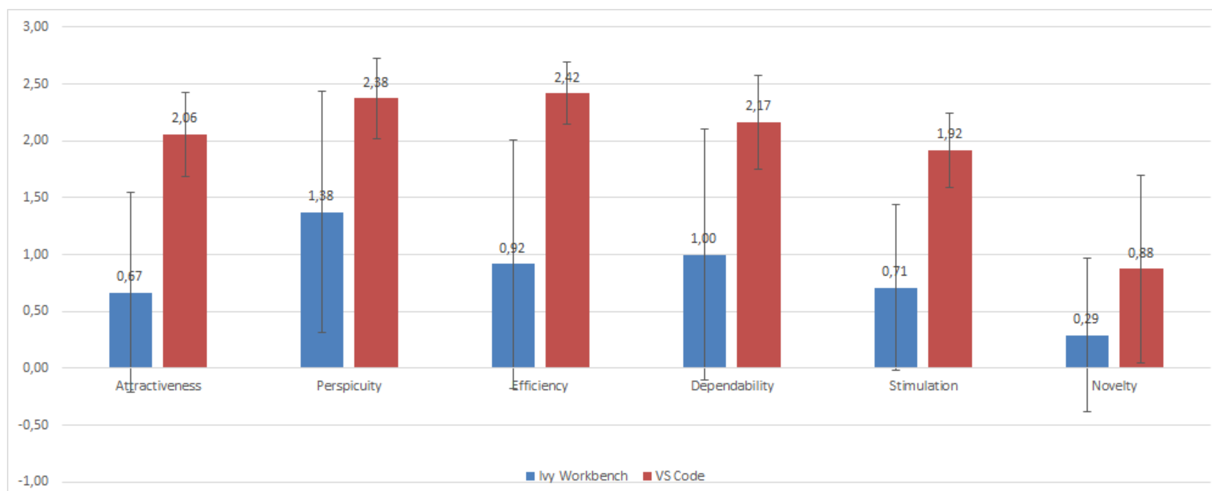


Figure 22: Results obtained via UEQ data analysis tool, where the participants started with Ivy Workbench.

Scale	Ivy Workbench Mean	STD	N	Confidence	Confidence Interval		VS Code Mean	STD	N	Confidence	Confidence Interval	
Attractiveness	0,67	1,10	6	0,88	-0,21	1,54	2,06	0,47	6	0,37	1,68	2,43
Perspicuity	1,38	1,33	6	1,06	0,31	2,44	2,38	0,44	6	0,35	2,02	2,73
Efficiency	0,92	1,37	6	1,09	-0,18	2,01	2,42	0,34	6	0,27	2,14	2,69
Dependability	1,00	1,38	6	1,10	-0,10	2,10	2,17	0,52	6	0,41	1,75	2,58
Stimulation	0,71	0,91	6	0,73	-0,02	1,44	1,92	0,41	6	0,33	1,59	2,24
Novelty	0,29	0,84	6	0,67	-0,38	0,97	0,88	1,03	6	0,83	0,05	1,70

Figure 23: Statistics from the UEQ analysis tool, where the participants started with Ivy Workbench.

- **RQ1:** Which of the two editors provides the best UX and guidance?

  Through the results of the UEQ, which were analyzed in Section 5.11, we were able to establish a comparison between the two editors. This lead us to conclude that the VS Code extension provided a better experience for the participants. The answers to the open questions mentioned in Section 5.9, also showed a preference towards the VS Code extension.

- **RQ2:** How and when do the users utilize the features of the new editor?

  During the test, the moderators were able to tell, which functionalities provided in the new editor were used by the participants. This observation is already described in Section 5.8. Most users used the functionalities of the new editor, even though some did not use the quick fixes functionality. The possibility to see the errors made while writing the model was used by the participants regularly, and some even fixed the errors present in the model before they were done writing it. However, since the test used a simple model, some of the features such as the **Axiom analysis** or the **Properties Creator** were not used.

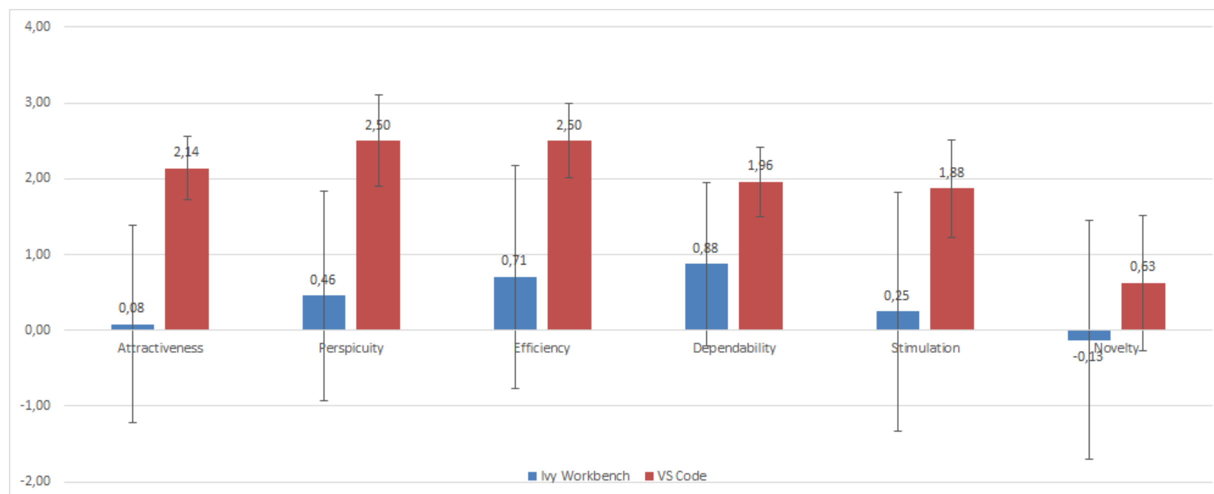


Figure 24: Results obtained via UEQ data analysis tool, where the participants started with the VS Code editor.

Scale	Ivy Workbench						VS Code					
	Mean	STD	N	Confidence	Confidence Interval		Mean	STD	N	Confidence	Confidence Interval	
Attractiveness	0,08	1,63	6	1,30	-1,22	1,39	2,14	0,52	6	0,42	1,72	2,56
Perspicuity	0,46	1,73	6	1,38	-0,92	1,84	2,50	0,76	6	0,61	1,89	3,11
Efficiency	0,71	1,84	6	1,47	-0,76	2,18	2,50	0,61	6	0,49	2,01	2,99
Dependability	0,88	1,35	6	1,08	-0,20	1,95	1,96	0,58	6	0,46	1,49	2,42
Stimulation	0,25	1,97	6	1,58	-1,33	1,83	1,88	0,80	6	0,64	1,23	2,52
Novelty	-0,13	1,96	6	1,57	-1,69	1,44	0,63	1,12	6	0,89	-0,27	1,52

Figure 25: Statistics from the UEQ analysis tool, where the participants started with the VS Code editor.

Alpha level:	0,05	
Attractiveness	0,0010	Significant Difference
Perspicuity	0,0066	Significant Difference
Efficiency	0,0038	Significant Difference
Dependability	0,0146	Significant Difference
Stimulation	0,0081	Significant Difference
Novelty	0,2106	No Significant Difference

Figure 26: Difference in UEQ results for both tools.

5.13 Threats to validity

While results are positive, there are some aspects of the study that might have influenced the results:

- The number of participants is low. More results would allow a higher confidence in the results; even so, there is a strong agreement between the test subjects.
- The fact that the tools were used remotely might have impacted negatively the UX, both in terms of the UI reaction time, and some problems with keyboard configurations, that made it harder to

write specific characters. To minimize this factor, the conditions were kept the same for both tools.

- The sample is not gender balanced, due to the small number of participants it was not possible to solve this.
- There seemed to be some level of 'Social Desirability' bias, with participant not wanting to score the Ivy legacy editor too low. We believe this has not affected the results as in all conditions, the legacy editor was, in any case, ranked lower.

Chapter 6

Conclusion

The Ivy Workbench is a tool that can be used to model UI, using MAL, which allows for formal verification of properties of that interface. This formal verification is useful especially in critical software where an unforeseen situation can lead to a complex problem.

However, the tool's architecture and features have gone through several changes over the years, and thus, it is hard to maintain and upgrade in a way that would allow Ivy to live up to current software standards. This is the reason why a better solution was needed to provide users with a comfortable model editor, while also making sure the software would be easier to maintain and update in the future. Since we wanted to update the software behind Ivy's model editor, we also sought to build a solution that would allow future developers to facilitate model development by users with less formal methods and programming technical background.

In order to build a better fitting solution, the first step was to find the problems with the current editor of Ivy. With the knowledge of the problem we could think of solutions and alternatives that would fix such problems and improve both the software and the UX. After finding an alternative manner to implement Ivy's new code editor, there was the need to understand this alternative better and implement the desired features, to do so we chose to implement an extension for the well know VS Code editor.

Finally, a user study was carried out, to validate the proposal.

6.1 Results

Objectives were set at the beginning of this thesis to guide the development of the solution. At this point, we can use such defined goals and understand what we were able to achieve during this thesis. The objectives were the following:

- **Identify Ivy Workbench current editor's problems and present possible alternatives to solve them.**

  The main problem with Ivy Workbench was that it was difficult to maintain and was in need of an update.

  Even though Ivy provided some features such as, basic highlighting or a find and replace mechanism, these were not enough by current IDE standards.

  Ivy's editor did not communicate enough information to the user, which lead to problems, especially for people that are not used to such interfaces, since Ivy's editor did not match the modern style of current code editors. It was lacking features such as code completion, lack of hints for possible errors, and it did not provide an easy way for users to write MAL properties, especially for those without a background in formal verification methods.

- **Research about different alternatives for building the new editor and present a solution**

  We explored many different solutions, from visual editor to textual ones, across multiple platforms, such as a web based solution, building an extension or writing a new plugin for Ivy.

  Eventually, we considered the idea of building an extension for VS Code the most suited solution, because it was the one that allowed us to reach more users while also providing an extensive amount of documentation for implementing all the desired features.

- **Develop and implement the new solution for the code editor**

  After having made the decision to build a VS Code extension we could implement a solution for the problems identified in the current editor.

  The code completion problem was solved by adding providers that would give the user the best match according to the current code they were writing. This is further explained in Section 4.2.8.

  The lack of hints present in Ivy was also solved, by displaying messages, or making parts of the code stand out to the user, to facilitate the process of finding mistakes and also solving them. This was done with multiple features in this solution, namely quick fixes and semantic highlight. These are described in Sections 4.2.4 and 4.2.2, respectively.

  Writing formal properties is also supported in this solution, with the introduction of the Properties Creator functionality, explained in Section 4.3.3.

- **Evaluate the develop solution in comparison to the old one**

  Finally, we performed an user study to evaluate the two solutions. The participants were asked to rate both solutions, so we could do a comparative analysis.

The results were enlightening. The VS Code extension got better results both in the UEQ, as well as in the open questions, where participants often said that the features missing on Ivy were present in the VS Code extension. The whole test process is presented in Chapter 5.

6.2 Future work

Even though the objectives were accomplished, there are still functionalities that could be added to improve the extension further.

A functionality that could be added is an option to format the code automatically, providing a clear structure that could be more easily read and understood, while also removing the responsibility of formatting the code from the user, improving the UX.

Adding a way for directly compiling the code in VS Code would also be a meaningful addition to the extension.

Another useful functionality would be to allow the aggregation of interactors written in different files, this change would also require changes in the compiler.

Aside from adding new features, expanding the properties creator to have more patterns and enhance the interaction with the user further would also be a point to take into consideration.

Bibliography

[1] S. Ahamed. "Studying the feasibility and importance of software testing: An Analysis". In: *arXiv preprint arXiv:1001.4193* (2010) (cit. on p. 1).

[2] D. Billman et al. "Complementary tools and techniques for supporting fitness-for-purpose of interactive critical systems". In: *Human-Centered and Error-Resilient Systems Development*. Springer, 2016, pp. 181–202 (cit. on p. 10).

[3] G. Brat et al. "Formal Analysis of Multiple Coordinated HMI Systems". In: *The Handbook of Formal Methods in Human-Computer Interaction*. Springer, 2017, pp. 405–431 (cit. on p. 8).

[4] J. C. Campos and M. D. Harrison. "From HCI to Software Engineering and back". In: International Federation for Information Processing (IFIP). 2003 (cit. on pp. 4, 6).

[5] J. C. Campos, M. D. Harrison, and K. Loer. "Verifying user interface behaviour with model checking". In: (2004) (cit. on p. 6).

[6] J. C. Campos et al. "Formal verification of a space system's user interface with the IVY workbench". In: *IEEE Transactions on Human-Machine Systems* 46.2 (2015), pp. 303–316 (cit. on p. 16).

[7] J. C. Campos et al. "Supporting the analysis of safety critical user interfaces: An exploration of three formal tools". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 27.5 (2020), pp. 1–48 (cit. on p. 15).

[8] G. Charness, U. Gneezy, and M. A. Kuhn. "Experimental methods: Between-subject and within-subject design". In: *Journal of economic behavior & organization* 81.1 (2012), pp. 1–8 (cit. on p. 49).

[9] R. Couto. *IVY 2 development guide V1.0*. Dec. 2021 (cit. on p. 23).

[10] *Custom Language Support Tutorial | IntelliJ Platform Plugin SDK*. url: https://plugins.jetbrains.com/docs/intellij/custom-language-support-tutorial.html (visited on 12/26/2021) (cit. on p. 21).

[11] *Draw.io integration - visual studio marketplace*. url: https://marketplace.visualstudio.com/items?itemName=hediet.vscode-drawio (cit. on p. 22).

[12] D. J. Duke and M. D. Harrison. "Abstract interaction objects". In: *Computer Graphics Forum*. Vol. 12. 3. Wiley Online Library. 1993, pp. 25–36 (cit. on p. 13).

[13] I. Eclipse. “Eclipse ide”. In: *Website www. eclipse. org Last visited: July* (2009) (cit. on p. 21).

[14] F. Erd�s. “Economical Aspects of UX Design and Development”. In: *2019 10th IEEE International Conference on Cognitive Infocommunications (CogInfoCom)*. IEEE. 2019, pp. 211–214 (cit. on p. 16).

[15] N. Fabre. *Reactception : Extending vs code extension with webviews and react*. Oct. 2019. url: https://medium.com/younited-tech-blog/reactception-extending-vs-code-extension-with-webviews-and-react-12be2a5898fd (cit. on p. 43).

[16] *FAQ How do I write an editor for my own language?* url: https://wiki.eclipse.org/FAQ_How_do_I_write_an_editor_for_my_own_language? (cit. on p. 21).

[17] M. S. Feary. “A toolset for supporting iterative human–automation interaction in design”. In: *NASA Ames Research Center, Tech. Rep. 20100012861* (2010) (cit. on p. 8).

[18] C. Fricke. “Standalone Web Diagrams and Lightweight Plugins for Web-IDEs such as Visual Studio Code and Theia”. In: (2021) (cit. on p. 42).

[19] D. Garlan. “Formal modeling and analysis of software architecture: Components, connectors, and events”. In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer. 2003, pp. 1–24 (cit. on p. 1).

[20] M. D. Harrison, P. Masci, and J. C. Campos. “Balancing the formal and the informal in user-centred design”. In: *Interacting with Computers* 33.1 (2021), pp. 55–72 (cit. on pp. 5, 11).

[21] M. D. Harrison et al. “Formal techniques in the safety analysis of software components of a new dialysis machine”. In: *Science of Computer Programming* 175 (2019), pp. 17–34 (cit. on p. 7).

[22] *Human-centred design processes for interactive systems*. Standard. Geneva, CH, 1999 (cit. on pp. 4, 5).

[23] A. Konios. “User Guide of IVY Workbench”. In: (2010) (cit. on pp. 1, 12).

[24] *Language grammars*. url: https://macromates.com/manual/en/language_grammars (cit. on pp. 26, 29).

[25] B. Laugwitz, T. Held, and M. Schrepp. “Construction and evaluation of a user experience questionnaire”. In: *Symposium of the Austrian HCI and usability engineering group*. Springer. 2008, pp. 63–76 (cit. on p. 50).

[26] S. Mak and P. Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. ”O’Reilly Media, Inc.”, 2017 (cit. on p. 23).

[27] Microsoft. *Extension API*. Nov. 2021. url: https://code.visualstudio.com/api (cit. on p. 22).

[28] Microsoft. *Language extensions overview*. Nov. 2021. url: https://code.visualstudio.com/api/language-extensions/overview (cit. on p. 22).

[29] Microsoft. *Semantic Highlight Guide*. Nov. 2021. url: https://code.visualstudio.com/api/language-extensions/semantic-highlight-guide (cit. on pp. 30, 31).

[30] Microsoft. *VS code API*. Nov. 2021. url: https://code.visualstudio.com/api/references/vscode-api (cit. on p. 22).

[31] Microsoft. *Your first extension*. Nov. 2021. url: https://code.visualstudio.com/api/get-started/your-first-extension (cit. on p. 25).

[32] A. F. Monk and M. B. Curry. "Discount dialogue modelling with Action Simulator". In: *Proceedings of the conference on People and computers IX*. 1994, pp. 327–338 (cit. on p. 11).

[33] Y. Moy et al. "Testing or formal verification: Do-178c alternatives and industrial experience". In: *IEEE software* 30.3 (2013), pp. 50–57 (cit. on p. 7).

[34] M. Müllerburg et al. "Systematic testing and formal verification to validate reactive programs". In: *Software Quality Journal* 4 (Dec. 1995), pp. 287–307. doi: 10.1007/BF00402649 (cit. on p. 7).

[35] T. Nemoto and D. Beglar. "Likert-scale questionnaires". In: *JALT 2013 conference proceedings*. 2014, pp. 1–8 (cit. on p. 53).

[36] P. Oladimeji et al. "PVSio-web: a tool for rapid prototyping device user interfaces in PVS". In: *Electronic Communications of the EASST* 69 (2014) (cit. on p. 10).

[37] D. R. Olsen Jr. "Propositional production systems for dialog description". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1990, pp. 57–64 (cit. on p. 11).

[38] J. L. Peterson. "Petri nets". In: *ACM Computing Surveys (CSUR)* 9.3 (1977), pp. 223–252 (cit. on p. 10).

[39] *Quickstart*. url: https://quilljs.com/docs/quickstart/ (cit. on p. 20).

[40] *React Data Grid: Documentation*. url: https://www.ag-grid.com/react-data-grid/ (cit. on p. 20).

[41] E. Saks. "JavaScript Frameworks: Angular vs React vs Vue." In: (2019) (cit. on p. 43).

[42] M. Schrepp. "User experience questionnaire handbook". In: *All you need to know to apply the UEQ successfully in your project* (2015) (cit. on pp. 50, 52, 53).

[43] L. Sherry. *Personal communication regarding the use of the Operational Procedure methodology in the design of Honeywell product development*. 1996 (cit. on p. 8).

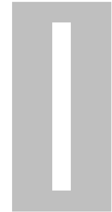
[44] I. Sintorn and M. Knöös Franzén. *Usability, User Experience and Aesthetics:-A Case Study at Toyota Material Handling*. 2022 (cit. on p. 50).

[45] *Stack Overflow Developer Survey 2021*. url: https://insights.stackoverflow.com/survey/2021#demographics-ethnicity-prof (visited on 12/26/2021) (cit. on pp. 21, 22).

[46] O. UML and I. MOF. *The unified modeling language UML*. 2011 (cit. on p. 15).

[47] M. P. Ward. “Reverse engineering from assembler to formal specifications via program transformations”. In: *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE. 2000, pp. 11–20 (cit. on p. 2).

Annex

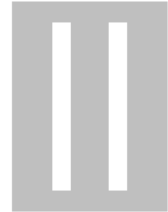


extension.ts

```
import * as vscode from "vscode";
import { addDiagnostic } from "./diagnostics/diagnostics";
import { clearStoredValues } from "./parsers/globalParserInfo";
import { _parseText } from "./parsers/textParser";

const tokenTypes = new Map<string, number>();
const tokenModifiers = new Map<string, number>();

const legend = (function () {
  const tokenTypesLegend = [
    "comment",
    "string",
    "keyword",
    "number",
    "regexp",
    "operator",
    "namespace",
    "type",
    "struct",
    "class",
    "interface",
    "enum",
    "typeParameter",
    "function",
    "method",
    "decorator",
    "macro",
    "variable",
```

```
29      "parameter",
30      "property",
31      "label",
32    ];
33    tokenTypesLegend.forEach((tokenType, index) =>
34      tokenTypes.set(tokenType, index)
35    );
36
37    const tokenModifiersLegend: any[] | undefined = [];
38    tokenModifiersLegend.forEach((tokenModifier, index) =>
39      tokenModifiers.set(tokenModifier, index)
40    );
41
42    return new vscode.SemanticTokensLegend(
43      tokenTypesLegend,
44      tokenModifiersLegend
45    );
46  })();
47
48  export function activate(context: vscode.ExtensionContext) {
49    vscode.window.onDidChangeActiveTextEditor(() => {
50      clearStoredValues();
51    });
52    context.subscriptions.push(
53      vscode.languages.registerDocumentSemanticTokensProvider(
54        { language: "mal" },
55        new DocumentSemanticTokensProvider(),
56        legend
57      )
58    );
59  }
60
61  class DocumentSemanticTokensProvider
62    implements vscode.DocumentSemanticTokensProvider
63  {
64    async provideDocumentSemanticTokens(
65      document: vscode.TextDocument,
66      token: vscode.CancellationToken
67    ): Promise<vscode.SemanticTokens> {
68      const allTokens = _parseText(document.getText());
69      const builder = new vscode.SemanticTokensBuilder();
70      allTokens.forEach((token) => {
71        builder.push(
72          token.line,
73          token.startCharacter,
```

```
        token.length,
        this._encodeTokenType(token.tokenType),
        this._encodeTokenModifiers(token.tokenModifiers)
      );
    });
    return builder.build();
  }

  private _encodeTokenType(tokenType: string): number {
    if (tokenTypes.has(tokenType)) {
      return tokenTypes.get(tokenType)!;
    } else if (tokenType === "notInLegend") {
      return tokenTypes.size + 2;
    }
    return 0;
  }

  private _encodeTokenModifiers(strTokenModifiers: string[]): number {
    let result = 0;
    for (let i = 0; i < strTokenModifiers.length; i++) {
      const tokenModifier = strTokenModifiers[i];
      if (tokenModifiers.has(tokenModifier)) {
        result = result | (1 << tokenModifiers.get(tokenModifier)!);
      } else if (tokenModifier === "notInLegend") {
        result = result | (1 << (tokenModifiers.size + 2));
      }
    }
    return result;
  }
}
```

Annex 

mal.tmLanguage.json

```
1  {
2    "$schema": "https://raw.githubusercontent.com/martinring/
     tmlanguage/master/tmlanguage.json",
3    "name": "MAL",
4
5    "patterns": [
6      {
7        "include": "#types"
8      },
9      {
10       "include": "#comments"
11     },
12     {
13       "include": "#keywords"
14     },
15     {
16       "include": "#strings"
17     },
18     {
19       "include": "#tags"
20     },
21     {
22       "include": "#defines"
```

```
23      },
24      {
25        "include": "#operators"
26      },
27      {
28        "include": "#components"
29      },
30      {
31        "include": "#numbers"
32      }
33    ],
34    "repository": {
35      "comments": {
36        "patterns": [
37          {
38            "name": "comment.line.number-sign.mal",
39            "match": "#.*"
40          }
41        ]
42      },
43      "numbers": {
44        "match": "[0-9]+",
45        "name": "constant.numeric.mal"
46      },
47      "operators": {
48        "patterns": [
49          {
50            "match": "((!\\=)|(\\=)|(!)|(&)|(->)|(<->)|(>)|(<)
     |(\\|)|(\\.\\.)|({)|(})|\\]|\\[)",
51            "captures": {
52              "2": { "name": "keyword.operator.not_equals.mal" },
53              "3": { "name": "keyword.operator.equals.mal" },
54              "4": { "name": "keyword.operator.false.mal" },
55              "5": { "name": "keyword.operator.and.mal" },
56              "6": { "name": "keyword.operator.implies.mal" },
57              "7": { "name": "keyword.operator.equivalent.mal" },
58              "8": { "name": "keyword.operator.greater.mal" },
```

```
59              "9": { "name": "keyword.operator.less.mal" },
60              "10": { "name": "keyword.operator.or.mal" },
61              "11": { "name": "keyword.operator.range.mal" },
62              "12": { "name": "keyword.operator.open_braces.mal" },
63              "13": { "name": "keyword.operator.close_braces.mal" },
64              "14": { "name": "keyword.operator.close_bracket.mal" },
65              "15": { "name": "keyword.operator.open_bracket.mal" }
66            }
67          }
68        ]
69      },
70      "tags": {
71        "begin": "\\[",
72        "beginCaptures": { "0": { "name": "keyword.operator.open_bracket.mal" } },
73        "end": "\\]",
74        "endCaptures": { "0": { "name": "keyword.operator.close_bracket.mal" } },
75        "patterns": [
76          {
77            "name": "constant.language.vis.mal",
78            "match": "\\s*vis\\s*"
79          }
80        ]
81      },
82      "defines": {
83        "begin": "defines",
84        "beginCaptures": { "0": { "name": "storage.defines.mal" } },
85        "end": "(?=\\b((types)|(interactor))\\b)",
86        "patterns": [
87          { "include": "$self" },
88          {
```

```
89            "name": "keyword.control.mal",
90            "match": "^\\s*[A-Za-z]+[A-Za-z\\_0-9]*(?=\\s*\\=)"
91          },
92          {
93            "name": "variable.language.next_state.mal",
94            "match": "(?<!^\\s*)[A-Za-z]+[A-Za-z\\_0-9]*'"
95          },
96          {
97            "name": "variable.parameter.mal",
98            "match": "(?<!^\\s*)[A-Za-z]+[A-Za-z\\_0-9]*"
99          }
100       ]
101     },
102     "types": {
103       "begin": "types",
104       "beginCaptures": { "0": { "name": "storage.types.mal" } }
   ,
105       "end": "(?=\\b((defines)|(interactor))\\b)",
106       "patterns": [
107         { "include": "$self" },
108         {
109           "name": "entity.name.type.custom.mal",
110           "match": "^\\s*[A-Za-z]+[A-Za-z\\_0-9]*\\s*(?=\\=)"
111         },
112         {
113           "name": "variable.parameter.mal",
114           "match": "(?<!^\\s*)[A-Za-z]+[A-Za-z\\_0-9]*"
115         }
116       ]
117     },
118     "components": {
119       "begin": "interactor",
120       "beginCaptures": { "0": { "name": "storage.interactor.mal
   " } },
121       "end": "(?=interactor)",
122       "patterns": [
123         { "include": "$self" },
```

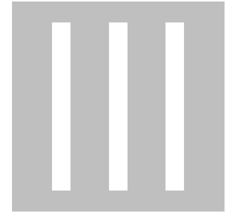
```
124         {
125           "begin": "attributes",
126           "beginCaptures": {
127             "0": { "name": "entity.name.tag.attributes.mal" }
128           },
129           "end": "(?=\\b((actions)|(axioms)|(test)|(interactor)
    |(aggregates)|(importing))\\b)",
130           "patterns": [
131             { "include": "$self" },
132             {
133               "name": "variable.parameter.mal",
134               "match": "(?<=(\\s*\\[\\s*vis\\s*\\]\\s*)?)(?<=([
    a-zA-Z0-9\\_]+\\s*,\\s*|(?<=(\\]|^)\\s+)))[a-zA-Z]+[a-zA-Z0-
    9\\_]*(?=(\\s+|,|:))"
135             },
136             {
137               "name": "entity.name.type",
138               "match": "(?<=:\\s*)[a-zA-Z]+[a-zA-Z0-9\\_]
    *(?=(\\s+|$))"
139             }
140           ]
141         },
142         {
143           "begin": "importing",
144           "beginCaptures": {
145             "0": { "name": "entity.name.tag.importing.mal" }
146           },
147           "end": "(?=\\b((actions)|(axioms)|(test)|(interactor)
    |(aggregates)|(attributes))\\b)",
148           "patterns": [
149             { "include": "$self" },
150             {
151               "name": "variable.parameter.mal",
152               "match": "(?<=(\\s*\\[\\s*vis\\s*\\]\\s*)?)(?<=([
    a-zA-Z0-9\\_]+\\s*,\\s*|(?<=(\\]|^)\\s+)))[a-zA-Z]+[a-zA-Z0-
    9\\_]*(?=(\\s+|,|:))"
153             },
```

```
154            {
155              "name": "entity.name.type",
156              "match": "(?<=:\\s*)[a-zA-Z]+[a-zA-Z0-9\\_]
   *(?=(\\s+|$))"
157            }
158          ]
159        },
160        {
161          "begin": "aggregates",
162          "beginCaptures": { "0": { "name": "entity.name.tag.
   aggregates.mal" } },
163          "end": "(?=\\b((attributes)|(axioms)|(test)|(
   interactor)|(actions)|(importing))\\b)",
164          "patterns": [
165            { "include": "$self" },
166            {
167              "name": "keyword.control.via.mal",
168              "match": "via"
169            }
170          ]
171        },
172        {
173          "begin": "actions",
174          "beginCaptures": { "0": { "name": "entity.name.tag.
   actions.mal" } },
175          "end": "(?=\\b((attributes)|(axioms)|(test)|(
   interactor)|(aggregates)|(importing))\\b)",
176          "patterns": [
177            { "include": "$self" },
178            {
179              "name": "entity.name.function.mal",
180              "match": "(?<=^\\s*(\\s*\\[\\s*vis\\s*\\]\\s*)?)[
   a-zA-Z]+[a-zA-Z0-9\\_]*(?=(\\s+|$))"
181            }
182          ]
183        },
184        {
```

```
185           "begin": "axioms",
186           "beginCaptures": { "0": { "name": "entity.name.tag.
    axioms.mal" } },
187           "end": "(?=\\b((attributes)|(actions)|(test)|(
    interactor)|(aggregates)|(importing))\\b)",
188           "patterns": [
189             {
190               "begin": "\\[",
191               "beginCaptures": {
192                 "0": { "name": "keyword.operator.open_bracket.
    mal" }
193               },
194               "end": "\\]",
195               "endCaptures": {
196                 "0": { "name": "keyword.operator.close_bracket.
    mal" }
197               },
198               "patterns": [
199                 {
200                   "name": "entity.name.variable.mal",
201                   "match": "[A-Za-z]+[A-Za-z\\_0-9]*"
202                 }
203               ]
204             },
205             {
206               "match": "\\bkeep\\b",
207               "name": "keyword.control.keep.mal"
208             },
209             {
210               "match": "\\bper\\b",
211               "name": "keyword.control.per.mal"
212             },
213             {
214               "match": "\\beffect\\b",
215               "name": "keyword.control.effect.mal"
216             },
217             {
```

```
218                 "name": "variable.language.next_state.mal",
219                 "match": "[A-Za-z]+[A-Za-z\\_0-9]*'"
220               },
221               {
222                 "name": "variable.language.mal",
223                 "match": "[A-Za-z]+[A-Za-z\\_0-9]*"
224               },
225               { "include": "$self" }
226             ]
227           },
228           {
229             "begin": "test",
230             "beginCaptures": { "0": { "name": "entity.name.tag.
    test.mal" } },
231             "end": "(?=((test)|(interactor)))",
232             "patterns": [
233               {
234                 "name": "entity.name.function.mal",
235                 "match": "(A|E)(G|F|X|U)"
236               },
237               { "include": "$self" }
238             ]
239           },
240           {
241             "name": "entity.name.tag.actions.mal",
242             "match": "actions"
243           },
244           {
245             "name": "entity.name.tag.axioms.mal",
246             "match": "axioms"
247           }
248         ]
249       },
250
251       "strings": {
252         "name": "string.quoted.double.mal",
253         "begin": "\"",
```

```
254       "end": "\"",
255       "patterns": [
256         {
257           "name": "constant.character.escape.mal",
258           "match": "\\\\."
259         }
260       ]
261     }
262   },
263   "scopeName": "source.i"
264 }
```



language-configuration.json

```
{
    "comments": {
        // symbol used for single line comment. Remove this entry if your language does not support line comments
        "lineComment": "#"
    },
    // symbols used as brackets
    "brackets": [
        ["{", "}"],
        ["[", "]"],
        ["(", ")"]
    ],
    // symbols that are auto closed when typing
    "autoClosingPairs": [
        ["{", "}"],
        ["[", "]"],
        ["(", ")"],
        ["\"", "\""],
        ["'", "'"]
    ],
    // symbols that can be used to surround a selection
    "surroundingPairs": [
        ["{", "}"],
```

```
23            ["[", "]"],
24            ["(", ")"],
25            ["\"", "\""],
26            ["'", "'"]
27        ]
28  }
```

Annex IV

Air Conditioner MAL example

```
defines
  MaxTemp = 50
  MinTemp = 10

types
  Temperature=MinTemp..MaxTemp
  Status = {on, off}

interactor main
attributes
  displayOn : boolean
  [vis] currTemp:Temperature
  [vis] status: Status
actions
  [vis] statusOffBtn
  [vis] UpTemp
  [vis] DownTemp
  [vis] toggleDisplay
axioms
  [] currTemp=MinTemp & status=off & !displayOn

  status=off -> [statusOffBtn] status'=on & displayOn' & keep(currTemp)
  status=on -> [statusOffBtn] status'=off & !displayOn' & keep(currTemp)

  per(UpTemp) -> status=on & currTemp < MaxTemp
  [UpTemp] (currTemp' = currTemp + 1) & keep(status, displayOn)

  per(DownTemp) -> status=on & currTemp > MinTemp
```

```
29 [DownTemp] (currTemp' = currTemp - 1) & keep(status, displayOn)
30
31 per(toggleDisplay) -> status=on
32 [toggleDisplay] displayOn'=!displayOn & keep(currTemp, status)
```