



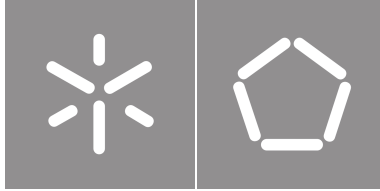
Universidade do Minho

Escola de Engenharia

Miguel André da Silva Solino

**Formal models based
interactive prototypes**

January, 2022



Universidade do Minho

Escola de Engenharia

Miguel André da Silva Solino

**Formal models based
interactive prototypes**

Master Dissertation

Integrated Master's in Informatics Engineering

Work developed under the supervision of:

José Creissac Campos

January, 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

I would like to thank first of all my family for the support and affection they always showed me so that I could finish my dissertation successfully. To my father I would like to say that he has always been an example for me, that the strength he has given me until today has always been a huge incentive for me to successfully finish my course and without the effort he made for me this would not be possible. My mother, besides also being an example for me, has always been with me at all times, and, besides her unimaginable effort, has always passed on to me a contagious and motivating energy so that working would be much easier. To my sister, I would like to say that this thesis is also dedicated to her, and that her support was crucial for me to have even more will to invest in my future.

Furthermore, I would like to say thank you very much to Professor José Creissac Campos for accepting to be my supervisor and for the support, availability and constant help that he always provided me so that I could finish my dissertation. I would also like to say thank you to Rafael Costa for the support he gave me during the preparation of the dissertation, together with Professor José Creissac Campos.

Finally, I would like to leave a mention to all my friends and work colleagues who were part of this journey, saying that with them the motivation I had to finish this journey was much greater. I will never forget all those who helped me to get this far.

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.

Braga
(Place), _____
31/10/2022
(Date)

Miguel André Silva Solino
(Miguel André da Silva Solino)

Abstract

Formal models based interactive prototypes

Since the UI is one of the first contacts that a user has with an interactive system being developed, it can include a user flow to be intuitive and effective. For this, prototypes are typically built in the early stages of the software development process so that both developers and users might have an early idea of what the final product will look like, smoothing development and testing in advance.

On critical interactive systems, the standard UI prototyping method is not enough. These systems require more advanced and detailed analysis methodologies to avoid failures that can lead to accidents. For this, there are approaches based on models and formal analysis methods, complemented by the prototyping process, which enhance the process of looking for these usability flaws and support the analysis of the interaction made by users.

The IVY Workbench is a tool initially focused on modelling and verification, which recently accepted the goal of combining formal model-based analysis with prototype interfaces. This objective is still under development. However, it is already possible to produce prototypes with some level of interaction combining models and UI mock-ups.

This work aims to develop improvements to this tool for these prototyping capabilities. The main objective is to automatically produce a web application capable of offering the expected behaviour and interaction as described in the model.

The results of this work will include an analysis of current prototyping methods, problems with these methods and possible solutions; analysis of different prototyping tools where functionalities and characteristics that may be useful for the IVY Workbench will be identified; development of an automatic system for generating web applications based on mock-ups and behaviour models.

Keywords: User interface, prototype, user-centred design, widget.

Resumo

Modelos formais baseados em protótipos interactivos

Sendo a UI um dos primeiros contactos que um utilizador tem com um sistema interativo, esta deve conseguir fornecer ao utilizador um fluxo de tarefas intuitivo e efetivo. Para isso, protótipos são normalmente construídos nas fases iniciais do processo de desenvolvimento de um software, para que seja feita a antecipação da fase de testes, e tanto os desenvolvedores como os utilizadores, tenham uma noção antecipada de como será o produto final, facilitando o desenvolvimento.

Em sistemas críticos, o método comum de prototipagem da UI não é suficiente. Estes sistemas requerem metodologias de análise mais avançadas e detalhadas para que falhas que possam originar em acidentes sejam evitadas. Para isso, existem abordagens com base em modelos e métodos de análise formal, complementadas com o processo de prototipagem, que suportam o processo de procura destas falhas de usabilidade, e a análise da interação feita pelos utilizadores.

Uma ferramenta originalmente focada na modelação e verificação, que recentemente aceitou o objetivo de combinar análise formal baseada em modelos com protótipos de interfaces, é a IVY Workbench. Este objetivo encontra-se ainda em processo de desenvolvimento. No entanto, já é possível produzir protótipos com algum nível de interação combinando modelos e mock-ups da UI.

Este trabalho visa a melhorar as capacidades de prototipagem desta ferramenta, sendo o objetivo principal a possibilidade de a produção automática de um protótipo web capaz de oferecer o comportamento e interação descritos no modelo.

Os resultados deste trabalho incluirão uma análise dos métodos atuais de prototipagem, problemas desses métodos e possíveis soluções; análise a diferentes ferramentas de prototipagem onde serão identificadas funcionalidades e características que poderão ser úteis para a IVY Workbench; desenvolvimento de um sistema automático de geração de protótipos web com base em mock-ups e modelos de comportamento da interface.

Palavras-chave: User interface, protótipo, user-centred design, widget. ...

Contents

List of Figures	xi
Acronyms	xiv
1 Introduction	1
1.1 Contextualization	1
1.2 Motivation	1
1.3 Aim of the work	2
1.4 Document structure	2
2 State Of The Art	3
2.1 User-centred Design	3
2.2 Prototyping	5
2.2.1 Prototyping dimensions	5
2.2.2 Prototyping types	8
2.3 Prototyping Tools	10
2.3.1 Mock-up Editors	10
2.3.2 Model-based tools	11
2.4 Problems of Interactive System Prototyping	13
2.5 Combining Models with user-centred design	14
3 Current State of The IVY Workbench	16
3.1 IVY's Prototyping Plugin	16
3.2 Limitations and Requirements	18
3.2.1 Limitations	18
3.2.2 Requirements	19
3.3 Technologies	20
3.4 Summary	23

4	Interactive Prototypes	24
4.1	A Modelling Convention for WIMP UIs	24
4.1.1	IVY Formal language	24
4.1.2	General Approach	26
4.1.3	WIMP UI Representation Convention	28
4.2	JSON Converter	31
4.3	JavaScript Generator	33
4.3.1	Web Server	37
4.4	Summary	37
5	Results and Analysis	39
5.1	Graphic Calculator Casio fx-CG50	39
5.2	Formal model	40
5.3	UI Mock-up	42
5.4	Generated files and Simulation	43
5.5	Summary	45
6	Conclusion	50
6.1	Results	50
6.2	Future work	51
	Bibliography	53
	Annexes	
I	Listings	57

List of Figures

1	UCD process and its activities represented as rectangles (image from [12] adapted from [22]).	4
2	Horizontal prototyping representing features versus vertical prototyping representing functionality.	6
3	Low and high-fidelity prototypes representation.	7
4	Prototyping plugin interface (image from [12]).	17
5	Example of a counter app	26
6	Example with two views and more actions	27
7	All the Casio fx-CG50 Mock-up views	40
8	Mock-up Screen ON View	41
9	Browser connected to the local web server	48
10	Example of a simulation with the results	48

List of Listings

3.1	Comparison between XML and JSON	20
3.2	Javascript code using DOM to retrieve the HTML content	21
3.3	Example of JavaScript code in a HTML file	22
4.1	IVY MAL language model example with only atributes and actions	25
4.2	IVY MAL language model axioms example	25
4.3	Deontic axiom example	26
4.4	Example of a model using IVY MAL Language	27
4.5	Convention - Interactors disposition	28
4.6	Convention - Redirection Logic	29
4.7	Convention - Visual representation of the elements	30
4.8	Convention - Interaction representation	30
4.9	"Example of a generated JSON from a model with 3 interactors: main decrementer and incrementer	32
4.10	Example of the actions and attributes converted to JSON	33
4.11	Example of the axioms converted to JSON	34
4.12	Structure with redirect action	35
4.13	Generated Javascript code from the Listing 4.12 model	35
4.14	Structure with process action	36
4.15	Generated Javascript code from the Listing 4.14 model	36
5.1	Example of the SVG elements identified by the model's attribute and action names	43
5.2	JSON generated from "screen_off" interactor	44
5.3	JavaScript generated from 5.2	44
5.4	Minimal JSON generated from "screen_on" interactor	46
5.5	Number button JavaScript code	47
5.6	Calculation logic JavaScript code	47
I.1	Final structure	57

I.2	Example of a JSON generated from the model from listing I.1	58
I.3	Calculator Prototype's formal model	60
I.4	Calculator Prototype's generated JSON	63
I.5	Calculator Prototype's "screen_on" generated JavaScript	72
I.6	Calculator Prototype's "screen_off" generated JavaScript	76
I.7	Calculator Prototype's "screen_number" generated JavaScript	77

Acronyms

API	Application Programming Interface 18 , 22
CSS	Cascading Style Sheets 11 , 21
DOM	Document Object Model 21 , 22 , 23 , 36
HTML	Hyper Text Markup Language 11 , 21 , 22 , 33 , 36 , 37 , 51
JSON	JavaScript Object Notation 19 , 20 , 23 , 24 , 31 , 32 , 36 , 37 , 39 , 43 , 45 , 49 , 50 , 51
MAL	Modal Action Logic 13 , 16 , 18 , 19 , 20 , 23 , 24 , 25 , 26 , 29 , 31 , 33 , 38 , 40 , 45 , 50 , 51
PVS	Prototype Verification System 12
SVG	Scalable Vector Graphics 16 , 17 , 21 , 33 , 36 , 37 , 49 , 51 , 52
UCD	User-centred Design 3 , 10 , 11 , 50
UI	User Interface 3 , 8 , 9 , 11 , 12 , 13 , 19 , 20 , 23 , 24 , 26 , 28 , 33 , 37 , 50 , 51
UML	Unified Modeling Language 13
WIMP	"windows, icons, menus, pointer" 19 , 20 , 23 , 24 , 26 , 28 , 33 , 37 , 50 , 51
XML	Extensible Markup Language 20

Introduction

1.1 Contextualization

A program's interface is crucial in the user's experience with the software, and problems with graphical interfaces are one of the main reasons for application failures [37]. Ways to rigorously analyse the behaviour of the interface become essential to guarantee the quality of an interactive system. The user interface constitutes the main point of interaction with the program's logic, and the most common solution for communicating the design of an interface is by using prototyping tools [2].

A model-based approach encourages a more sustainable development process and a better understanding of a system design. Combined with prototyping, it facilitates user participation in the development and testing of interfaces. Prototyping and testing do not guarantee the absence of errors, but they minimize the probability of their occurrence [10].

Some existing tools combine user-centred design with rigorous analysis of interfaces. However, there is still no tool that fully supports modelling, prototyping, and analysing. The IVY tool, in particular, supports model verification but is limited in prototyping [9].

1.2 Motivation

Creating well-designed and intuitive interactive systems requires much time and effort. This process can be particularly challenging when developing safety-critical systems due to their complexity. Model-based approaches are a typical solution to deal with complexity.

This type of approach, however, requires deep knowledge of modelling languages and technical expertise [9].

This work focuses, through the IVY Workbench, on developing a model-based approach capable of combining interface prototyping and formal analysis in order to produce rich interactive prototypes. By expanding the tool's capabilities in this way, it will be possible to include in the analysis potential users who are not experts in formal modelling and verification.

1.3 Aim of the work

The main objective is to analyse tools that allow the creation of mock-ups of interfaces and, through the IVY Workbench modelling tool, generate interactive prototypes. These prototypes will be made using Web technology and, for that, we will need a translator of the models to JavaScript or a JavaScript interpreter that can read the model and animate it [32] to create a simulation of the control logic of these interfaces [13].

The expected solution should then allow importing of mock-ups generated in vector-based prototyping tools, importing a navigation map for the mock-up (generated by the IVY tool), and generating a web application that simulates the interaction with the mock-ups according to the behavioural model of the interface.

1.4 Document structure

The dissertation has the following structure:

- Chapter 1 is an introductory chapter where are presented the contextualization, motivation, and aim of the work.
- Chapter 2 will discuss the state of art. This chapter presents previously carried out studies regarding the methodology of user-centred design, techniques and types of prototyping, the most used prototyping tools, and relevant prototyping problems. The chapter ends with a possible solution to the issues presented.
- Chapter 3 explores the features and capabilities of the current IVY workbench and prototyping plugin.
- Chapter 4 describes the contribution to the IVY Workbench.
- Chapter 5 validates the work by presenting a realistic prototype example using the work done.
- Chapter 6 presents the conclusions, the results of the work done and identifies possible future work.

State Of The Art

Tools that can model and analyse user interface are needed, as [User Interface \(UI\)](#) issues are often encountered in the development of software for various applications. Moreover, the lack of tools and support for assessing and resolving these issues often leads to system failure. Tools for modelling and analysing critical user interfaces typically require the expertise of various engineering disciplines, such as domain experts, human factors engineers, and software engineers. These tools combine formal verification and user-centred design. They support various levels of analysis, from the study of user tasks to the analysis of interactive system components. Developers must make the right choice when choosing a tool for their projects because. In most cases, choosing the wrong tool can lead to errors and frustration.

This chapter describes the [User-centred Design \(UCD\)](#) methodology and its relevance to critical systems design and provides examples of the various tools that support prototype development.

2.1 User-centred Design

This section describes user-centred design, a process which is focused on the needs of the users [22]. It also discusses alternatives for technologies that are focused on the user experience.

Considering the design of the user interface in the initial stage of a project's development process is beneficial and will improve its quality [11]. A well-reasoned problem is often the key to successfully developing a design concept. When the problem is poorly conceived, the efforts to implement it may be vain. According to researchers, user centred design can also improve the performance of teams during software development [26].

To achieve that, the [UCD](#) process defines four fundamental principles:

- **Since the beginning, users are involved in the process:** A clear understanding of the users' requirements will not be obtained unless they are involved from the start. Doing so can prevent from going too far in developing a solution that is not designed to work seamlessly for them.
- **Importance of requirement clarification and empathy:** Trying to adopt the user perspective should be a first step to understand their pain points and then develop a product that will meet their

needs.

- **Introduce multiple user feedback loops in the life cycle:** Gather various types of data to evaluate and improve the product. This data helps the team make more informed decisions.
- **Iterative design process:** The product is improved using the feedback obtained on the last stage, and the process repeats until the results meet the requirements.

With the key principles in mind, the adoption of the user-centred design process will be easier, and, like the principles, there are some usual steps to help at this stage. Many product teams rely on the Hasso-Pitzer Institute of Design's five-stage design process [3] to develop their products. **Research** where it should aim to develop a deep understanding of our users' needs, **define** where that needs and requirements are defined, **design** where prototyping is carried out, **evaluate** where user feedback is considered again, and **iterate** repeatedly until the goal is reached. User-centred design can be used for all phases of the design process. It can help the team better formulate the requirements and avoid wasting time, so the final process [22] will be something like (see figure 1):

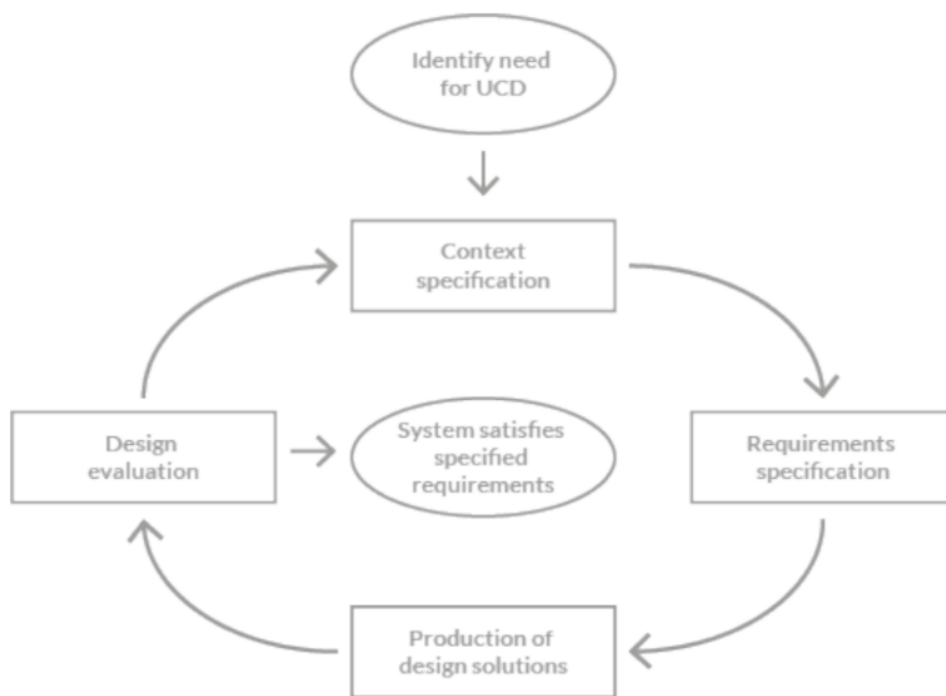


Figure 1: UCD process and its activities represented as rectangles (image from [12] adapted from [22]).

- **Specify the context of use:** Identification of the people who are going to use the product and what conditions they will use it in.
- **Specify requirements:** Identification of the business requirements and user goals that must be met to successfully implement the product.

- **Create design solutions:** Step that may be completed in stages, starting with the bare-bones wireframe.
- **Evaluate designs:** Integral part of any good software development project, it typically involves conducting user testing with actual users. This will help gather valuable feedback and insight.

2.2 Prototyping

The product development communities have acknowledged Early-stage prototyping in recent years. Contrary to conventional wisdom, prototyping is not limited to designing a product or a service [15]. Instead of validating ideas, it is better to use prototyping to stimulate imagination and build to think. This process can be beneficial for improving the quality of finished products.

So, prototypes help designers formulate ideas [7]. They promote collaboration and communication between various parties involved in the design process, allow early evaluation by collecting user feedback throughout the design process with usability tests, anticipate potential issues, and mitigate errors in the early stages.

This section discusses the various design principles and how prototypes should be utilized in product development. It aims to provide a comprehensive overview of the multiple facets of prototyping and its multiple applications.

2.2.1 Prototyping dimensions

Sometimes, horizontal and vertical (see figure 2) prototypes can be used during the design and analysis phases. However, they can benefit from specific requirements, but they can also present some pitfalls.

Prototypes can vary in the breadth and depth with which they cover the featured and functionality of the target system. To acknowledge this distinction, we can talk about horizontal versus vertical prototypes.

Horizontal

Prototypes are commonly used during the early stages of analysis. They provide a broad view of the application with sample screens and menus, without fully implementing the underlying functionalities and features.

This type of prototype helps describe the breadth of a system and how features will be brought together within a particular layer. They can also help gather ideas about how components will be assembled and allow the user to understand the system's functions, but only with limited functionality [23].

Vertical

Prototypes are a step-by-step approach that shows the complexity of a system's features and functions. It is typically used in the later design stages to drill down to specific details.

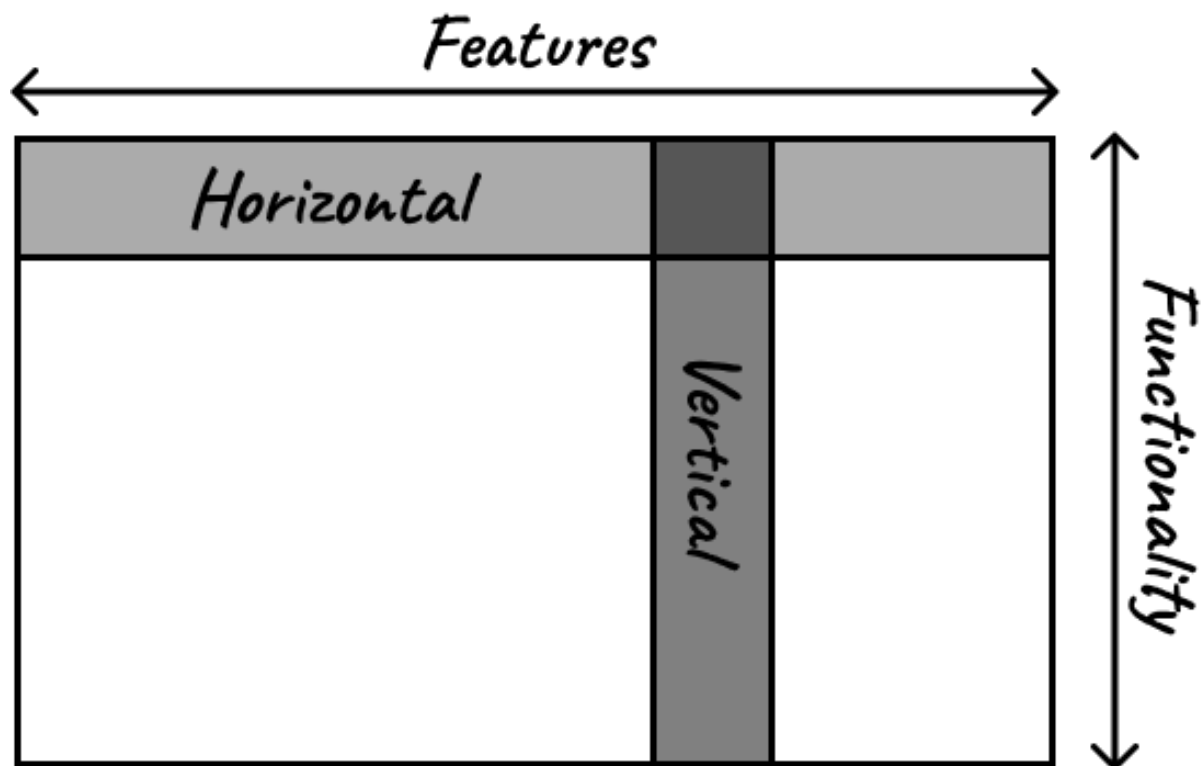


Figure 2: Horizontal prototyping representing features versus vertical prototyping representing functionality.

Unlike traditional prototypes, vertical prototypes try to provide a detailed view of part of the application. They do not attempt to provide full details, but rather, they try to implement a specific feature, or set of features, in a more refined manner. With a deep dive into a system's core functionality, users can explore it in depth through all its layers, allowing them to get a deeper understanding of its various components [23]. Hence, they can be useful, for example, to validate technological decisions.

Regardless of being Horizontal or Vertical, prototypes can further be characterized, according to four dimensions [5]:

- Representation
- Precision
- Interactivity
- Evolution

In the **representation** dimension, prototypes are divided into two sorts of representation: offline and online. Offline prototypes are usually created using paper sketches, storyboards, and videos. They do not require a computer to operate, are typically made quickly and discarded when not used. They allow the designer to explore various design alternatives and pick the most effective solution. Online prototypes are

usually created on a computer. They generally have a more complex creation than the offline representation and may require specific experts.

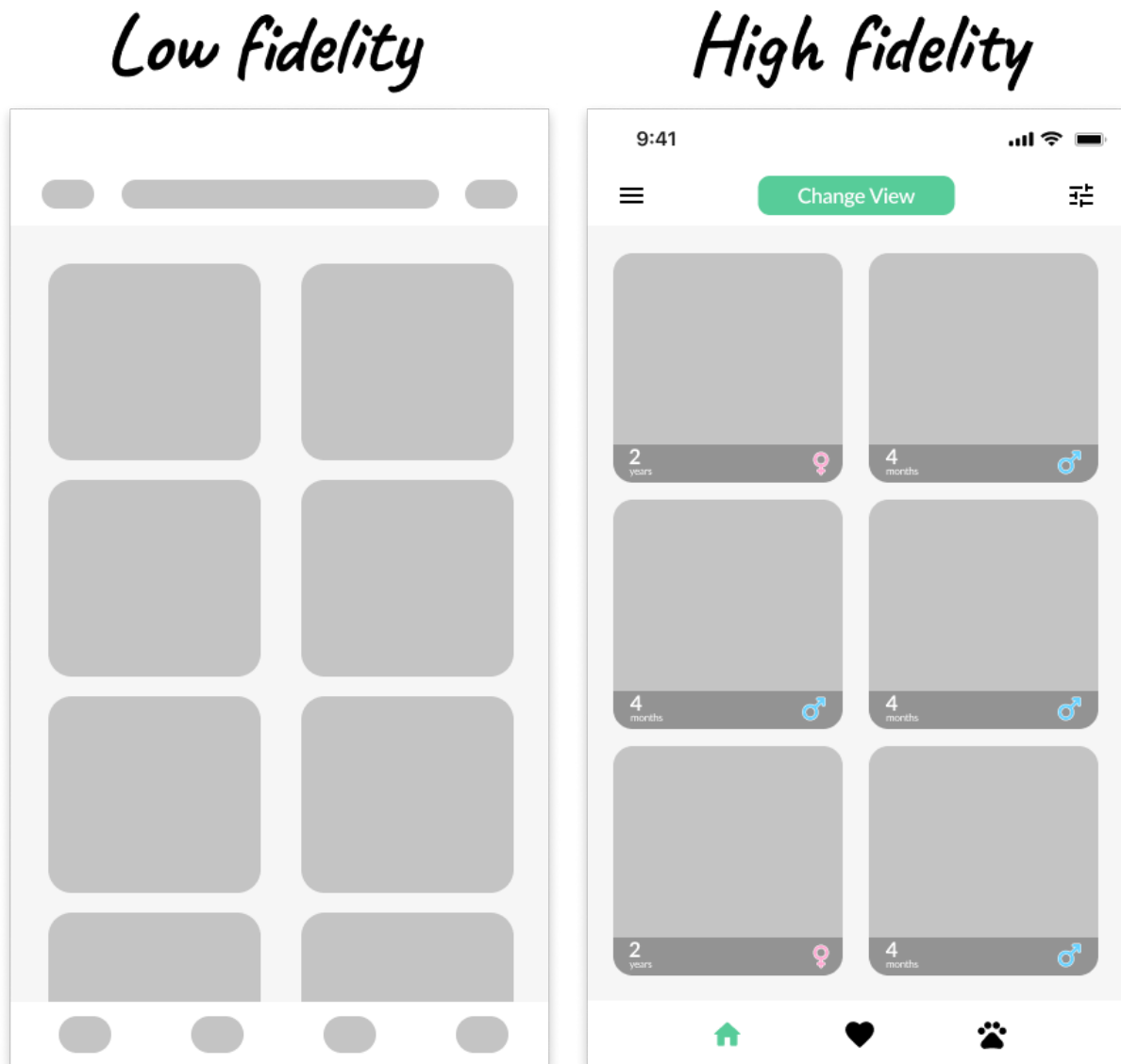


Figure 3: Low and high-fidelity prototypes representation.

The level of **precision** typically increases with the number of details that are included on the prototypes. By providing explicit representations of the system being built, prototypes help designers and engineers to explain the system's design and make it confidently. Instead of describing the user's actions, prototypes imply that the system displays the results. Details relevant to the prototype's goal are often present in sketches. The level of detail of the representation needs to be balanced against the time needed to reach it. However, even though it can require much time to be more precise, it has many advantages. The more detailed the prototypes are, the more information can be used for future evaluation

and iterated. A way to analyse the level of precision on prototypes is by characterising them by fidelity (see figure 3):

- **Low-fidelity prototypes** can help designers identify potential design issues at an early stage of the development process and usually are carried out by paper sketches or simple screen mock-ups.
- **Medium-fidelity prototypes** focus on the interactive aspects, such as the navigation between components and functionality, and have a more detailed look and feels than a lower-fidelity version.
- **High-fidelity prototypes** are more expensive and require more time to develop than low- and medium-fidelity ones. They are also helpful in identifying design problems that may not be addressed with low and medium-fidelity prototypes. Usually, low and even medium fidelity prototypes are not enough for safety-critical domains.

Designing an effective interaction is very important for interactive systems. At the same time, prototyping **interactivity** is also not an easy task as it requires a detailed definition of how the system will work and be used. Often, too much focus is given to the system visual, ignoring user interaction and the user flow [5]. This dimension, usually the larger it is, the greater the user's satisfaction, as the focus is on the efficiency and effectiveness of the solution. Prototypes are classified into three types based on their level of interactivity: fixed, fixed-path, and open [5]. A fixed prototype previews the system but does not support user interaction. A fixed-path prototype is a type of **UI** that shows little interaction between the user and the component it is designed to represent. Typically, there is only one behaviour that can be demonstrated. Open prototypes allow developers to test and modify the behaviour and flow of their systems to determine how users will interact with the system.

In the **evolution** dimension, prototypes have varying lifespans that can have different purposes and are chosen depending on the environment where they are being developed. The following section will discuss these types and give detailed information about each one.

2.2.2 Prototyping types

Today, the four most common types of prototyping utilised in the software industry are [21]:

- *Throwaway/Rapid prototyping* that refers to building a prototype that will probably be discarded in the end.
- *Incremental prototyping* that allows the final product to be developed as a series of smaller prototypes [36].
- *Evolutionary prototyping*, where a prototype is built in an organized approach that is progressively refined to create the system.
- *Extreme prototyping* is a type mainly focused on web development application [14].

Throwaway/Rapid prototyping

Throwaway prototyping can be represented both offline and online prototypes. The most common offline and online representations would be paper prototypes and away wireframes, respectively. This is because, as the name implies, these are prototypes that will later be thrown away. Therefore, it is essential that, in the early stages of the design process, building the prototypes is not time-consuming, but that it has an indispensable value for the communication, exploration and evaluation of the design of the interfaces [4]. Both paper prototypes and wireframes do not require much investment in terms of time and resources.

This prototype type is indicated to offer a quick view of the interfaces, focusing on the position of the UI elements, the layout [5]. Furthermore, it is possible, through them, to make quick decisions on which areas of the interface can be occupied and which can be left open for future elements to be discussed.

Incremental prototyping

In incremental prototyping, separate small prototypes are built in parallel [21]. Every single prototype is tested separately and integrated with the other prototypes at the end of the development process [19]. The feedback time between the application development team and the user can be reduced by this approach [36].

However, with incremental programming, the look can change a lot from prototype to prototype, so the modules can be different from the others [21]. To ensure consistency, the design team must come up with some guiding principles in advance and communicate with each other frequently.

Evolutionary prototyping

Evolutionary prototyping is focused on developing prototypes that evolve into components or whole systems [5], and are a subset of iterative prototypes. The goal is to build a usable system or product that is gradually, keeping the system adapting to the changes that arise in the environment [29].

Unlike the previous approaches, evolutionary prototypes involve more planning and practice. This process can be viewed as building a series of versions, where the design is refined without being replaced [7], and tested before they are officially used. Yet, this process tends to introduce more commitment to the work than finding a particular solution, reducing the amount of time invested in exploring more alternatives.

Extreme prototyping

This approach is more widespread in projects involving the creation of web apps [19]. The user interface is designed and developed before any of the technology is put in place [21]. This is what makes it “extreme”. The process can be described by the following steps [21] [14]:

1. The designers work on the HTML wireframe first;
2. The wireframes are transformed into functional pages;

3. The services layer is implemented and connected with the final prototype.

The main advantages of this approach are the incorporation of all the development specifics, and the time wasted is short before showing a working app to the client.

However, extreme prototyping focus mostly of the time on coding and not on design. It also involves changes that are hard to document, and not every design can code this type of prototype.

2.3 Prototyping Tools

In the initial development of a project, one must decide what prototype category will be used, and for what purpose. In this way, the choice of a prototyping tool will be made to suit the intended objective.

Tools of two different types will be presented in this section: vector mock-up editors and tools based on formal models and methods. The main focus is not to decide or evaluate the best tool but the selection and collection of possible features to be implemented in the IVY workbench tool in the future. For this, tools that promote the principles of the [UCD](#) methodology will be taken into account.

2.3.1 Mock-up Editors

Getting the right tools and the proper training to use them is essential in minimizing the time spent designing and implementing prototypes. Mock-up editors are commonly used to enable users to visualize the system's physical design. It can generate ideas for new features and functionalities [\[5\]](#).

Since interactive prototypes are significant for the current project, it is more appropriate to consider tools that can produce such setups instead of fixed mock-ups. Examples are: Adobe XD and Figma.

Figma

Figma is a web-based graphics editor that allows to create user interfaces and graphic design work [\[16\]](#). Figma is similar to other apps, like Adobe XD, in its basic features, but with some significant differences. Figma gives users a free plan, making it ideal for graphic designers who are new to prototyping.

Unlike other graphic editors, Figma works seamlessly on any web browser. It is a design tool that lets anyone collaborate on files without installing additional software. One of its greatest strengths and Figma main focus, is its ability to allow users to collaborate live, in real-time. This feature is especially beneficial for teams constantly adding new changes and features. Real-time collaboration helps prevent design drifting, which occurs when an idea is developed while a project is in progress and suddenly deviates from the agreed-upon design.

Figma has various tools that will help complete the design phase of a project. These include tools for fully-fledged illustration and built-in libraries that allow designers to create reusable components [\[17\]](#). Figma features a flexible and powerful system to handle vector shapes called "vector networks". Unlike traditional vector paths, that connect two points, "vector networks" allow paths to connect multiple points.

It also features a prototyping feature [18] that eliminates the need for other tools to create fixed-path animations. This specific feature can be explored to develop a simpler version of the user flow, but it still needs a lot of work.

Finally, Figma supports in-app comments, allowing designers to discuss their work and organize the problems, fixing the issues faster and coordinatively.

Adobe XD

Adobe Experience Design is a tool that enables designers to create detailed mock-ups, using similar features to other Adobe tools, such as Photoshop and Illustrator [35]. This tool has three primary functions: design, prototyping, and sharing.

The design feature lets designers create mock-ups with various highly customisable elements and allows them to import various widget libraries from the Adobe XD community. It is also possible to reuse components of drawings, which can enable several instances of the same component to have their unique attributes. Other features that make it easier to create prototypes are attaching constraints to the elements, ensuring the components are responsive (i.e. supported for various devices and screen resolutions), and creating multiple states. A state defines actions that a specific user can trigger, such as a keyboard press or a hover. These states are also associated with visual changes in the mock-up that are triggered at the moment of the action.

The prototyping feature allows designers to simulate the flow of the [UI](#) by allowing pages to be connected and animations between the pages created. This way, it is possible to run fixed-path prototype simulations. It also allows users to export their prototypes to [Hyper Text Markup Language \(HTML\)](#) and [Cascading Style Sheets \(CSS\)](#), but they do not contain any internal logic, and the results are not perfect, as expectable. For example, the resulting [CSS](#) frequently relies on absolute position values, which is inefficient.

The sharing feature allows designers to easily share their completed prototype with the team, contributing new ideas and improving the product. They can review and see all the changes' history. However, these sharing features have some limitations on the free version. Everything else is free.

Finally, Adobe XD has Android and iOS mobile apps that allow users on their mobile devices to connect to their accounts and test the prototypes they have generated.

2.3.2 Model-based tools

Tools based on models and formal analysis are not the most adopted in the market due to the need for a higher level of knowledge. However, when properly used, they facilitate the detection of problems and failures in the early stages of the project.

This section will discuss two model-based tools: PVSio-Web and IVY Workbench. These tools were chosen because they support the [UCD](#) methodology, like the tools mentioned in the previous section, and besides verification they can produce prototypes for validation.

PVSio-Web

PVSio-Web [28] is a web-based environment that enables designers and engineers to formally model, verify and prototype interactive systems. The tool is built on top of the [Prototype Verification System \(PVS\)](#).

The front-end of PVSio-Web is a graphical interface that enables the user to quickly prototype interactive systems. The PVSio-Web user generates realistic prototypes by first loading a user interface picture and then linking it to a formal model. The programmable interface has visual areas that allow the user to specify how actions should be translated into [PVS](#) expressions, and also serve to represent the configurable visual elements for the prototype. PVSio-Web prototypes have been used in training materials for healthcare facilities and device manufacturers to raise awareness about the various issues with [UI](#) software [27].

The concept of PVSio-web is based on the Model-View-Controller architecture, which separates the modules responsible for the prototype's behaviour from those that provide its visual appearance. This tool incorporates the following essential environments:

- The **Simulator View** is used to execute prototypes within PVSio-web. It logs all actions that the user performs during the course of their interaction with the prototype. The translated user actions are then sent to the [PVS](#) model, and the visual elements of the prototype are rendered in real-time with the response from the model.
- The **Prototype Builder** supports the generation of a prototype. It provides a graphical environment that enables the user to define the prototype's visual aspect and create programmable overlay areas that are mapped to the model to allow interaction with the prototype.
- The **EmuCharts Editor** is a module that enables developers to create formal models by drawing Emucharts (a variant of state transition diagram). Model generators can generate representations of the state machines in different languages (including [PVS](#)).
- The **Model Editor** is a text editor that enables developers to create formal models in [PVS](#). It provides the usual features of modern IDEs.
- The **PVSio Wrapper** is a module that supports sending commands to PVSio and receiving callbacks when the model returns a result.
- The **Co-Simulator** is a module that enables the exchange of simulation data between [PVS](#) models and other simulation environments. It is especially useful for the development of complex systems.

Summing up, PVSio-Web SDK offers a high-fidelity prototype generator that allows developers to import visual images from any source. It features a set of predefined [UI](#) elements and is designed to evolve as the design process continues.

IVY Workbench

IVY [9] is a tool for modelling interactive systems. It consists of a set of tools that provide a front end for the NuSMV model checker. The tools support a conceptual notation called Modal Action Logic, which enables the specification of complex interactive systems [9]. A template-based approach helps develop properties that are required for the analysis of models. The goal of IVY was to provide tools that are more accessible to developers and end-users. Its architecture is organized into an easily accessible set of components.

The IVY toolkit now consists of six components:

- **Modal Action Logic (MAL) editor:** This component supports the viewer's ability to visualize the structure of MAL models. The language used for modelling is based on the MAL logic, and provides mechanisms for describing interactors in terms of a state and set of actions, available to the user through a presentation medium.
- **Property editor:** The properties of a model are formulated using the CTL notation. This component can also generate properties from templates instantiated with attributes or action names derived from the MAL model.
- **Trace visualizer:** This component can produce visualizations of counter-examples. The counter-example can be described in various formats, such as a table or, an Unified Modeling Language (UML) activity diagram.
- **Model Animator:** with this plugin, the IVY user can generate traces dynamically. The plugin can be used to explore alternative paths and perform analysis of "what if" scenarios. It uses two representations: a tabular one and a state-based one.
- **Prototyper:** this plugin focuses on the UI design. It has two different modes: the animate mode and the edit mode. The editor mode allows designers to import a mock-up of their user interface and add action elements that can be used to interact with the prototype, such as clicking and holding a keyboard, which can be linked to the behaviour defined in the MAL editor. The animate mode allows users to simulate the interaction between the prototype and the user, using what has been configured on the edit mode.

2.4 Problems of Interactive System Prototyping

A traditional user-centred design approach focuses on the user's tasks early in the design process and then develops the design incrementally. The evaluation phases focus on scenarios and sketch designs, which represent different design solutions, and user studies to gather feedback and judge the design's appropriateness. These methods help gather essential insights about how devices are used and provide critical information about different users' needs.

However, in safety-critical environments, where failure or incorrect use could lead to loss or severe harm, such as hospitals or aircraft cockpits, the level of assurance that traditional UCD techniques can provide is not enough to prevent these issues. Even if the prototype is clearly defined, manual inspection does not provide the necessary level of assurance [20]. Safety-critical systems must be developed using robust engineering techniques to make sure they behave and are used correctly [6].

As a concrete example, the Smiths Medical Medfusion syringe pumps were used in hospitals and intensive care units to deliver blood, drugs, and other fluids to a patient's body. After some use, some versions of the Medfusion 4000 and 3500 syringe pumps were recalled due to a software issue that could cause them to over-delivery and under-delivery [1], which could cause serious health complications. The issue occurred if the pumps were not appropriately setup or were initiated after a specific sequence of events.

Another example is a car model produced by Ford, the Lincoln MKC. All 2014 Lincoln MKC vehicles manufactured from September 2014 to September 2015 were subject to recall due to various problems that could cause the vehicle to shut down suddenly [25]. The ignition button on the vehicle was close to the radio buttons and the touch screen, and when the drivers were using that area, just by resting the palm was sufficient to click the ignition button and shut off the vehicle while in motion, having the possibility of causing a road accident.

In conclusion, mistakes similar to these are not at all uncommon. Ignoring compliance requirements leaving with costly redesign processes; overthinking the initial design by making the wrong adjustments and focusing only on the visuals which causes a waste to prototyping; forgetting about testing during the prototyping phase which leads to costly errors, delays and missing specific error cases; are all examples of these types of mistakes. Therefore, one way to solve most of these problems is to combine and include models in this prototyping process, which will be discussed in the next section.

2.5 Combining Models with user-centred design

The approach to improve the current user-centred prototyping method and to solve most of the problems mentioned in the last section consists of including a formal model of the system in the process to separate verification. With that said, the solution combines the formal modelling technique with user-centred design [6]. The design process will consist of the following steps:

- **Step 1:** The first set of user-centred requirements is formulated using contextual inquiry. The requirements are then validated or supplemented by other requirements relevant to the model's current state.
- **Step 2:** The formal model is developed to represent the system's functionalities. The model is checked using formal methods to ensure that it meets the requirements defined in the last step. This step is carried out to validate the correctness of the proposed specification and conditions.

- **Step 3:** A mock-up prototype is created to present the system's intended functionalities to end-users. The visual appearance of the prototype is then driven by the formal model developed in Step 2. This step aims to validate the requirements of the project. It does so by looking for potential model and system issues.
- **Step 4:** Requirements are proved using formal methods, such as model checking or proving approaches. The formal model is then modified when necessary. Requirements can also be modified as needed, and end-users are notified when modifications are made, so they can reiterate.
- **Step 5:** All steps are repeated until a good design has been produced.

This approach requires establishing a formal verification tool that can support both the verification and animation of the models. The IVY Workbench supports automated verification, through model checking, but its prototyping abilities are limited. In this paper, the focus will be on improving the tool, so it can generate animation for the resulting prototype through the model. This way, this project will be focused on the step 3 of the approach.

Current State of The IVY Workbench

This chapter will describe the features and capabilities of the current IVY workbench and its prototyping plugin. Furthermore, current limitations will be identified. Once that is done, an overview of the requirements to be fulfilled in this dissertation, together with the technologies that will be used to achieve them, is put forward.

3.1 IVY's Prototyping Plugin

The IVY Workbench is composed of five components that provide different functionalities, as already discussed in Section 2.3.2. The focus of this dissertation is directed to the last component mentioned in that section, the Prototyper Plugin. The main goal of this plugin is to support the exploration of the formal model, written in the MAL language, by joining a visual representation of the user interface with the model to create a prototype. For this, Scalable Vector Graphics (SVG) mock-ups are used, and elements in them associated with action in the models, generating interactivity. The mock-ups can be created in the majority of tools able to export SVG files. The use of the plugin is separated in two modes: editing mode and animation mode.

In editing mode, users can import the SVG file with the mock-up, visualize it, and map the attributes and actions of the MAL model to the mock-up elements. These functionalities are provided by three areas in the user interface of this mode (see Figure 4). The SVG renderer, SVG tree sidebar, and the states and events sidebar. The SVG renderer is responsible for rendering the imported SVG. Elements can be selected by clicking them, facilitating the interaction of the user with the mock-up at the time of configuration. This selection is visible in the second area, the SVG tree sidebar, which lists the hierarchy of the imported SVG file, excluding those elements that are not supported by the tool for interactions. As well as listing, the tree sidebar also supports selection, as the visible elements are selectable. It can also group the dynamic widgets by the root element. IVY Workbench dynamic widgets are SVG files with included JavaScript code to specify their behaviour. Other features of the tree sidebar are configuring an element's visibility, and inserting and removing elements from the file.

The states and events sidebar allows the user to map the mock-up elements to the formal model.

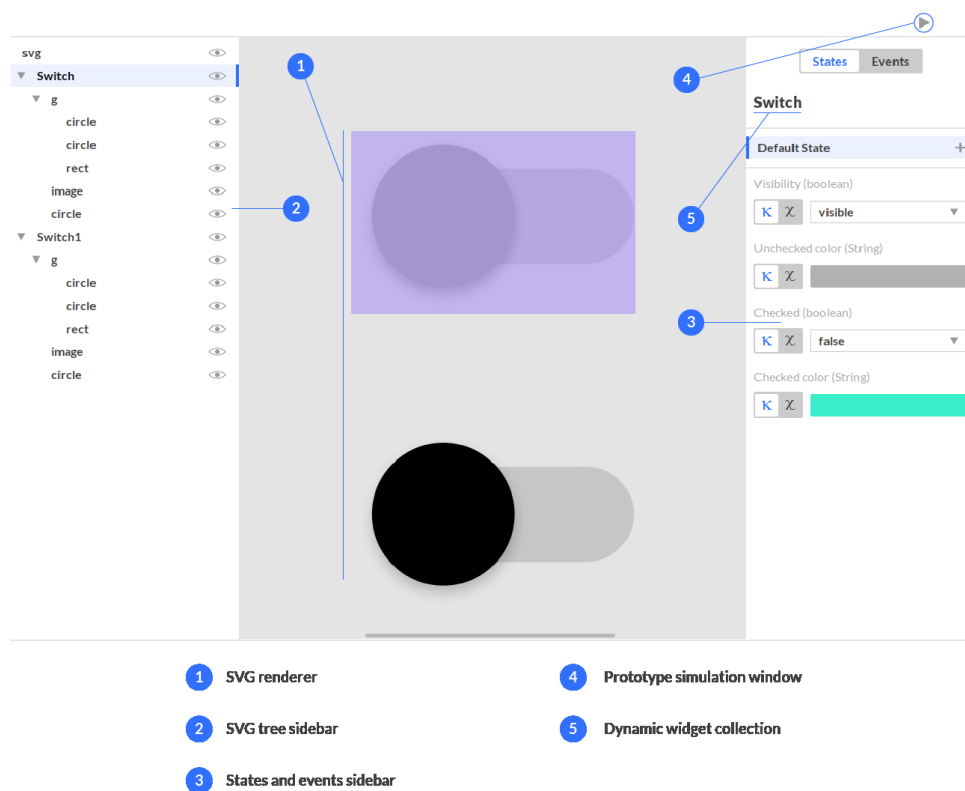


Figure 4: Prototyping plugin interface (image from [12]).

Each [SVG](#) element in the prototype can have multiple states. This sidebar allows the IVY user to define one or more states to the selected element. For example, in Figure 4, the top element of label 1 (a switch widget) has been selected. Boolean values can then be associated with its checked or visibility attributes at label 3. The element will thus have states associated to it. The attributes can have a default state that represents its initial appearance. Users can modify or add new states that trigger when a specific condition is met. These are conditional states, triggered when the model matches the criteria based on its value. At the time of simulation, the environment checks whether any of the conditional states of an element matches its criteria and the corresponding state is displayed. The event configuration allows users to associate multiple events to prototype elements. An event has a trigger, associates with an action of the formal model and executes a function of the prototype's widgets [12]. The trigger is defined by user interactions, or by periodic timer events. The functional component allows the appearance of the prototype to be changed without the help of the formal model. This feature allows the creation of advanced prototypes without increasing the complexity of the formal models created.

When the configuration and associations of the formal model with the imported mock-up is done, the state changes and animations are visible in the animation mode. In this mode, the plugin starts by initializing the simulation environment, receiving the [SVG](#) document and all the configuration made with the formal model of the prototype. When the initialization finishes, the environment starts the simulation. This simulation is visible through a [SVG](#) renderer. A sidebar is also present, which displays the possible

actions of the formal model. Users are allowed to interact with the prototype by triggering the defined events, or the actions in the sidebar. Finally, the simulation environment uses IVY's internal messaging system to communicate with the NuSMV model checker. All the interaction made by the user with the simulation view is sent to the model checker, which will then be applied to the model, calculating the next state and updating the simulation representation.

Finally, there is the opportunity of saving all the work carried out so that it is possible to pick up where it was off later. This ensures that there is no need to repeat the settings every time the plugin is opened.

3.2 Limitations and Requirements

3.2.1 Limitations

The IVY Workbench and the prototyper plugin have four major limitations.

As previously referenced, the NuSMV model checker is currently used to animate the prototypes. It uses the input from the prototype to calculate the next state of the model. This presents two limitations. NuSMV is a command line tool that does not provide an API. Hence, it must be run as a separate process, and communication is done by reading from and writing to its standard output and input respectively. This is a brittle solution and indeed newer version of NuSMV use a readline library that is not accessible to IVY. Additionally, this means that prototypes are not portable or easily accessible through a web based solution. They can only be run if NuSMV is installed. An initial version of a JavaScript generator was proposed in [12], to explore the feasibility of animating prototypes without the need for NuSMV. The tool, however, was a proof of concept with limited functionality.

Presently, there is no interchange format to which the model can be exported, or from which it can be read. In other words, if a developer has any idea for a tool that uses [MAL](#) models, they will either have to write a compiler, or they will have to write the tool as a IVY plugin, so it can access the existing compiler and model [Application Programming Interface \(API\)](#). This means that there is little flexibility in tool development.

In addition to the issues mentioned above, there is one more concerned with the modelling itself. The [MAL](#) language is geared towards analysis, not implementation. There is no convention for modelling specific user interface styles (e.g. how to model menus or windows). Designers will have to define their own conventions. This, in itself, is not a problem, but means that the conventions might not be compatible with each other and, more relevant in this case, makes it impossible to define a general approach to map a model to a prototype. This is the reason that, until now, most prototyping efforts in IVY were essentially restricted to one panel interfaces, composed of displays and input controls.

In summary, the problems indicated above are:

- The current conversion of animation is not scalable, is resource heavy and does not support navigation between mock-ups;

- Lack of an interchange format that contains the information on the IVY model;
- Lack of convention for a model structure that represents specific types of UI;
- JavaScript generation from the model is not fully developed;

3.2.2 Requirements

Considering the current version's limitations, a number of opportunities for improvements were identified. In this case, the definition of an interchange format for the formal model using a standardized, such as [JavaScript Object Notation \(JSON\)](#); the definition of a convention to model "windows, icons, menus, pointer" (WIMP) user interfaces, so their realistic prototype might be created; and a transition of the [MAL](#) models into JavaScript that delivers interactivity to the mock-up without the need to rely on NuSMV.

The following list identifies the requirements derived from these chosen constraints:

1. **The work done must allow IVY Workbench to export the formal model to [JSON](#) without losing information.** In the current version of IVY, there is no interchange format to which users can export. This prevents projects around IVY to be easily developed, given the difficulty associated with parsing the model. This requirement aims to solve this problem by providing an easily importable and parsable file.
2. **The work done should include the definition of a convention for how to structure the formal model that is easy to understand and more clearly describes the behaviour of [WIMP UIs](#).** Currently, the formal model is mostly used for verification and validation of a system, but does not assume any specific interaction style or type of UI. As IVY's formal language has a good modelling ability, this improvement arises to study the best way to, through this language, define a structure whose goal is to represent [WIMP UIs](#) and navigation between them. If necessary, controlled changes to the language can be made to improve the representation of the UI.
3. **The structure investigated should be as generic as possible, so it can describe most of [WIMP UIs](#).** The research should consider that the framework will be used to describe different types of [WIMP UIs](#). It should be able to describe more common examples of real systems without requiring changes to the language for now, and at the same time leave room for future changes, i.e. it should be scalable.
4. **The work done should support a more flexible and scalable animation of the [MAL](#) models** This requirement is only possible with the previous ones implemented, since the [JSON](#) will be used to generate JavaScript. This comes from the desire to be able to animate a mock-up using a formal model. In a way, it will also serve as an example of a possible use for the generated [JSON](#) format, since this JavaScript will be the result of reading that file.

```
1 <menu id="sidebar">
2   <popup>
3     <item value="Home" onclick="Home()" />
4     <item value="About" onclick="About()" />
5     <item value="Contacts" onclick="Contacts()" />
6   </popup>
7 </menu>
8
9 The same text expressed as JSON:
10
11 {"menu": {
12   "id": "sidebar",
13   "popup": {
14     "item": [
15       {"value": "Home", "onclick": "Home()"},
16       {"value": "About", "onclick": "About()"},
17       {"value": "Contacts", "onclick": "Contacts()"}
18     ]
19   }
20 }}
```

Listing 3.1: Comparison between XML and JSON

5. **The simulation of the interactivity of the prototype must be possible to access on desktop and mobile devices.** The work done will enable [WIMP UIs](#) of different resolutions to be simulated. An example would be a responsive web application in the early stages of development. As it is a responsive application it means that it will be compatible with mobile devices. This way, being able to access the simulation of the prototype on any device will ensure that it can be tested on the devices that the application to be developed should be compatible with.

3.3 Technologies

JSON

[JSON](#) [24], which stands for JavaScript Object Notation, is a format used to structure data in text format and transfer it between systems, such as in client-server or mobile applications. The [JSON](#) specification appeared around the year 2000, but only became part of the JavaScript language after the release of version 5 of ECMAScript. Currently, this format is supported by several types of programming languages, being a lighter alternative to the [Extensible Markup Language \(XML\)](#) data format (Listing 3.1 shows a comparison of a same object between XML and JSON). It is important to note that, despite its origins in the JavaScript programming language, it can be used in different programming languages and systems. The data contained in a [JSON](#) format file must be structured as a collection of names and value pairs or as an ordered list of values.

Since in this format the information is grouped by key and value, it fits well with how information is represented in the [MAL](#) language of IVY. Thus, it was decided that it would be the basis for the new

```

1 <!DOCTYPE HTML>
2 <html>
3 <head>
4   <title>DOM EXAMPLE!</title>
5 </head>
6 <body>
7   <h1 id="first">DOM</h1>
8   <h2>Example</h2>
9   <script type="text/javascript">
10     var text = document.getElementById("one").innerHTML;
11     alert("The first heading is " + text);
12   </script>
13 </body>
14 </html>

```

Listing 3.2: Javascript code using DOM to retrieve the HTML content

interchange format to be developed in this work.

DOM

The Document Object Model is a standardized interface that allows browsers and scripts to manipulate the content of a web page [30]. Furthermore, it is used by the browser to render a web page. It provides a tree structured representation of the document and also defines methods that allow the structure, style, and content of the document to be changed. The [Document Object Model \(DOM\)](#) is used when the objective is to update a website or build an advanced user interface. It allows moving items around within a page or create [CSS](#) effects. With the [DOM](#), the user has an endless supply of possibilities. It allows, for example, applications to be created that change the page data without the need to refresh the web page. It is also possible to create applications that can be customized by the user and change the page layout. Furthermore, it allows getting data from the elements (like on the Listing 3.2) and to be dragged, moved, and deleted easily.

IVY Workbench uses this technology for handling the [SVG](#) and [HTML](#) elements of prototypes. These elements are linked to the formal model behaviours that include input events that users may initiate. This technology offers many possibilities for interactivity generation, depending on the development done. Elements can be added and removed from the prototype, the visibility of an element can be changed, and the appearance of the prototype can be manipulated during its simulation. This way, the [DOM](#) is an essential technology to implement the requirements proposed in this IVY Workbench development.

JavaScript

JavaScript is a programming language designed for web applications [31]. It is easy to execute and can be run from a web browser. Today, it is one of the most popular programming languages in the world [34]. This prominence is partly due to the way in which it allows integration and communication within a site.

```
1 <!DOCTYPE HTML>
2 <html>
3 <body>
4   <p>Before the script.</p>
5   <script>
6     alert( 'Hello, world!' );
7   </script>
8   <p>After the script.</p>
9 </body>
10</html>
```

Listing 3.3: Example of JavaScript code in a HTML file

The language is used in [HTML](#) documents, generating communications that are the basis of any activity within a page. JavaScript is responsible for writing commands for layouts, animations, click responses and even 3D environments.

Another significant consideration is the fact that JavaScript is a client-side language, meaning that it is executed by the client, in this case the browser. Therefore, it is not necessary to access servers, which generally involves more complex procedures. Thus, the user will be able to work from existing traditional browsers in the market. Without the need to create executables, the programmer can simply develop the scripts and insert them into [HTML](#) pages, as the Listing 3.3 represents. All this code is automatically interpreted by the browser and the process is repeated every time it loads the [HTML](#) page.

As this project uses [DOM](#) as the main dependency for handling and generating interactivity, by default the language this dependency needs is JavaScript. In this case, JavaScript is one of the technologies needed to achieve the defined requirements.

NodeJS and ExpressJS

Node JS is an open-source platform that allows JavaScript code to be executed at both the front-end and back-end level [33]. It was released in 2009 under the MIT license. The platform has several applications, among which can be mentioned:

- Web applications working in real-time;
- CLI applications (Client Line Interface).
- Creation of scalable [API](#);
- Creation of chat and instant messaging applications, among others;

Express.js is a fast framework used with Node.js, which facilitates the development of back-end applications [38]. It can also be used with templating systems to develop full-stack applications. Express.js is written in JavaScript and is used by several renowned companies around the world [38]. It is very popular

with both large companies and the community, making it easy to create applications using Node with JavaScript. This ecosystem is even more powerful as a result.

Express has several features that help simplify the development of the applications. Among its main features, it can be mentioned:

- It features a fully functional routing system;
- It allows the handling of exceptions within the application;
- It allows the integration of several template systems that simplify the creation of web pages;
- Made for rapid creation of applications using a small set of files and folders;

Additionally, one of the defined requirements is to allow the access of the prototypes in desktop and mobile devices, and ExpressJS covers it. It generates a server which can be accessed via a local IP on any device on the same network. This allows prototypes to be tested in different resolutions and devices without the need to have the files locally in every device it will be tested on.

3.4 Summary

The IVY workbench, in its current version, presents some limitations in its functionalities. This chapter evaluated the functionalities of the tool and the limitations they present. The generation of prototype's is limited by the NuSMV model checker and the lack of support for navigation between multiple interactors. The only way to compile a model is through the tool itself and there is no interchange format to which it can be exported, limiting the developments around the [MAL](#) language to the creation of plugins in the tool itself. Furthermore, being the language focused on system modelling, there is no convention modellers designers can follow to represent a [WIMP UI](#) without conflicting approaches between designers.

This chapter also presents a list of requirements for new functionality based on these limitations. A list of technologies that will be used to build these functionalities is also presented. This dissertation shall define a convention that can describe [WIMP UIs](#) using the [MAL](#) language to avoid discrepancy of approaches between modellers. In addition, the [JSON](#) format, a popular interchange format, will be used to export the model without losing any state or behaviour information. JavaScript and [DOM](#) will use the new modelling convention and interchange format to generate animation from the models. Using NodeJS with ExpressJS, the generated code will be hosted on an automated local server so that these prototypes can be tested and accessed on desktop and mobile devices on the same network.

The next chapter considers the requirements identified above and focuses on the development of the new functionalities. It details the defined convention, the new interchange format to which the model is exported, and the JavaScript generator.

Interactive Prototypes

The work done in this dissertation is focused not only on the IVY Workbench itself, it is also centred on establishing the conditions for the tool's future development.

The definition of a modelling convention for the MAL language that would support the modelling of **WIMP UI** was undertaken. This definition required changes to be introduced in the language. In addition, a new converter from the formal model to **JSON** was developed, creating an interchange format to the IVY Workbench. Furthermore, a parser responsible for generating JavaScript using the **JSON** file was constructed. This chapter describes each phase of the work carried out, starting with the investigation of a possible structure and the necessary additions to the formal language. Next, the strategy used to convert the formal model into a **JSON** file. Finally, the chapter ends with the description of a possible approach used in the JavaScript parser to generate animation.

4.1 A Modelling Convention for WIMP UIs

The aim of this research ultimately is to arrive at a modelling convention that can represent a **WIMP UI**. To this end, an introduction to the IVY Workbench modelling language and its capabilities will be necessary to have a notion of what it can currently do. A modelling convention will then be devised that replicates the interactivity of a realistic prototype and which is both readable and clear to those analysing it.

4.1.1 IVY Formal language

The IVY Workbench follows an approach focused on formal modelling and analysis of interactive systems [8]. It uses Modal Action Logic (**MAL**) as the modelling language. Models are structured into interactions, characterised by attributes that represent the state of the system, and actions and axioms that represent the behaviour of the system. Interactors are organized hierarchically and are thus composed of:

- a set of typed attributes that define the state of the interactor
- a set of actions that define the operations on the interactor state

```

1 types
2   int = 0..1000
3
4 interactor main
5   attributes
6     display: int
7   actions
8     increment

```

Listing 4.1: IVY MAL language model example with only attributes and actions

```

1 axioms
2   [] display=0
3   [increment] display'=display+1

```

Listing 4.2: IVY MAL language model axioms example

- a set of axioms that define the semantics of the actions that act on the interactor state

The attributes can also be decorated with modalities, only supporting currently the “vis” modality that represents the visibility of the attribute or the action. As axioms define how the state of the interactor is changed in response to the defined actions, this can be of three types:

- **Propositional axioms** – these axioms define conditions that are always true in the interactor state (i.e. invariants).
- **Modal axioms** – these axioms define the effect that an action has on the interactor’s state. They can also be used for the initialization of the value of an attribute;
- **Deontic axioms** – these axioms define the conditions on which actions will be allowed or required to be triggered. That is, they do not indicate any state change and serve only to limit when actions can be performed.

For demonstration purposes, the example of a counter is given in Figure 5. A counter is composed of a screen that will display the value which will be incremented, and a button that when clicked will handle this incrementation. So, in the MAL language, a model to represent the counter will be defined a type, an attribute, and an action. The type will be the representation of integers, and since types in IVY have to be finite, it will be limited to the range 0 to 1000. The attribute is the screen that will show the value, in this case the variable that will store it. It will suffer changes through the action that represents the click on the increment button. Thus, an example of the initial model would be the Listing 4.1.

It only remains to define the logic behind the counter, that is, to create axioms that represent the state transitions through the action. Since a counter only allows the increment of the value and this value starts at 0, the axioms that can describe this behaviour are represented in the Listing 4.2

It can be seen that the two axioms follow a similar structure: “[action] attribute’=new value”. These axioms are of the Modal type, in which within the square brackets (the model operator) is the action that

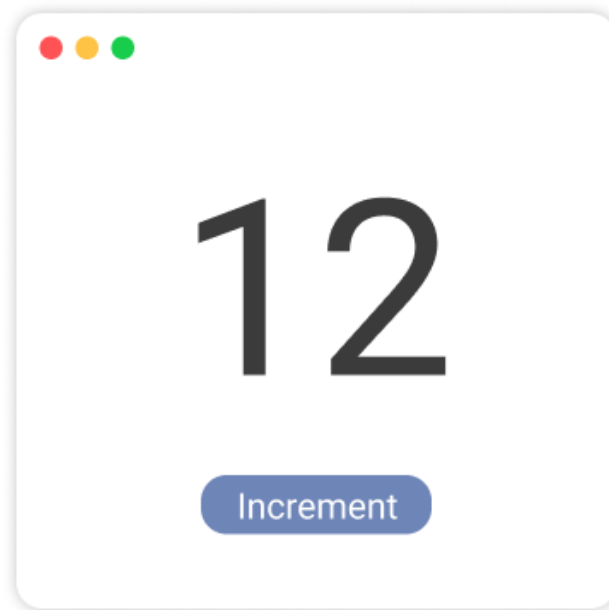


Figure 5: Example of a counter app

```
1 per(increment) -> display<1000
```

Listing 4.3: Deontic axiom example

triggers the change, followed by the definition of the changed attribute/s with the new value/s (the ' after the attribute means the value of the attribute after the action). If nothing is present in place of the action, it means that it is an initialization axiom, as is the case with the first one in the example. Given that type was defined as an integer in the range 0 to 1000, then for the system not to exceed the upper bound, the axiom is written as represented in Listing 4.3 is needed.

This new axiom is of Deontic type, i.e. to a specific action is assigned an expression that conditions when that action can be performed. Since in this example, there is not a value or something that is invariant, the Propositional axioms are not used. Their structure is the same as for the modal axiom, but without the model operator ($[]$). Since these axioms apply to states, not transitions, an action is not needed. Thus finalizing our model, Listing 4.4 would be the final structure.

It can then be concluded that a model in the IVY Workbench can be viewed as a state machine, where states are defined by attribute values, and the transitions by the actions that cause attribute changes and the axioms that describe these changes.

4.1.2 General Approach

The approach that was followed to achieve a description of a [WIMP UI](#) more efficiently was to analyse a real case. The objective is to describe this case using the [MAL](#) language of IVY and arrive at a convention

```

1 types
2   int = 0..1000
3
4 interactor main
5   attributes
6     [value] display: int
7   actions
8     [click] increment
9   axioms
10    [] display=0
11    [increment] display'=display+1
12    per(increment) -> display<1000

```

Listing 4.4: Example of a model using IVY MAL Language

that is simple to understand, and can also support the navigation between panels. To explain the whole process, an extended version of the example in Figure 5 will be used, ensuring that it encompasses what was defined in the requirements. So, the extended example is present in the Figure 6.

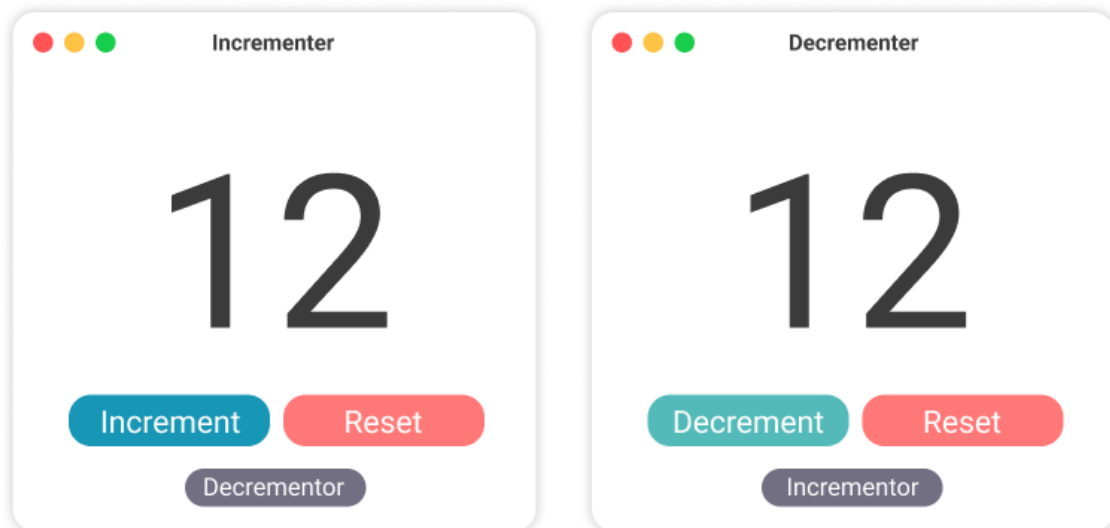


Figure 6: Example with two views and more actions

This prototype can be split into two views: the incrementer view and the decrementer view. The two are relatively similar visually, but are different in logic. The first one increments the counter and the second decrements it. Thus, it makes sense that the model guarantees that this separation exists, that it is evident and, at the same time, that it allows navigation between them, moving towards the fulfilment of the navigation requirement.

Going deeper into each of the views, we start by evaluating the buttons that are present, since they are, theoretically, actions that trigger changes in the prototype's state. As the two views are similar, we will just take one and perform the analysis. The first view has three buttons: increment button, reset button, and

```
1 interactor incrementer
2   ...
3
4 interactor decrementer
5   ...
6
7 interactor main
8   aggregates
9     incrementer via incrementer
10    decrementer via decrementer
11    ...
```

Listing 4.5: Convention - Interactors disposition

the button to navigate to the other view. Moreover, these buttons will generate behaviour that will change the state. In this case, being an incrementer, there is a value that will be incremented. That is, our model, besides the separation between views, will have to clearly describe the interactions in the prototype and the state variables that support the changes based on the actions performed.

So, to represent this prototype, our model will have to:

- Ensure all important elements in the [UI](#) logic are present in the system model;
- Ensure there is a clear separation and encapsulation of the logic of the prototype's pages within the model;
- Include a redirection logic between pages of a prototype;
- Include the actions and the type of interactivity that trigger them from each page of the prototype;
- Include variables that simulate and store the different states of the prototype;

4.1.3 WIMP UI Representation Convention

The main purpose of defining this convention for representing [WIMP UIs](#) is that it is obvious what behaviour this [UI](#) will have beyond the interactions that triggered changes to the state of each interactor. The defined convention considered the definition of visual elements that represent values and actions. Besides that, it was thought to make clear the separation of the existing views in the mock-up and still guarantee that there is a navigation logic between them. In what concerns interactions, these were thought to the point of requiring changes in the language to support them so that they could be properly described.

To explain what this convention consists of, the example in [Figure 6](#) will be used. Firstly, each panel in the mock-up is represented by an interactor. This way, it is clear that any interactor besides the main interactor composes a panel. These panels, in turn, will only be considered if the main interactor has aggregated them. Thus, the interactors will be organized in the main and there may be reusable interactors.

```

1 types
2   int = 0..1000
3   Page = {incrementer_page, decrementer_page}
4
5 interactor incrementer
6   ...
7   actions
8     redirect
9     ...
10
11 interactor decrementer
12   ...
13   actions
14     redirect
15     ...
16
17 interactor main
18   aggregates
19     incrementer via incrementer
20     decrementer via decrementer
21   attributes
22     page: Page
23   axioms
24     [incrementer.redirect] page'=decrementer_page
25     [decrementer.redirect] page'=incrementer_page

```

Listing 4.6: Convention - Redirection Logic

In Listing 4.5, where a [MAL](#) pseudo-model is present, the interactors that represent the mock-up views are represented between line 1 and 5. They are also aggregated in lines 9 and 10.

As the interactor main is responsible for knowing which interactors will be used, it makes sense that it is also responsible for redirecting between them. For that, a type (Page) is defined to represent the views existing in the mock-up (a name is defined for each of them). This type is used to control the redirection between the views in the axioms of the interactor main, identifying to which interactor the redirection action will redirect. For that to be possible, an attribute of this type is also defined, to represent the currently visible page. It can be seen then that, to represent a redirection, it is necessary to have a type that identifies the existing views, an attribute that indicates the current view, a redirect action that represents the element that will cause the change of active view, and an axiom that controls this logic.

In lines 8 and 14 of Listing 4.6, two actions were added to each of the interactors. These actions represent the visual element that will have the redirection function, which, looking at Figure 6, correspond to the buttons “Incrementer” and “Decrementer”. In the interactor main, lines 19 and 20, Modal Axioms are added that, using the actions of the respective interactors, change the current visible page.

Visual elements can be described in two different ways, depending on their function. Visual elements that will only represent values or data that can change are represented by attributes. Visual elements which provide interaction will be represented by actions, as was defined in the example of the redirection buttons on lines 8 and 14. Therefore, currently for each view in the example there will be three actions and

```
1 interactor incrementer
2   attributes
3     label: Page
4     display: int
5   actions
6     redirect
7     reset
8     increment
9   ...
10
11 interactor decrementer
12   ...
13
14 interactor main
15   ...
```

Listing 4.7: Convention - Visual representation of the elements

```
1 interactor incrementer
2   attributes
3     label: Page
4     [value] display: int
5   actions
6     [click] redirect
7     [click] reset
8     [click] increment
9   axioms
10    [] display=0
11    [increment] display'=display+1
12    [reset] display'=0
13    per(increment) -> display<1000
14    per(reset) -> display>0
15    ...
```

Listing 4.8: Convention - Interaction representation

one attribute, corresponding to the buttons and the number, respectively. In Listing 4.7 the new attributes and actions added are represented.

The interactivity of the system involves combining these attributes and actions in the axioms of each interactor. In other words, for each action, one or more axioms that interact with the existing attributes will be built. For each attribute, an axiom can also be created without action to initialize the value. Furthermore, to make it clear what each attribute and action is for, improvements were made in the language so that the modalities support different values besides “vis”. An example could be, through these modalities, identifying the attributes as “state” and “value”. The “value” attributes would be the visible values and the “state” attributes would be auxiliary variables for more complex behaviours.

Listing 4.8, on line 11, shows an example of an axiom that uses the “increment” action to add one unit to the “display” attribute. Looking at the action, it can be seen that the interaction required to trigger this change is a click. It can also be seen that the changed attribute is visible as defined by the “value”

modality. This attribute is also initialized as 0 (see initialization axiom in line 10).

Given that types are finite values, it is necessary that these constraints are present in the model. Not only are there constraints due to types, but also behavioural constraints that can be included. For this, the deontic axioms can be used, as already explained.

4.2 JSON Converter

The current version of the IVY Workbench, is capable of generating an SMV version of the models. However, exporting to this format is not intended to support model sharing. It was done to support the integration with NuSMV. The export to [JSON](#), on the other hand, comes as an addition to the tool so that there is an interchange format to which the [MAL](#) model can be exported. Even though the exporter to SMV does not have this objective, its export logic was used as a basis, to take advantage of the existing abstract syntax tree and the transversal approach.

This converter follows an approach where it iterates through the model, interactor by interactor, hierarchically, starting always with the main aggregations. That is, as the main will have aggregated interactors, these will be visited first and, as the processing is done, it will build the [JSON](#) file in that order. In the resulting file, these interactors will be organized as key-value pairs where the key will be the identifier of the interactor, and the value all the information related to that interactor. Interactors besides the main will be detailed first, and only then the interactor main is analysed in depth. Hence, following the previously defined convention, the last part exported will be the redirection logic. Even if the model does not follow this convention, the result will be valid. It can be said that this converter works regardless of the model, as long as it is valid.

The converter identifies each interactor by its interactor name and instance used. Assuming there are two aggregations with different names but the same interactor, these will be identified as “aggregation_name::interactor_name”.

The representation of an interactor is composed of three substructures: actions, attributes and axioms. These substructures, as in the interactors’ structure, are key-value pairs. The key represents the substructure it is describing, and the value is a JSON object describing it. The key “actions” is a listing of all the actions of the interactor to which it belongs. Each element of the actions’ listing is composed of its ID and its modality. The value of the “attributes” key follows the same structure as the actions. Its value is also a listing of the attributes that compose the interactor, and each element is composed by ID, type, and modality.

Axioms are structured in a more complex way. They also have values composed by key-values and are distinguished by the type of axiom they represent. There is the “inits” key where all interactor initialization axioms are grouped (line 27 of Listing [4.11](#)). For non-conditional modal axioms, the key is the action that constitutes it and the value describes the changes triggered by that action. Lines 19 to 21 represent an example of the structure of this axiom. For the same types of axioms but with the introduction of

```

1 // IVY MAL Model
2
3 ...
4 interactor incrementer
5   ...
6 interactor decrementer
7   ...
8 interactor main
9   aggregates
10     incrementer via incrementer
11     decrementer via decrementer
12     ...
13
14 // Generated JSON
15
16 {
17   "decrementer::decrementer": {
18     ...
19   },
20   "incrementer::incrementer": {
21     ...
22   },
23   "main": {
24     ...
25   }
26 }

```

Listing 4.9: "Example of a generated JSON from a model with 3 interactors: main decrementer and incrementer"

a condition, the representation in [JSON](#) only differs in the key. To this key, the condition is introduced following the format "action::condition". The value object is identical to the non-conditional modal axiom. Finally, axioms of the deontic type do not contain an object as a value, they only contain the condition by which they are constituted. Their key follows the same format as the key for modal axioms with condition, where the condition is replaced by the word permission, i.e. "action::permission". Line 25 of the [Listing 4.11](#) indicates an example of how deontic axiom is structured.

Briefly, the converter follows the following order of steps:

1. It analyses each of the interactors, analysing first those that are hierarchically aggregated;
2. For each interactor, the actions, and attributes are grouped with names, types, and modalities;
3. For each axiom, the type of axiom is considered and based on that it defines a specific structure so that no information is lost.

```

1  // IVY MAL Model
2
3  ...
4  interactor decrementer
5    attributes
6      label: Page
7      [value] display: int
8    actions
9      [click] redirect_page
10     [click] process_reset
11     [click] process_decrement
12     ...
13  ...
14
15  // Generated JSON
16
17  {
18    "decrementer::decrementer": {
19      "actions": [
20        { "name": "process_reset", "type": "click"},
21        { "name": "redirect_page", "type": "click"},
22        { "name": "process_decrement", "type": "click"}
23      ],
24      "attributes": [
25        { "name": "display", "type": "int", "attribute": "value" },
26        { "name": "label", "type": "Page", "attribute": "" }
27      ],
28      ...
29    },
30    ...
31  }

```

Listing 4.10: Example of the actions and attributes converted to JSON

4.3 JavaScript Generator

The current prototyping features of the IVY Workbench rely on NuSMV for behaviours. Creating a JavaScript version of the model will eliminate that dependency, and allow prototypes to be run in different types of prototypes. There is already a JavaScript code generator from [MAL](#) language on the current version of the IVY Workbench, but is just a proof of concept with limited functionality.

Using the convention defined for the representation of [WIMP UIs](#), it is possible to generate animation without the need to write code. These process implies establishing additional conventions regarding how model and mock-ups are linked so that the prototype is correctly generated:

- The IDs of in the model must match the IDs of the [HTML](#) or [SVG](#) elements. In order for the JavaScript generator to make associations between the mock-up elements and the model's actions/attributes, these IDs must be the same.
- Each action must use a nomenclature that defines the type of action which is in the format "type_actionname", being the current existent types: "redirect" and "process". This is a necessity

```

1  // IVY MAL Model
2
3  interactor decrementer
4  ...
5  axioms
6  [] display=1000
7  [process_decrement] display'=display-1
8  [process_reset] display'=1000
9  per(process_decrement) -> display>0
10 per(process_reset) -> display>1000
11 ...
12
13 // Generated JSON
14
15 {
16   "decrementer::decrementer": {
17     ...
18     "axioms": {
19       "process_decrement": {
20         "display": "display - 1"
21       },
22       "process_reset": {
23         "display": "1000"
24       },
25       "process_decrement::permission": "display > 0",
26       "process_reset::permission": "display > 1000",
27       "inits": {
28         "display": "1000"
29       }
30     }
31   },
32   ...
33 }

```

Listing 4.11: Example of the axioms converted to JSON

because, depending on the interaction, the behaviour can change. Associating an interaction type indicates to the generator, at the time of generation, how it should handle the action, making sure the changes passed in the axioms will be associated to the correct behaviour.

- Modalities “value” and “state” should be added to the attributes. That is, attributes that are visible in the mock-up and which will undergo changes, respectively. An example is a display element of a calculator that must have the modality “[value]”. The attributes that are not visible, but are responsible for storing temporary or state values, can have the mode “[state]”, but is not mandatory.
- In axioms, the changes to the attributes must follow a specific order. To prevent values override in JavaScript, attributes that use current or old values of other attributes, should appear first.

The action names are essential for JavaScript generation. As mentioned in the rules, three categories of actions are supported. The *redirect* type is used in actions to serve for redirection and navigation

```

1 types
2   int = 0..1000
3   Page = {incrementer_page, decrementer_page}
4
5 interactor incrementer
6   attributes
7     label: Page
8     ...
9   actions
10    [click] redirect_decrementer
11    ...
12   axioms
13    ...
14
15 interactor main
16   aggregates
17     incrementer via incrementer
18     decrementer via decrementer
19   attributes
20    [page] page: Page
21   axioms
22     incrementer.label=incrementer_page
23     [incrementer.redirect_decrementer] page'=decrementer_page
24     ...

```

Listing 4.12: Structure with redirect action

```

1 document
2   .getElementById("decrementer")
3   .addEventListener("click", () => (window.location.href = "/decrementer"));

```

Listing 4.13: Generated Javascript code from the Listing 4.12 model

between the panels existent in the system. An action is defined without an axiom in its interactor, but that indicates that the interactor has a redirection. Then an axiom in the main interactor is defined which indicates to what panel this redirection is performed.

Listing 4.12 presents the running example model with this additional change. It is visible that on the line 10 the action has the name “redirect_decrementer”, where “redirect” is the new action type and “decrement” the button ID which will trigger the redirection. This means that when generating the code, the JavaScript generator will assume that this action, via a click, will trigger a redirect. This redirect is represented in the axiom in line 23 of the model, and the generator will also take it into account. The code resulting from this generation is as shown in Listing 4.13.

The *process* type is used in actions that serve to process and display data. That is, assuming there is a screen that displays a value and this value is changed by buttons, the button action uses “process” as its type. It then parses the value that is present in the display and changes it based on the axiom of that action. There is no change limit, any attribute can be changed but depending on the modality, it will behave differently. That is, the attributes with the modality “value” will have visible changes in the

```

1 types
2   int = 0..1000
3   ...
4
5 interactor incrementer
6   attributes
7     label: Page
8     [value] display: int
9   actions
10    [click] process_increment
11    ...
12  axioms
13    [] display=0
14    [process_increment] display'=display+1
15    per(process_increment) -> display<1000
16    ...

```

Listing 4.14: Structure with process action

```

1 document.getElementById("increment").addEventListener("click", () => {
2   id = "display";
3   display = parseInt(document.getElementById(id).textContent);
4
5   if (display < 1000) {
6     document.getElementById(id).textContent = display + 1;
7   }
8 });

```

Listing 4.15: Generated Javascript code from the Listing 4.14 model

prototype. The attributes with the modality “state” will serve as auxiliary variables and will be altered in the system without visible representation.

The incrementer example can be used to illustrate this, resulting in the Listing 4.14. It is visible that on line 8 the value to be displayed on the screen has the modality “value”, so it will have a visual representation on the prototype. The axiom on line 14, which uses the “increment” action with the *process* type, changes the attribute for the incremented result. The JavaScript resulting from this type of action is the one shown in Listing 4.15. Since the attribute have the modality of “value”, JavaScript assumes it is a value represented in the view and therefore gets the value from the [SVG/HTML](#) element via the [DOM](#). Another important detail to mention is the condition present in line 5 generated through the Deontic axiom that limits the value presented based on the type associated with this attribute.

Briefly, the model, using the earlier defined conventions, when generating the JavaScript will follow the following process:

1. The model is converted to [JSON](#) for better structuring and parsing in JavaScript;
2. Analyses all the pages/panels of the model through the aggregations of the main interactor;

3. Each interactor is considered iteratively starting by defining the initial values of the attributes in global variables as per the initialization Modal Axioms of each interactor.
4. Next, a function is generated that, based on the actions made in the prototype, updates the state of the page to meet the permissions made in the Deontic Axioms model, thus blocking interactions based on the global state;
5. EventListeners are generated based on the Modal Axioms in the model so that each mock-up element that has the action ID represents the assumed changes.

4.3.1 Web Server

Since these mock-ups are controlled by JavaScript in [HTML](#) files, a server is needed to serve the files. Individually, or if there is only one page, this would be possible by opening the generated [HTML](#) file. However, if there are multiple interactors to simulate redirects and more complex behaviour, this would become hard to manage and scale. Furthermore, as one of the goals to be fulfilled is for these prototypes to be easily accessible to other devices on the same network. It will not be practical to have to pass the files directly to those devices. In this way, a Web Server was created using NodeJS technology together with the ExpressJS framework.

To get the most out of this Web Server, it is necessary to follow a certain folder structure already defined, placing the necessary files in the correct folders for its correct operation. Theoretically, the only files to introduce manually will be the [SVG](#) mock-ups with the same name as the interactors present in the model and the model itself. The remaining files, such as the generated [JSON](#) and JavaScript, will be automatically moved and imported into the correct [HTML](#) files. Then, the Web Servers in its initialization will use the model and [JSON](#) interpret which pages exist and are necessary for the proper functioning of the prototype. Finally, the Web Server opens a local server with port 8080 that will be accessible from any device on the same network and the URL will consist of "localhost:8080/{interactor_name}".

4.4 Summary

This project adds to the IVY Workbench improvements in model export and animation generation in prototypes. A convention has been defined to allow designers to describe [WIMP UIs](#) in standardised manner. This convention considers also, the mapping between models and mock-ups. In addition, a new feature has been introduced to the language so that the convention can describe various types of interactions.

Expanding the development environment of a tool and allowing external developments to take place with it is beneficial for its improvement. In this way, an exporter was implemented for a [JSON](#)-based interchange format, following the existing SMV export logic. This allows for developments with the same scope and without necessarily needing to be a plugin for IVY Workbench.

Additionally, a JavaScript code generator was developed. This generator is the basis for an alternative to the current NuSMV-based prototypes generator of the IVY Workbench. It supports multiple interactors in the [MAL](#) model and redirections between panels of the mock-ups. Through it, it is possible to obtain automatic behaviour and interactions without writing code.

The next chapter presents results of these new features. Through a real example that encompasses all the developed functionalities, the final result of an example prototype is described.

Results and Analysis

This chapter explores the features added to the IVY Workbench, described in Chapter 4. The prototype of a graphing calculator, the Casio fx-CG50, will be described, which covers most of the mentioned features. This chapter also covers the details of the [JSON](#) generator and the approach to export to JavaScript, using the mentioned prototype. Finally, it will describe how the simulation is done.

5.1 Graphic Calculator Casio fx-CG50

The Casio fx-CG50 is a calculator that in addition to the basic functions of a standard calculator contains various graphing features, such as four types of graph representation (sphere, cylinder, plane, and line). It also has relationships between expressions and these graphs, a function catalogue, connection to physical measuring devices, and more. This calculator has several panels that can be accessed through the menu or the quick access buttons. Therefore, the prototype of this device focuses on simulating the basic functions of a common calculator and on the simulation of navigation between panels.

Figure 7 represents the mock-up built for this device, created with Figma, where all the panels used for this results' evaluation are present. This prototype aims to represent the following behaviours:

- When a user presses the “AC/ON” button, the calculator screen should toggle between on (Figure 8) and off. The calculator will be off when the screen is dark and on when it displays any of the existing screens.
- When the prototype is on and displays the panel with a 0 or other digits (first panel), the prototype accepts input by the user pressing numbers, and the screen value will change based on the numbers pressed. Next, by pressing one of the operator buttons (add, divide, subtract or multiply), the screen will set the digits to 0 so that a new number can be entered to complete the mathematical operation. Finally, by pressing other numbers and the “EXE” button, the screen will display the value resulting from the mathematical operation performed.
- When the prototype displays the screen on and a user presses one of the quick access buttons below the display, the screen should display each of the panels corresponding to the buttons pressed.



Figure 7: All the Casio fx-CG50 Mock-up views

This prototype requires a formal model that specifies its behaviour to animate the mock-up. The following sections describe these components and the settings required to achieve the prototype simulation.

5.2 Formal model

The formal model which the prototype is based upon is composed of four interactors: one that represents the screen off, one that represents the main screen, one that represents all the panels accessible by the quick access buttons and the main one that controls all the navigation between panels. Each interactor is also composed by the attributes, actions, and axioms that will give behaviour to the mock-ups. Listing 1.3 (in page 60) shows the resulting MAL model.

The simplest interactor of the four is “screen_off”, which represents the screen being off. This only has one action that represents the AC/ON button, the only button which is functional when the calculator is off (line 10).

The “screen_number” interactor abstracts all the panels accessible from the top menu, the panel for performing calculations excluded. As these will not have a very complex behaviour, they only have the

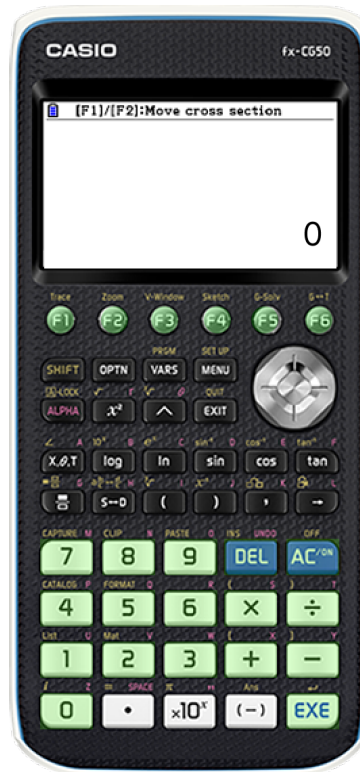


Figure 8: Mock-up Screen ON View

actions to represent all the buttons to switch between screens (line between 69 and 82).

The interactor “screen_on” is the most complex of the model. Starting with the attributes, it has an attribute that represents the value shown on the screen (line 17), an attribute that stores the screen value when a mathematical operator is pressed (line 18) and an attribute that stores the selected mathematical operator (line 19). Regarding the actions, considering that this interactor will be responsible for modelling the interaction with the numeric and operator buttons, for each button there is an action that represents it. Besides, like the interactor “screen_number”, it has actions for all the buttons to switch between screens (lines between 22 and 27). In what concerns axioms, for each number action there is an axiom that adds the pressed value to the right of the screen value (lines between 52 and 61). In the case of operators, these just save the respective operation in the state variable that represents the current operator (lines between 48 and 51). Then save the current value in the state attribute “old_display”, and set the value on the screen to 0 so that another value can be entered. Finally, there are the axioms for the EXE button that can only be triggered when there are two values to be calculated and an operation defined (lines between 63 and 66). There is also the axiom for the DEL button that only sets the value visible on the screen to 0 (line 62).

The interactor “main” differs from the previous ones in being responsible for redirections. The reason is for being the only one with the knowledge of the existence of the pages (interactors) existent in the prototype. This way, for each page, there will be an aggregator which will use the previous interactors

(lines between 86 and 92). As attributes, there is only one that represents the current page (line 94). To create the logic of redirections, in the axioms of the interactor main, for each interactor, an axiom is made per redirection that uses the actions previously defined in the interactors (lines between 96 and 143).

5.3 UI Mock-up

The UI mock-up, shown in Figure 8, represents the screen ON of the calculator. This contains SVG elements relevant to the generation of the prototype. As the screens in Figure 7 are all similar, only the mock-up of the screen ON will be used as an example, as it is the most complex.

As it can be seen that some buttons are present that are highlighted in green in the mock-up of Figure 8. As the mock-up is composed of a real image of the calculator, it is necessary that for each button used in the model, an SVG element is created and associated with the model. The solution to this was to place light green elements over the buttons to indicate that those buttons are “active” and that it is possible to interact with them. There are nine green elements over the number pad of the calculator. These are associated with the number actions of the formal model. The buttons will accept clicks, and this will change the value shown on the display, which associated with the display attribute of the model. There are also four buttons in the same area of the mock-up that represent the operators ($\times$, $\div$, $+$, $-$) for the possible calculations to be performed, represented in the model by the operators’ actions. These buttons will only be active for clicking after one of the numeric buttons has been pressed. The same happens with the “DEL” and “EXE” buttons, which will be active to erase the value on the screen and show the result of the operation on the screen, respectively. Each of these buttons is also controlled by an action in the formal model. The “AC” and “Fn” buttons change the screen or menu of the calculator that is currently visible. In the model the different screens are represented by interactors and these buttons are responsible for navigating between them.

As mentioned before, all the SVG elements of the mock-up that are highlighted in green have interaction in the final prototype. This happens because to each of them is associated an attribute or action that represents or controls it. However, this association requires some manual mapping on the SVG mock-up that is currently not possible to achieve automatically. This mapping involves ensuring that each mock-up element associated with an attribute or action in the formal model is identified by the same name as the attribute or action. That is, the calculator screen that will represent the value will have to have the ID “display” (see Listing 5.1, line 18) to be in concordance with the name of the attribute in line 17 of Listing I.3. The same happens for the buttons, but with a small difference. Given that actions in the model are written as “type_name”, the ID of the SVG element will be the name of the action, ignoring the type (see Listing 5.1, lines 8 and 15). By attributing these IDs to the SVG elements, the interactions described above become possible.

```

1  // MAL Model action and attribute
2  interactor screen_on
3    attributes
4      [value] display
5      ...
6    actions
7      [click] process_subtract
8      ...
9  ...
10
11 // SVG Element's
12 <svg ...>
13   <g id="screen_on">
14     ...
15     <rect id="subtract" .../>
16     ...
17     <text ...>
18       <tspan id="display" ...>
19         0
20       </tspan>
21     </text>
22   </g>
23 </svg>

```

Listing 5.1: Example of the SVG elements identified by the model's attribute and action names

5.4 Generated files and Simulation

Assuming the model of the calculator mentioned in the previous section, a demonstration of the simulation generated by the [JSON](#) converter and the JavaScript code generator will be made.

Starting with the screen off view, this presents only one state and one action. In [JSON](#), the structure generated for this view is as shown in Listing 5.2. The key representing the interactor is in the format “name::instance”. The value object contains the “redirect_power” action, with the click interaction as modality, and the attribute that only represents the name of this view, as it is initialized in the axiom present in the axioms key (line 19). As the only action of this interactor is a redirect to the linked screen, the axiom of the main interactor containing this logic must be considered. From this part of the [JSON](#) is generated the JavaScript present in Listing 5.3. The JavaScript call presented in this listing uses the modality “click” and the action type “redirect” to create the redirect action to the view “screen_on”.

Regarding the screen on, the interactor of this view is the one that contains the most complex logic, among all. Like the previous interactor, the “screen_on” also features the power button (line 14 of Listing 5.4). Both in [JSON](#) and JavaScript, the results are identical. This interactor, being more complex, presents more actions and attributes. The main attribute is “display” (line 18) which represents the value on the calculator's display. This value is then altered by clicking on the number buttons, each of which is represented by an action (line 4 is an example of a number button). These actions all also have the “click” mode. However, the types differ: the actions of the number buttons (“process_number”), reset (“process_reset”), operators (“process_operator”) and result (“process_equal”) are of type “process”;

```
1  ...
2  "screen_off::screen_off": {
3    "actions": [
4      {
5        "name": "redirect_power",
6        "type": "click"
7      }
8    ],
9    "attributes": [
10     {
11       "name": "label",
12       "type": "Page",
13       "attribute": "state"
14     }
15   ],
16   "axioms": {
17     "inits": {
18       "label": "screen_off"
19     }
20   }
21 },
22 ..
23 "main": {
24   "axioms": {
25     ...
26     "screen_off.redirect_power": {
27       "page": "screen_on"
28     },
29     ...
30   }
31 }
```

Listing 5.2: JSON generated from “screen_off” interactor

```
1 document
2   .getElementById("power")
3   .addEventListener("click", () => (window.location.href = "/screen_on"));
```

Listing 5.3: JavaScript generated from 5.2

the actions of the quick access buttons are of type “redirect” (“redirect_menux”). Comparing the model of Listing I.1 and the JSON generated from Listing 5.4, it is possible to see that all the information was correctly exported, including the axioms. For representativeness reasons, only a minimalist version of the JSON is presented here, which hides the repetitive parts. However, the complete JSON is present in Listing I.4 in the annexes.

To create the simulation, each of these JSON parts is transformed into JavaScript. Regarding the redirects for the quick access buttons, these present the same logic as the power button that powers the calculator off and on, as presented in Listing 5.2. Regarding the calculations, these include a bit more generated code. To simulate a calculation, a one-number button must first be pressed. This button will have the code shown in Listing 5.5.

By pressing one of the operator buttons, it saves the current display value in an auxiliary variable created for the “old_display” attribute (line 6 of Listing 5.4). It then sets the display value to 0 and saves the pressed operator in another auxiliary variable “oper” (lines between 6 and 10 of Listing 5.6). Since the previous display value is stored in the “old_display” attribute, any number now pressed will override the current display value. In the presence of values in the auxiliary variable of the old value, the operator and the current display value, the result button is unlocked. This behaviour is present in the view refresh function created via analysing the “equal” deontic axiom (line 44 of Listing 5.4) and is shown in Listing 5.6 between lines 1 and 5. Pressing the “EXE” button, mapped with the ID “equal”, a calculation is done through the values in the “display” and “old_display” variables, and updates the display to the resulting value (lines between 13 and 24 of Listing 5.6). There is also the “reset” button that when clicked will set the display value to 0.

The remaining views of the quick access menu present a similar behaviour to the off-screen view. The only differences are that besides the “power” button that powers off the calculator, there are also redirections to all the possible views of the prototype. For each quick access button present in the mock-up, an action is exported that redirects to the respective views. Again, for representativeness reasons, only a minimalist version of the JavaScript is presented here, which hides the repetitive function calls. However, the complete JavaScript code is present in Listing I.5 in the annexes.

Finally, the generated JavaScript files and the mock-ups are connected using the web server (see Figure 9). An example of the result’s simulation is presented in the Figure 10.

5.5 Summary

This chapter presents results based on a real example of the Casio fx-CG50 graphic calculator. This example encompasses all new features and puts to the test what has been developed in this project. It allowed testing how well the convention can describe the behaviour and interactions of a real example. The export of the MAL model to the new format is put to the test with a complete model, which includes redirections and calculations, aiming to test if all the information is exported. Finally, through the resulting

```

1 "screen_on::screen_on": {
2   "actions": [
3     ...
4     { "name": "process_one", "type": "click" },
5     ...
6     { "name": "process_subtract", "type": "click" },
7     ...
8     { "name": "process_equal", "type": "click" },
9     ...
10    { "name": "redirect_menu6", "type": "click" },
11    ...
12    { "name": "process_reset", "type": "click" },
13    ...
14    { "name": "redirect_power", "type": "click" },
15    ...
16  ],
17  "attributes": [
18    { "name": "display", "type": "int", "attribute": "value" },
19    { "name": "oper", "type": "OperatorType", "attribute": "state" },
20    { "name": "label", "type": "Page", "attribute": "state" },
21    { "name": "old_display", "type": "int", "attribute": "state" }
22  ],
23  "axioms": {
24    ...
25    "process_subtract": {
26      "oper": "subtract",
27      "display": "0",
28      "old_display": "display"
29    },
30    ...
31    "process_one": {
32      "display": "display * 10 + 1"
33    },
34    ...
35    "process_reset": {
36      "display": "0"
37    },
38    ...
39    "process_equal::oper = subtract": {
40      "display": "old_display - display",
41      "oper": "none"
42    },
43    ...
44    "process_equal::permission": "oper != none",
45    ...
46  }
47 },

```

Listing 5.4: Minimal JSON generated from “screen_on” interactor

```
1 document.getElementById("one").addEventListener("click", () => {
2   id = "display";
3   display = parseInt(document.getElementById(id).textContent);
4
5   document.getElementById(id).textContent = display * 10 + 1;
6   reloadInit();
7 });
```

Listing 5.5: Number button JavaScript code

```
1 function reloadInit() {
2   !(oper != "none")
3     ? (document.getElementById("equal").disabled = true)
4     : (document.getElementById("equal").disabled = false);
5 }
6
7 document.getElementById("subtract").addEventListener("click", () => {
8   oper = "subtract";
9   old_display = parseFloat(document.getElementById("display").textContent);
10  document.getElementById("display").textContent = 0;
11 });
12
13 document.getElementById("equal").addEventListener("click", () => {
14   id = "display";
15   display = parseInt(document.getElementById(id).textContent);
16
17   if (oper != "none") {
18     if (oper === "subtract") {
19       document.getElementById(id).textContent = old_display - display;
20       oper = "none";
21       reloadInit();
22     }
23   }
24 });
25
26 document.getElementById("reset").addEventListener("click", () => {
27   id = "display";
28   display = parseInt(document.getElementById(id).textContent);
29
30   document.getElementById(id).textContent = 0;
31   reloadInit();
32 });
```

Listing 5.6: Calculation logic JavaScript code



Figure 9: Browser connected to the local web server

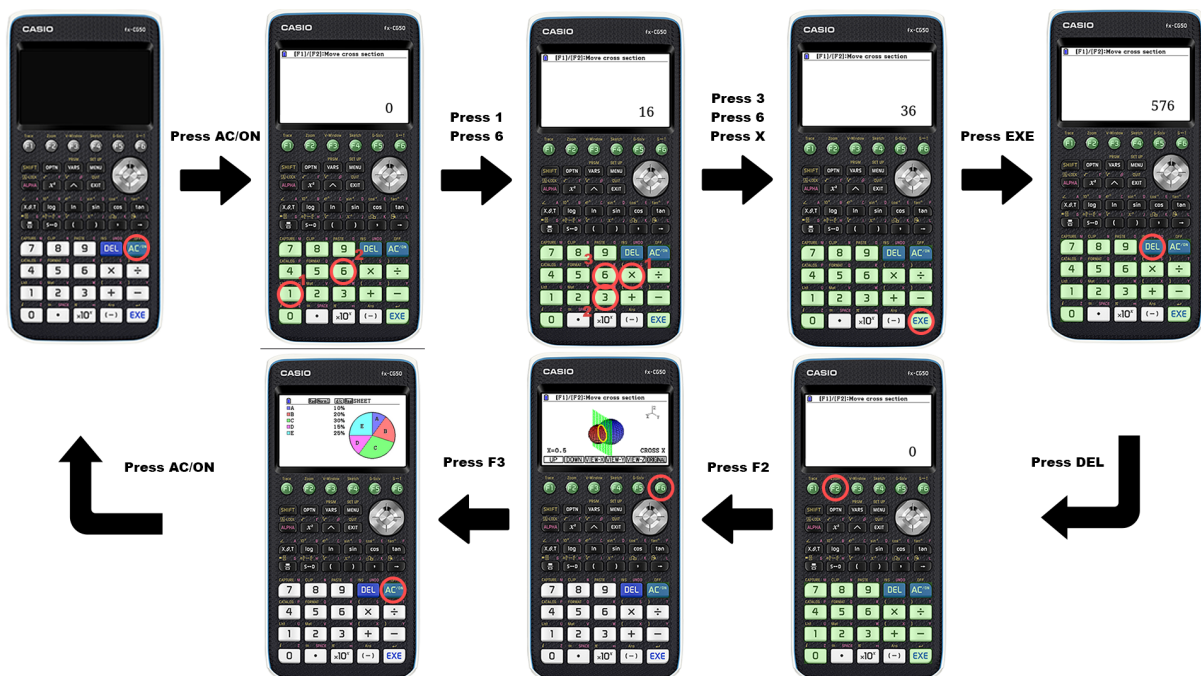


Figure 10: Example of a simulation with the results

[JSON](#), the JavaScript generator is also tested to understand if the animation and interaction previously described in the model is generated correctly.

In conclusion, this project implemented and achieved the requirements initially proposed. It resulted in an automatic animated prototype, with navigation between interactors, without the need to write code and only with the use of [SVG](#) mock-ups created in an external editor.

Conclusion

Prototyping is an iterative process that supports an early analysis of a system. This process is low cost and follows the concepts of the [UCD](#) methodology. Combining this common practice for the design of iterative systems with the development of formal models, capable of making a more exhaustive analysis of a system's behaviour, it is possible to achieve a more comprehensive analysis.

This work focused on expanding and combining model-based tools and user interface mock-ups to generate interactive prototypes. The existing version of the IVY Workbench tool already provided a solution for this combination, however, the technological solution adopted presented limitations, and could only handle single panel interfaces. Hence, a new solution was developed that addresses these issues. The contribution of this dissertation is included in the IVY Workbench and is able to generate interactive prototypes consisting of multiple panels and that can be run independently from the tool.

6.1 Results

IVY Workbench uses the [MAL](#) language to define models that describe the behaviour and state of a system. All the work done in this language and the models can be saved for later use. However, IVY provides no interchange format so, if a user wants to use the model outside the context of the tool, they will have to necessarily develop their own compiler, or else, develop a new IVY plugin to have access to IVY's compiler. To solve this problem, a converter from [MAL](#) to [JSON](#) interchange format was developed.

The models created in the IVY Workbench provide a description of UIs at a high level of abstraction. This gives modellers the freedom to model without considering concrete interaction styles and implementation details. However, this freedom comes with its own limitations. Two different projects might describe two WIMP interfaces using different modelling styles. This becomes a problem when we want to map those models to mock-ups of the interfaces, to create prototypes. In this dissertation, a convention was defined to describe [WIMP UIs](#) to avoid these incompatibilities. Furthermore, this convention can be used to develop behaviour generators.

Furthermore, in addition to the definition of the convention, some changes to the [MAL](#) language were required. This is because, in the language, attributes and actions can have modalities. Before modalities

were only used to defining the visualization of these elements in the system through only one modality, this is restrictive. In this way, the language was changed to support any modality, this being generic.

Using the new interchange format of the project, a JavaScript generator was developed to produce interactive prototypes. An example testing this generator was presented in Chapter 5.

The prototype is represented in the final result by [HTML](#) files where the [SVG](#) mock-ups and the JavaScript behaviour are inserted. In order for these files to be accessible and browsable, an automated web server was built. This server ensures that all the behaviour is loaded, including redirections between views. Moreover, even though it is a local server, it exposes a local IP and port that can be accessed by any device with internet access through the same network.

Finally, the new version of the IVY Workbench and the developed project solved the problems described in Section 3.2.1 of the previous version. Briefly, the contributions to the resolution of these problems were:

- A converter from the [MAL](#) language to the [JSON](#) interchange format, without losing information from the models. This converter removed the dependency on the tool to be able to read and analyse [MAL](#) models if an external project is to be developed.
- Definition of a convention to describe [WIMP UIs](#) using the [MAL](#) language. This convention was made to be scalable and that allows. In addition, the definition of the convention has meant that the [MAL](#) language has undergone improvements.
- Interactive prototypes based on the logic of the formal [MAL](#) model of a [WIMP UI](#), created by the developed JavaScript code generator. This approach requires the previously defined modelling convention to be followed for the resulting prototype to have the expected behaviour and animation.

6.2 Future work

As illustrated in Chapter 5, it is now possible to generate working prototypes. This section identifies lines of future work to further improve the approach.

The converter to [JSON](#) is in a position where most of the models will be able to be exported. However, there are features of the [MAL](#) language that were not fully explored, as they were not immediately useful for the prototypes (for example, actions with parameters). These should be analysed and, as next steps, support to the missing ones should be added.

Another possible improvement, indirectly related to the previous one, would be focused on improving the modelling convention. A study directed towards this improvement lead to further improvements in the JavaScript generator. For instance, it would be helpful to find a way to describe the behaviour of actions, without the need for rules in their naming.

Additionally, the modelling convention assumes a modelling style when one model axiom per action exists. Other styles are possible, with multiple axioms per action, and supporting them would make the approach more flexible.

Finally, it would be useful to integrate the work done herein with the [SVG](#) editor of the prototyper plugin, to ensure that the IDs of the [SVG](#) elements and the names of actions and attributes are correctly mapped. This functionality would reduce the manual work needed and facilitate the mapping of these IDs.

Bibliography

- [1] American Society of Anesthesiologists. *Medfusion 3500 and 4000 Syringe Pumps by Smiths Medical Class I Recall*. American Society of Anesthesiologists. Accessed: 2022-01-03. Feb. 2021. url: <https://www.asahq.org/advocacy-and-asapac/fda-and-washington-alerts/fda-alerts/2021/01/medfusion-3500-and-4000-syringe-pumps-by-smiths-medical-class-i-recall> (cit. on p. 14).
- [2] João Miguel Araújo, Rui Couto, and José Creissac Campos. “A Generator of User Interface Prototypes for the IVY Workbench”. In: *2019 International Conference on Graphics and Interaction (ICGI)*. 2019, pp. 32–39. doi: [10.1109/ICGI47575.2019.8955088](https://doi.org/10.1109/ICGI47575.2019.8955088) (cit. on p. 1).
- [3] Nick Babich. *User Centered Design Principles Methods*. Adobe. Accessed: 2022-01-03. Oct. 2019. url: <https://xd.adobe.com/ideas/principles/human-computer-interaction/user-centered-design/> (cit. on p. 4).
- [4] Brian Bailey, Jacob Biehl, Damon Cook, and Heather Metcalf. “Adapting paper prototyping for designing user interfaces for multiple display environments”. In: vol. 12. Mar. 2008, pp. 269–277. doi: [10.1007/s00779-007-0147-2](https://doi.org/10.1007/s00779-007-0147-2) (cit. on p. 9).
- [5] Michel Beaudouin-Lafon and Wendy Mackay. “Prototyping Tools and Techniques”. In: *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*. USA: L. Erlbaum Associates Inc., 2002, pp. 1006–1031. isbn: 0805838384 (cit. on pp. 6, 8–10).
- [6] Judy Bowen and Steve Reeves. “Combining Models for Interactive System Modelling”. In: Apr. 2017, pp. 161–182. doi: [10.1007/978-3-319-51838-1_6](https://doi.org/10.1007/978-3-319-51838-1_6) (cit. on p. 14).
- [7] Bradley Camburn, Vimal Viswanathan, Julie Linsey, David Anderson, Daniel Jensen, Richard Crawford, Kevin Otto, and Kristin Wood. “Design prototyping methods: State of the art in strategies, techniques, and guidelines”. In: vol. 3. Jan. 2017. doi: [10.1017/dsj.2017.10](https://doi.org/10.1017/dsj.2017.10) (cit. on pp. 5, 9).
- [8] J Creissac Campos. “High assurance interactive computing systems”. In: (2014) (cit. on p. 24).
- [9] José Campos, Camille Fayollas, Michael Harrison, Célia Martinie, Paolo Masci, and Philippe Palanque. “Supporting the Analysis of Safety Critical User Interfaces: An Exploration of Three Formal Tools”. In: vol. 27. Aug. 2020, pp. 1–48. doi: [10.1145/3404199](https://doi.org/10.1145/3404199) (cit. on pp. 1, 13).

- [10] José Campos and S. Mendes. “FlexiXML – A portable user interface rendering engine for UsiXML”. In: Jan. 2011 (cit. on p. 1).
- [11] Adriana Chammas, Manuela Quaresma, and Claudia Mont’Alvão. “A Closer Look on the User Centred Design”. In: *Procedia Manufacturing* 3 (Dec. 2015), pp. 5397–5404. doi: [10.1016/j.promfg.2015.07.656](https://doi.org/10.1016/j.promfg.2015.07.656) (cit. on p. 3).
- [12] Rafael Costa. “Animating user interface prototypes with formal models”. In: July 2021 (cit. on pp. 4, 17, 18).
- [13] David Dorran, Paul Leamy, and Ted Burke. “A Guide to Implementing Online Simulations using HTML, JavaScript and SVG images”. In: Mar. 2021. doi: [10.13140/RG.2.2.12283.23843](https://doi.org/10.13140/RG.2.2.12283.23843) (cit. on p. 2).
- [14] educative. *What are different types of software prototyping?* educative. Accessed: 2022-01-07. url: <https://www.educative.io/answers/what-are-different-types-of-software-prototyping> (cit. on pp. 8, 9).
- [15] Christer Elverum, Torgeir Welo, and Sigmund Tronvoll. “Prototyping in New Product Development: Strategy Considerations”. In: *Procedia CIRP* 50 (Dec. 2016), pp. 117–122. doi: [10.1016/j.procir.2016.05.010](https://doi.org/10.1016/j.procir.2016.05.010) (cit. on p. 5).
- [16] Figma. *Design features - It’s time to design on your terms*. Figma. Accessed: 2022-01-05. url: <https://www.figma.com/design/> (cit. on p. 10).
- [17] Figma. *Design systems features - https://www.figma.com/design-systems/*. Figma. Accessed: 2022-01-05. url: <https://www.figma.com/design-systems/> (cit. on p. 10).
- [18] Figma. *Prototyping features - Prototype while you design, and vice versa*. Figma. Accessed: 2022-01-05. url: <https://www.figma.com/prototyping/> (cit. on p. 11).
- [19] ADUK GmbH. *What Types of Prototyping Are There?* ADUK GmbH. Accessed: 2022-01-07. Sept. 2021. url: <https://aduk.de/blog/types-of-prototypes-i-what-types-of-prototyping-are-there/> (cit. on p. 9).
- [20] Michael Harrison, Paolo Masci, and José Campos. “Balancing the formal and the informal in user-centred design”. In: *Interacting with Computers* 33 (Jan. 2021), pp. 55–72. doi: [10.1093/iwcomp/iwab012](https://doi.org/10.1093/iwcomp/iwab012) (cit. on p. 14).
- [21] Matt Hull. *4 Types of Prototyping*. The AndPlus Blog. Accessed: 2022-01-03. July 2020. url: <https://www.andplus.com/blog/4-types-of-prototyping> (cit. on pp. 8, 9).
- [22] ISO. *ISO 9241-210:2019 Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems*. International Organization for Standardization. 2019. url: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=52075 (cit. on pp. 3, 4).

-
- [23] Malo Jones, Ingbert Floyd, and Michael Twidale. "Patchwork Prototyping: A Rapid Prototyping Technique That Harnesses the Power of Open-Source Software". In: (Dec. 2021) (cit. on pp. 5, 6).
- [24] JSON. *Introducing JSON*. JSON. Accessed: 2022-09-30. url: <https://www.json.org/json-en.html> (cit. on p. 20).
- [25] George Kennedy. *Lincoln MKC recalled for starter button placement while reaching for radio*. Consumer Reports. Accessed: 2022-01-03. Jan. 2015. url: <https://www.consumerreports.org/cro/news/2015/01/lincoln-mkc-recalled-for-starter-button-placement/index.htm> (cit. on p. 14).
- [26] Justin Lai, Tomonori Honda, and Maria Yang. "A study of the role of user-centered design methods in design team projects". In: *Artificial intelligence for engineering design analysis and manufacturing* 24 (Aug. 2010), pp. 303–316. doi: [10.1017/S0890060410000211](https://doi.org/10.1017/S0890060410000211) (cit. on p. 3).
- [27] Paolo Masci, Patrick Oladimeji, Paul Curzon, and Harold Thimbleby. "Using PVSio-web to Demonstrate Software Issues in Medical User Interfaces". In: July 2017, pp. 214–221. isbn: 978-3-319-63193-6. doi: [10.1007/978-3-319-63194-3_14](https://doi.org/10.1007/978-3-319-63194-3_14) (cit. on p. 12).
- [28] Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. "PVSio-web 2.0: Joining PVS to HCI". In: vol. 9206. July 2015. doi: [10.1007/978-3-319-21690-4](https://doi.org/10.1007/978-3-319-21690-4) (cit. on p. 12).
- [29] P.J. Mayhew and P. Dearnley. "An Alternative Prototyping Classification". In: *The Computer Journal* 30 (Dec. 1987), pp. 481–484. doi: [10.1093/comjnl/30.6.481](https://doi.org/10.1093/comjnl/30.6.481) (cit. on p. 9).
- [30] MDN. *Introduction to the DOM*. MDN. Accessed: 2022-09-30. Sept. 2022. url: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (cit. on p. 21).
- [31] MDN. *What is JavaScript?* MDN. Accessed: 2022-09-30. Sept. 2022. url: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript (cit. on p. 21).
- [32] Julius Mong and David Brailsford. "Using SVG as the rendering model for structured and graphically complex web material". In: Jan. 2003, p. 88. doi: [10.1145/958232.958236](https://doi.org/10.1145/958232.958236) (cit. on p. 2).
- [33] NodeJS. *Introduction to Node.js*. NodeJS. Accessed: 2022-09-30. url: <https://nodejs.dev/en/learn/introduction-to-nodejs/> (cit. on p. 22).
- [34] Stack Overflow. *Stack overflow developer survey 2022*. Stack Overflow. Accessed: 2022-09-30. url: <https://survey.stackoverflow.co/2022/#most-popular-technologies-language> (cit. on p. 21).
- [35] Matt Rae. *What is Adobe XD and what is it used for?* Adobe. Accessed: 2022-01-05. Oct. 2020. url: <https://www.adobe.com/products/xd/learn/get-started/what-is-adobe-xd-used-for.html> (cit. on p. 11).
- [36] Itesh Sharma. *What is Software Prototyping And Its Types?* TatvaSoft. Accessed: 2022-06-04. May 2022. url: <https://www.tatvasoft.com/outsourcing/2022/05/what-is-software-prototyping.html> (cit. on pp. 8, 9).

BIBLIOGRAPHY

- [37] Carlos Silva, Paolo Masci, Yi Zhang, Paul Jones, and José Campos. “A Use Error Taxonomy for Improving Human-Machine Interface Design in Medical Devices”. In: Apr. 2018 (cit. on p. 1).
- [38] Wikipedia. *Express.js*. Wikipedia. Accessed: 2022-09-30. Sept. 2022. url: <https://en.wikipedia.org/wiki/Express.js> (cit. on p. 22).

Listings

```
1 types
2   int = 0..1000
3   Page = {incrementer_page, decrementer_page}
4
5 interactor incrementer
6   attributes
7     label: Page
8     [value] display: int
9   actions
10    [click] redirect
11    [click] del
12    [click] increment
13  axioms
14    [] display=0
15    [increment] display'=display+1
16    [del] display'=0
17    per(increment) -> display<1001
18    per(del) -> display>0
19
20 interactor decrementer
21   attributes
22     label: Page
23     [value] display: int
24   actions
25    [click] redirect
26    [click] del
27    [click] decrement
28  axioms
29    [] display=1000
30    [decrement] display'=display-1
31    [del] display'=1000
32    per(decrement) -> display>0
```

```

33     per(del) -> display>1000
34
35 interactor main
36     aggregates
37         incrementer via incrementer
38         decrementer via decrementer
39     attributes
40         [page] page: Page
41     axioms
42         incrementer.label=incrementer_page
43         decrementer.label=decrementer_page
44         [incrementer.redirect] page'=decrementer_page
45         [decrementer.redirect] page'=incrementer_page

```

Listing I.1: Final structure

```

1  {
2  "decrementer::decrementer": {
3      "actions": [
4          {
5              "name": "process_del",
6              "type": "click"
7          },
8          {
9              "name": "redirect_page",
10             "type": "click"
11         },
12         {
13             "name": "process_decrement",
14             "type": "click"
15         }
16     ],
17     "attributes": [
18         {
19             "name": "display",
20             "type": "int",
21             "attribute": "value"
22         },
23         {
24             "name": "label",
25             "type": "Page",
26             "attribute": ""
27         }
28     ],
29     "axioms": {
30         "decrement": {

```

```

31     "display": "display - 1"
32 },
33 "del": {
34     "display": "1000"
35 },
36 "decrement::permission": "display > 0",
37 "del::permission": "display > 1000",
38 "inits": {
39     "display": "1000"
40 }
41 },
42 },
43 "incrementer::incrementer": {
44     "actions": [
45         {
46             "name": "process_del",
47             "type": "click"
48         },
49         {
50             "name": "redirect_page",
51             "type": "click"
52         },
53         {
54             "name": "process_increment",
55             "type": "click"
56         }
57     ],
58     "attributes": [
59         {
60             "name": "display",
61             "type": "int",
62             "attribute": "value"
63         },
64         {
65             "name": "label",
66             "type": "Page",
67             "attribute": ""
68         }
69     ],
70     "axioms": {
71         "increment": {
72             "display": "display + 1"
73         },
74         "del": {
75             "display": "0"

```

```

76     },
77     "increment::permission": "display < 1001",
78     "del::permission": "display > 0",
79     "inits": {
80         "display": "0"
81     }
82 }
83 },
84 "main": {
85     "actions": [],
86     "attributes": [
87         {
88             "name": "page",
89             "type": "Page",
90             "attribute": "page"
91         }
92     ],
93     "axioms": {
94         "incrementer.redirect": {
95             "page": "decrementer_page"
96         },
97         "decrementer.redirec": {
98             "page": "incrementer_page"
99         }
100     }
101 }
102 }

```

Listing I.2: Example of a JSON generated from the model from listing [I.1](#)

```

1 types
2   int = 0..1000
3   Page = {screen_off, screen_on, screen_2, screen_3, screen_4, screen_5, screen_6}
4   OperatorType = {none, sum, subtract, multiply, divide, exe}
5
6 interactor screen_off
7   attributes
8     [state] label: Page
9   actions
10    [click] redirect_power
11  axioms
12    label=screen_off
13
14 interactor screen_on
15   attributes
16    [state] label: Page

```

```

17     [value] display: int
18     [state] old_display: int
19     [state] oper: OperatorType
20 actions
21     [click] redirect_power
22     [click] redirect_menu1
23     [click] redirect_menu2
24     [click] redirect_menu3
25     [click] redirect_menu4
26     [click] redirect_menu5
27     [click] redirect_menu6
28     [click] process_plus
29     [click] process_subtract
30     [click] process_divide
31     [click] process_multiply
32     [click] process_exe
33     [click] process_del
34     [click] process_zero
35     [click] process_one
36     [click] process_two
37     [click] process_three
38     [click] process_four
39     [click] process_five
40     [click] process_six
41     [click] process_seven
42     [click] process_eight
43     [click] process_nine
44 axioms
45     label=screen_on
46     [] old_display=0
47     [] display=0
48     [process_plus] oper='sum & display'=0 & old_display'=display
49     [process_subtract] oper='subtract & display'=0 & old_display'=display
50     [process_divide] oper='divide & display'=0 & old_display'=display
51     [process_multiply] oper='multiply & display'=0 & old_display'=display
52     [process_zero] display'=display*10 & keep(old_display, oper)
53     [process_one] display'=display*10+1 & keep(old_display, oper)
54     [process_two] display'=display*10+2 & keep(old_display, oper)
55     [process_three] display'=display*10+3 & keep(old_display, oper)
56     [process_four] display'=display*10+4 & keep(old_display, oper)
57     [process_five] display'=display*10+5 & keep(old_display, oper)
58     [process_six] display'=display*10+6 & keep(old_display, oper)
59     [process_seven] display'=display*10+7 & keep(old_display, oper)
60     [process_eight] display'=display*10+8 & keep(old_display, oper)
61     [process_nine] display'=display*10+9 & keep(old_display, oper)

```

```

62     [process_del] display'=0 & keep(old_display, oper)
63     oper=sum -> [process_exe] display'=old_display+display & oper=none & keep(
        ↪ old_display)
64     oper=subtract -> [process_exe] display'=old_display-display & oper=none & keep(
        ↪ old_display)
65     oper=divide -> [process_exe] display'=old_display/display & oper=none & keep(
        ↪ old_display)
66     oper=multiply -> [process_exe] display'=old_display*display & oper=none & keep(
        ↪ old_display)
67     per(process_exe) -> oper != none
68
69 interactor screen_number
70     attributes
71         [state] label: Page
72         [value] display: int
73     actions
74         [click] redirect_power
75         [click] redirect_menu1
76         [click] redirect_menu2
77         [click] redirect_menu3
78         [click] redirect_menu4
79         [click] redirect_menu5
80         [click] redirect_menu6
81     axioms
82         label=screen_on
83
84 interactor main
85     aggregates
86         screen_off via screen_off
87         screen_on via screen_on
88         screen_number via screen_2
89         screen_number via screen_3
90         screen_number via screen_4
91         screen_number via screen_5
92         screen_number via screen_6
93     attributes
94         [page] page: Page
95     axioms
96         [screen_off.redirect_power] page'=screen_on
97         [screen_on.redirect_power] page'=screen_off
98         [screen_on.redirect_menu1] page'=screen_on
99         [screen_on.redirect_menu2] page'=screen_2
100        [screen_on.redirect_menu3] page'=screen_3
101        [screen_on.redirect_menu4] page'=screen_4
102        [screen_on.redirect_menu5] page'=screen_5

```

```

103 [screen_on.redirect_menu6] page'=screen_6
104
105 [screen_2.redirect_power] page'=screen_off
106 [screen_2.redirect_menu1] page'=screen_on
107 [screen_2.redirect_menu2] page'=screen_2
108 [screen_2.redirect_menu3] page'=screen_3
109 [screen_2.redirect_menu4] page'=screen_4
110 [screen_2.redirect_menu5] page'=screen_5
111 [screen_2.redirect_menu6] page'=screen_6
112
113 [screen_3.redirect_power] page'=screen_off
114 [screen_3.redirect_menu1] page'=screen_on
115 [screen_3.redirect_menu2] page'=screen_2
116 [screen_3.redirect_menu3] page'=screen_3
117 [screen_3.redirect_menu4] page'=screen_4
118 [screen_3.redirect_menu5] page'=screen_5
119 [screen_3.redirect_menu6] page'=screen_6
120
121 [screen_4.redirect_power] page'=screen_off
122 [screen_4.redirect_menu1] page'=screen_on
123 [screen_4.redirect_menu2] page'=screen_2
124 [screen_4.redirect_menu3] page'=screen_3
125 [screen_4.redirect_menu4] page'=screen_4
126 [screen_4.redirect_menu5] page'=screen_5
127 [screen_4.redirect_menu6] page'=screen_6
128
129 [screen_5.redirect_power] page'=screen_off
130 [screen_5.redirect_menu1] page'=screen_on
131 [screen_5.redirect_menu2] page'=screen_2
132 [screen_5.redirect_menu3] page'=screen_3
133 [screen_5.redirect_menu4] page'=screen_4
134 [screen_5.redirect_menu5] page'=screen_5
135 [screen_5.redirect_menu6] page'=screen_6
136
137 [screen_6.redirect_power] page'=screen_off
138 [screen_6.redirect_menu1] page'=screen_on
139 [screen_6.redirect_menu2] page'=screen_2
140 [screen_6.redirect_menu3] page'=screen_3
141 [screen_6.redirect_menu4] page'=screen_4
142 [screen_6.redirect_menu5] page'=screen_5
143 [screen_6.redirect_menu6] page'=screen_6

```

Listing I.3: Calculator Prototype's formal model

```

1 {
2 "screen_6::screen_number": {

```

```

3    "actions": [
4      { "name": "redirect_menu1", "type": "click" },
5      { "name": "redirect_menu6", "type": "click" },
6      { "name": "redirect_menu4", "type": "click" },
7      { "name": "redirect_menu5", "type": "click" },
8      { "name": "redirect_menu2", "type": "click" },
9      { "name": "redirect_power", "type": "click" },
10     { "name": "redirect_menu3", "type": "click" }
11   ],
12   "attributes": [
13     { "name": "display", "type": "int", "attribute": "value" },
14     { "name": "label", "type": "Page", "attribute": "state" }
15   ],
16   "axioms": {
17     "inits": {
18       "label": "screen_on"
19     }
20   }
21 },
22 "screen_5::screen_number": {
23   "actions": [
24     { "name": "redirect_menu1", "type": "click" },
25     { "name": "redirect_menu6", "type": "click" },
26     { "name": "redirect_menu4", "type": "click" },
27     { "name": "redirect_menu5", "type": "click" },
28     { "name": "redirect_menu2", "type": "click" },
29     { "name": "redirect_power", "type": "click" },
30     { "name": "redirect_menu3", "type": "click" }
31   ],
32   "attributes": [
33     { "name": "display", "type": "int", "attribute": "value" },
34     { "name": "label", "type": "Page", "attribute": "state" }
35   ],
36   "axioms": {
37     "inits": {
38       "label": "screen_on"
39     }
40   }
41 },
42 "screen_on::screen_on": {
43   "actions": [
44     { "name": "process_subtract", "type": "click" },
45     { "name": "redirect_menu1", "type": "click" },
46     { "name": "process_one", "type": "click" },
47     { "name": "process_divide", "type": "click" },

```

```

48     { "name": "process_plus", "type": "click" },
49     { "name": "process_six", "type": "click" },
50     { "name": "process_five", "type": "click" },
51     { "name": "process_zero", "type": "click" },
52     { "name": "process_two", "type": "click" },
53     { "name": "process_del", "type": "click" },
54     { "name": "process_three", "type": "click" },
55     { "name": "process_seven", "type": "click" },
56     { "name": "process_nine", "type": "click" },
57     { "name": "process_eight", "type": "click" },
58     { "name": "process_exe", "type": "click" },
59     { "name": "redirect_menu6", "type": "click" },
60     { "name": "redirect_menu4", "type": "click" },
61     { "name": "process_four", "type": "click" },
62     { "name": "redirect_menu5", "type": "click" },
63     { "name": "redirect_menu2", "type": "click" },
64     { "name": "process_multiply", "type": "click" },
65     { "name": "redirect_power", "type": "click" },
66     { "name": "redirect_menu3", "type": "click" }
67 ],
68 "attributes": [
69     { "name": "display", "type": "int", "attribute": "value" },
70     { "name": "oper", "type": "OperatorType", "attribute": "state" },
71     { "name": "label", "type": "Page", "attribute": "state" },
72     { "name": "old_display", "type": "int", "attribute": "state" }
73 ],
74 "axioms": {
75     "process_plus": {
76         "oper": "sum",
77         "display": "0",
78         "old_display": "display"
79     },
80     "process_subtract": {
81         "oper": "subtract",
82         "display": "0",
83         "old_display": "display"
84     },
85     "process_divide": {
86         "oper": "divide",
87         "display": "0",
88         "old_display": "display"
89     },
90     "process_multiply": {
91         "oper": "multiply",
92         "display": "0",

```

```
93     "old_display": "display"
94 },
95 "process_zero": {
96     "display": "display * 10",
97     "keep": ["old_display", "oper"]
98 },
99 "process_one": {
100     "display": "display * 10 + 1",
101     "keep": ["old_display", "oper"]
102 },
103 "process_two": {
104     "display": "display * 10 + 2",
105     "keep": ["old_display", "oper"]
106 },
107 "process_three": {
108     "display": "display * 10 + 3",
109     "keep": ["old_display", "oper"]
110 },
111 "process_four": {
112     "display": "display * 10 + 4",
113     "keep": ["old_display", "oper"]
114 },
115 "process_five": {
116     "display": "display * 10 + 5",
117     "keep": ["old_display", "oper"]
118 },
119 "process_six": {
120     "display": "display * 10 + 6",
121     "keep": ["old_display", "oper"]
122 },
123 "process_seven": {
124     "display": "display * 10 + 7",
125     "keep": ["old_display", "oper"]
126 },
127 "process_eight": {
128     "display": "display * 10 + 8",
129     "keep": ["old_display", "oper"]
130 },
131 "process_nine": {
132     "display": "display * 10 + 9",
133     "keep": ["old_display", "oper"]
134 },
135 "process_del": {
136     "display": "0",
137     "keep": ["old_display", "oper"]
```

```

138     },
139     "process_exe::oper = sum": {
140         "display": "old_display + display",
141         "oper": "none",
142         "keep": ["old_display"]
143     },
144     "process_exe::oper = subtract": {
145         "display": "old_display - display",
146         "oper": "none",
147         "keep": ["old_display"]
148     },
149     "process_exe::oper = divide": {
150         "display": "old_display / display",
151         "oper": "none",
152         "keep": ["old_display"]
153     },
154     "process_exe::oper = multiply": {
155         "display": "old_display * display",
156         "oper": "none",
157         "keep": ["old_display"]
158     },
159     "process_exe::permission": "oper != none",
160     "inits": {
161         "label": "screen_on",
162         "old_display": "0",
163         "display": "0"
164     }
165 },
166 },
167 "screen_4::screen_number": {
168     "actions": [
169         { "name": "redirect_menu1", "type": "click" },
170         { "name": "redirect_menu6", "type": "click" },
171         { "name": "redirect_menu4", "type": "click" },
172         { "name": "redirect_menu5", "type": "click" },
173         { "name": "redirect_menu2", "type": "click" },
174         { "name": "redirect_power", "type": "click" },
175         { "name": "redirect_menu3", "type": "click" }
176     ],
177     "attributes": [
178         { "name": "display", "type": "int", "attribute": "value" },
179         { "name": "label", "type": "Page", "attribute": "state" }
180     ],
181     "axioms": {
182         "inits": {

```

```

183     "label": "screen_on"
184   }
185 }
186 },
187 "screen_3::screen_number": {
188   "actions": [
189     { "name": "redirect_menu1", "type": "click" },
190     { "name": "redirect_menu6", "type": "click" },
191     { "name": "redirect_menu4", "type": "click" },
192     { "name": "redirect_menu5", "type": "click" },
193     { "name": "redirect_menu2", "type": "click" },
194     { "name": "redirect_power", "type": "click" },
195     { "name": "redirect_menu3", "type": "click" }
196   ],
197   "attributes": [
198     { "name": "display", "type": "int", "attribute": "value" },
199     { "name": "label", "type": "Page", "attribute": "state" }
200   ],
201   "axioms": {
202     "inits": {
203       "label": "screen_on"
204     }
205   }
206 },
207 "screen_off::screen_off": {
208   "actions": [{ "name": "redirect_power", "type": "click" }],
209   "attributes": [{ "name": "label", "type": "Page", "attribute": "state" }],
210   "axioms": {
211     "inits": {
212       "label": "screen_off"
213     }
214   }
215 },
216 "screen_2::screen_number": {
217   "actions": [
218     { "name": "redirect_menu1", "type": "click" },
219     { "name": "redirect_menu6", "type": "click" },
220     { "name": "redirect_menu4", "type": "click" },
221     { "name": "redirect_menu5", "type": "click" },
222     { "name": "redirect_menu2", "type": "click" },
223     { "name": "redirect_power", "type": "click" },
224     { "name": "redirect_menu3", "type": "click" }
225   ],
226   "attributes": [
227     { "name": "display", "type": "int", "attribute": "value" },

```

```

228     { "name": "label", "type": "Page", "attribute": "state" }
229 ],
230 "axioms": {
231     "inits": {
232         "label": "screen_on"
233     }
234 },
235 },
236 "main": {
237     "actions": [],
238     "attributes": [{ "name": "page", "type": "Page", "attribute": "page" }],
239     "axioms": {
240         "screen_off.redirect_power": {
241             "page": "screen_on"
242         },
243         "screen_on.redirect_power": {
244             "page": "screen_off"
245         },
246         "screen_on.redirect_menu1": {
247             "page": "screen_on"
248         },
249         "screen_on.redirect_menu2": {
250             "page": "screen_2"
251         },
252         "screen_on.redirect_menu3": {
253             "page": "screen_3"
254         },
255         "screen_on.redirect_menu4": {
256             "page": "screen_4"
257         },
258         "screen_on.redirect_menu5": {
259             "page": "screen_5"
260         },
261         "screen_on.redirect_menu6": {
262             "page": "screen_6"
263         },
264         "screen_2.redirect_power": {
265             "page": "screen_off"
266         },
267         "screen_2.redirect_menu1": {
268             "page": "screen_on"
269         },
270         "screen_2.redirect_menu2": {
271             "page": "screen_2"
272         },

```

```
273     "screen_2.redirect_menu3": {
274         "page": "screen_3"
275     },
276     "screen_2.redirect_menu4": {
277         "page": "screen_4"
278     },
279     "screen_2.redirect_menu5": {
280         "page": "screen_5"
281     },
282     "screen_2.redirect_menu6": {
283         "page": "screen_6"
284     },
285     "screen_3.redirect_power": {
286         "page": "screen_off"
287     },
288     "screen_3.redirect_menu1": {
289         "page": "screen_on"
290     },
291     "screen_3.redirect_menu2": {
292         "page": "screen_2"
293     },
294     "screen_3.redirect_menu3": {
295         "page": "screen_3"
296     },
297     "screen_3.redirect_menu4": {
298         "page": "screen_4"
299     },
300     "screen_3.redirect_menu5": {
301         "page": "screen_5"
302     },
303     "screen_3.redirect_menu6": {
304         "page": "screen_6"
305     },
306     "screen_4.redirect_power": {
307         "page": "screen_off"
308     },
309     "screen_4.redirect_menu1": {
310         "page": "screen_on"
311     },
312     "screen_4.redirect_menu2": {
313         "page": "screen_2"
314     },
315     "screen_4.redirect_menu3": {
316         "page": "screen_3"
317     },
```

```
318     "screen_4.redirect_menu4": {
319         "page": "screen_4"
320     },
321     "screen_4.redirect_menu5": {
322         "page": "screen_5"
323     },
324     "screen_4.redirect_menu6": {
325         "page": "screen_6"
326     },
327     "screen_5.redirect_power": {
328         "page": "screen_off"
329     },
330     "screen_5.redirect_menu1": {
331         "page": "screen_on"
332     },
333     "screen_5.redirect_menu2": {
334         "page": "screen_2"
335     },
336     "screen_5.redirect_menu3": {
337         "page": "screen_3"
338     },
339     "screen_5.redirect_menu4": {
340         "page": "screen_4"
341     },
342     "screen_5.redirect_menu5": {
343         "page": "screen_5"
344     },
345     "screen_5.redirect_menu6": {
346         "page": "screen_6"
347     },
348     "screen_6.redirect_power": {
349         "page": "screen_off"
350     },
351     "screen_6.redirect_menu1": {
352         "page": "screen_on"
353     },
354     "screen_6.redirect_menu2": {
355         "page": "screen_2"
356     },
357     "screen_6.redirect_menu3": {
358         "page": "screen_3"
359     },
360     "screen_6.redirect_menu4": {
361         "page": "screen_4"
362     },
```

```

363     "screen_6.redirect_menu5": {
364         "page": "screen_5"
365     },
366     "screen_6.redirect_menu6": {
367         "page": "screen_6"
368     }
369 }
370 }
371 }

```

Listing I.4: Calculator Prototype's generated JSON

```

1  var label = "screen_on";
2
3  var oper = undefined;
4  var old_display = undefined;
5
6  function reloadInit() {
7      !(oper != "none")
8      ? (document.getElementById("equal").disabled = true)
9      : (document.getElementById("equal").disabled = false);
10 }
11
12 reloadInit();
13
14 document
15     .getElementById("menu1")
16     .addEventListener("click", () => (window.location.href = "/screen_on"));
17
18 document.getElementById("one").addEventListener("click", () => {
19     id = "display";
20     display = parseInt(document.getElementById(id).textContent);
21
22     document.getElementById(id).textContent = display * 10 + 1;
23     reloadInit();
24 });
25
26 document.getElementById("six").addEventListener("click", () => {
27     id = "display";
28     display = parseInt(document.getElementById(id).textContent);
29
30     document.getElementById(id).textContent = display * 10 + 6;
31     reloadInit();
32 });
33
34 document.getElementById("subtract").addEventListener("click", () => {

```

```

35     oper = "subtract";
36     old_display = parseFloat(document.getElementById("display").textContent);
37     document.getElementById("display").textContent = 0;
38 });
39
40 document.getElementById("five").addEventListener("click", () => {
41     id = "display";
42     display = parseInt(document.getElementById(id).textContent);
43
44     document.getElementById(id).textContent = display * 10 + 5;
45     reloadInit();
46 });
47
48 document.getElementById("zero").addEventListener("click", () => {
49     id = "display";
50     display = parseInt(document.getElementById(id).textContent);
51
52     document.getElementById(id).textContent = display * 10;
53     reloadInit();
54 });
55
56 document.getElementById("two").addEventListener("click", () => {
57     id = "display";
58     display = parseInt(document.getElementById(id).textContent);
59
60     document.getElementById(id).textContent = display * 10 + 2;
61     reloadInit();
62 });
63
64 document.getElementById("three").addEventListener("click", () => {
65     id = "display";
66     display = parseInt(document.getElementById(id).textContent);
67
68     document.getElementById(id).textContent = display * 10 + 3;
69     reloadInit();
70 });
71
72 document.getElementById("plus").addEventListener("click", () => {
73     oper = "sum";
74     old_display = parseFloat(document.getElementById("display").textContent);
75     document.getElementById("display").textContent = 0;
76 });
77
78 document.getElementById("seven").addEventListener("click", () => {
79     id = "display";

```

```
80   display = parseInt(document.getElementById(id).textContent);
81
82   document.getElementById(id).textContent = display * 10 + 7;
83   reloadInit();
84 });
85
86 document.getElementById("nine").addEventListener("click", () => {
87   id = "display";
88   display = parseInt(document.getElementById(id).textContent);
89
90   document.getElementById(id).textContent = display * 10 + 9;
91   reloadInit();
92 });
93
94 document.getElementById("eight").addEventListener("click", () => {
95   id = "display";
96   display = parseInt(document.getElementById(id).textContent);
97
98   document.getElementById(id).textContent = display * 10 + 8;
99   reloadInit();
100 });
101
102 document.getElementById("multiply").addEventListener("click", () => {
103   oper = "multiply";
104   old_display = parseFloat(document.getElementById("display").textContent);
105
106   document.getElementById("display").textContent = 0;
107 });
108
109 document.getElementById("equal").addEventListener("click", () => {
110   id = "display";
111   display = parseInt(document.getElementById(id).textContent);
112
113   if (oper !== "none") {
114     if (oper === "sum") {
115       document.getElementById(id).textContent = old_display + display;
116       oper = "none";
117       reloadInit();
118     }
119   }
120 });
121
122 document.getElementById("equal").addEventListener("click", () => {
123   id = "display";
124   display = parseInt(document.getElementById(id).textContent);
```

```

125
126     if (oper !== "none") {
127         if (oper === "subtract") {
128             document.getElementById(id).textContent = old_display - display;
129             oper = "none";
130             reloadInit();
131         }
132     }
133 });
134
135 document.getElementById("equal").addEventListener("click", () => {
136     id = "display";
137     display = parseInt(document.getElementById(id).textContent);
138
139     if (oper !== "none") {
140         if (oper === "divide") {
141             document.getElementById(id).textContent = old_display / display;
142             oper = "none";
143             reloadInit();
144         }
145     }
146 });
147
148 document.getElementById("equal").addEventListener("click", () => {
149     id = "display";
150     display = parseInt(document.getElementById(id).textContent);
151
152     if (oper !== "none") {
153         if (oper === "multiply") {
154             document.getElementById(id).textContent = old_display * display;
155             oper = "none";
156             reloadInit();
157         }
158     }
159 });
160
161 document.getElementById("divide").addEventListener("click", () => {
162     oper = "divide";
163     old_display = parseFloat(document.getElementById("display").textContent);
164     document.getElementById("display").textContent = 0;
165 });
166
167 document
168     .getElementById("menu6")
169     .addEventListener("click", () => (window.location.href = "/screen_6"));

```

```

170
171 document.getElementById("del").addEventListener("click", () => {
172     id = "display";
173     display = parseInt(document.getElementById(id).textContent);
174
175     document.getElementById(id).textContent = 0;
176     reloadInit();
177 });
178
179 document
180     .getElementById("menu4")
181     .addEventListener("click", () => (window.location.href = "/screen_4"));
182
183 document.getElementById("four").addEventListener("click", () => {
184     id = "display";
185     display = parseInt(document.getElementById(id).textContent);
186
187     document.getElementById(id).textContent = display * 10 + 4;
188     reloadInit();
189 });
190
191 document
192     .getElementById("menu5")
193     .addEventListener("click", () => (window.location.href = "/screen_5"));
194
195 document
196     .getElementById("menu2")
197     .addEventListener("click", () => (window.location.href = "/screen_2"));
198
199 document
200     .getElementById("power")
201     .addEventListener("click", () => (window.location.href = "/screen_off"));
202
203 document
204     .getElementById("menu3")
205     .addEventListener("click", () => (window.location.href = "/screen_3"));

```

Listing I.5: Calculator Prototype's "screen_on" generated JavaScript

```

1 var label = "screen_off"
2
3 function reloadInit() {
4 }
5 reloadInit()
6 document.getElementById('power').addEventListener('click', () => window.location.href
    ↪ = "/screen_on")

```

Listing I.6: Calculator Prototype’s “screen_off” generated JavaScript

```
1 var label = "screen_on"
2
3 function reloadInit() {
4 }
5 reloadInit()
6 document.getElementById('menu1').addEventListener('click', () => window.location.href
   ↳ = "/screen_on")
7 document.getElementById('menu6').addEventListener('click', () => window.location.href
   ↳ = "/screen_6")
8 document.getElementById('menu4').addEventListener('click', () => window.location.href
   ↳ = "/screen_4")
9 document.getElementById('menu5').addEventListener('click', () => window.location.href
   ↳ = "/screen_5")
10 document.getElementById('menu2').addEventListener('click', () => window.location.href
   ↳ = "/screen_2")
11 document.getElementById('power').addEventListener('click', () => window.location.href
   ↳ = "/screen_off")
12 document.getElementById('menu3').addEventListener('click', () => window.location.href
   ↳ = "/screen_3")
```

Listing I.7: Calculator Prototype’s “screen_number” generated JavaScript

