Research Commons

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Design Patterns:

Infrastructure and Examples

A thesis

submitted in partial fulfillment

of the requirements for the degree

of

Master of Science in Computer Science

by

Scott Crickett

supervised by

Steve Reeves

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2017

Abstract

Modern interactive systems can be incredibly complex, with a variety of screens, menus, widgets, etc. available to the user. Due to this, modelling these interactive systems can also be incredibly complex and while there are techniques to help overcome this, it can often lead to radically different models depending on the modeller.

This thesis explores the use of two design patterns created in order to help simplify modelling interactive systems. In the process of doing this, we first explore μ-Charts and its semantics, which are defined in Z, in order to understand its capabilities. We then discover a feature we name the Return feature, which we break down into two parts, Return Home and Return Back, and create patterns in order to concisely model them. Finally, we test these patterns and evaluate the tools we used to create the μ-charts, translate them into their Z semantics and test them.

Acknowledgements

First, I want to thank my supervisor, Professor Steve Reeves. His encouraging guidance, accurate proof-reading, helpful pointers and seemingly endless patience have been instrumental to the completion of my research and this thesis and it is all greatly appreciated.

I would like to thank Colin Pilbrow for his help with Z, during our mutual research into μ-Charts semantics and as a sounding board when I needed some quick feedback. His helpfulness cannot be understated.

I would also like to thank Sapna Jaidka for her assistance at numerous points, as well as Dr. Judy Bowen for her help at different points, particularly in regards to AMuZed and ZooM.

Finally, I would like to thank my my parents for being hugely supportive during my research, as well as my little niece and nephew, whose enthusiasm and energy is as infectious as it is exhausting.

Contents

1 Introduction **1**

2 Background **3**

2.1 Z . . . 3

2.2 μ-Charts . . . 6

2.2.1 Sequential charts . . . 6

2.2.2 Decomposition operator . . . 15

2.2.3 Feedback operator . . . 19

2.2.4 Value carrying signals . . . 22

2.2.5 Local variables . . . 24

2.3 Tools . . . 27

2.3.1 AMuZed and ZooM . . . 28

2.3.2 ProZ . . . 29

2.4 Design Patterns . . . 31

2.4.1 Callback pattern . . . 31

2.4.2 Binary Choice pattern . . . 32

3 Design Patterns **34**

3.1 Discovering Design Patterns . . . 34

3.2 Return Home pattern . . . 46

3.3 Return pattern . . . 51

4 Testing **62**

4.1 ZooM Modifications 62

4.2 μ-Charts Semantics Modifications 64

4.2.1 Translating decomposition re-initialisation 64

4.2.2 AMuZed and ZooM bugs 69

4.3 ProZ 70

4.3.1 ProZ formatting issues 71

4.3.2 ProZ memory issues 72

4.3.3 Testing the patterns 73

5 Conclusion and Future Work **74**

5.1 Overview of Project Goals 74

5.2 Summary of Results 74

5.3 Future Work 75

Appendix A Initial Return Home Pattern Semantics **79**

Appendix B Final Return Home Pattern Semantics **85**

Appendix C Return Pattern Semantics **91**

List of Figures

2.1 Simple sequential μ-chart . . . 7
2.2 μ-chart with more complex transitions . . . 14
2.3 Decomposed μ-chart . . . 16
2.4 μ-chart with feedback . . . 20
2.5 μ-chart with value carrying signals . . . 22
2.6 μ-chart with local variable . . . 24
2.7 AMuZed user interface . . . 29
2.8 ProZ user interface . . . 30
2.9 Callback pattern . . . 31
2.10 Binary Choice pattern . . . 32

3.1 PS Dynamic Menu user interface . . . 35
3.2 PS Dynamic Menu chart with states representing duplicate branches 37
3.3 PS Dynamic Menu chart with local variable storing previous state 39
3.4 PS Dynamic Menu chart with local variable and duplicate states . 41
3.5 PS Dynamic Menu chart with multiple local variables . . . 45
3.6 Party segment from PS Dynamic Menu chart with Return Home feature . . . 47
3.7 Simple Return Home pattern . . . 48
3.8 Full PS Dynamic Menu chart with Return Home function added . 49
3.9 Return Home pattern applied to Party segment from PS Dynamic Menu chart . . . 50

3.10 Return Home pattern applied to full PS Dynamic Menu chart . . 52
3.11 Party section from PS Dynamic Menu chart with both Return features 54
3.12 Simple Return pattern 55
3.13 Revised Return pattern using feedback and value carrying signals 57
3.14 Final Return Home pattern 58
3.15 Final Return pattern 59
3.16 Final Return pattern applied to Party segment of PS Dynamic Menu chart 60

Chapter 1

Introduction

Model checking is a branch of formal methods that aims to formally verify and validate finite-state systems, by creating a formal model to represent a system and then, if possible, exhaustively checking whether this model meets a set of requirements, which are specified as properties. These models can be represented textually, using textual specification languages such as Z and B, as well as graphically, using graphical specification languages such as StateCharts [4] and μ-Charts [9] [12].

Graphical specification languages were initially introduced in the form of StateCharts by Harel. Harel argued that reactive systems require a different approach to their specification than textual languages and that visual languages and methodologies are the best way to specify them. This marked the beginning of research into graphical specification languages, which led to the creation of other similar languages such as Mini-Statecharts [7] and μ-Charts.

The levels of complexity within modern user interfaces means that creating models at a suitable level of abstraction that are reasonably sized, readable, and therefore usable, is very challenging. In response to this issue, the idea of using design patterns to simplify and make models more readable was offered [2].

Software design patterns are reusable methods that can be applied to com-

monly occurring software design problems [5]. They have been used within software design since 1987 but it was not until Gamma, Helm, Johnson and Vlissides published the book *Design Patterns: Elements of Reusable Object-Oriented Software* [5] that the idea of applying patterns to computer programming started to gain popularity. Their use within interface design analysis and verification, however, has been limited.

The use of design patterns to model interactive systems is an area Bowen and Reeves explored in 2015 [2]. Their goal was to make certain patterns explicit in the hope that a body of known patterns could then be developed in a similar manner as programming patterns. They introduced two patterns, the Callback pattern and the Binary Choice pattern, while formalising them and using them within examples.

In this research project we aim to continue the work of Reeves and Bowen to explore design patterns within graphical models in an attempt to more elegantly model complex systems. Using the μ-Charts semantics, we then intend to investigate and prove properties within these design patterns by translating them into the Z specification language. We will also evaluate the tools, AMuZed, Zoom and ProZ, used within this process.

The rest of this thesis is structured as follows: in chapter two we discuss the basics of the specification languages Z and μ-Charts, in order to clarify aspects of these languages that will be used in later chapters; in chapter three we discuss the Return Home and Return Back design patterns, as well as the interpretations of μ-Charts made to model them; in chapter four we discuss the testing of the design patterns, as well as the tools that have been used as part of the testing process; and finally, we conclude in chapter five.

Chapter 2

Background

The goal of this chapter is to provide the background knowledge that is required to understand the content within the subsequent chapters. It provides overviews of the specification languages Z and μ-Charts, as well as the tools AMuZed, ZooM and ProZ. It also discusses the prior work on using design patterns to model interactive systems.

2.1 Z

Z is a formal specification language based upon set theory and first order predicate logic [3] widely used within the formal methods community to describe and model computing systems. Specification languages differ from programming languages in that they are used to describe systems, not produce executable code. A typical Z-modelled system will consist of a state space, which represents the state of the system, and operations that change said state. Operations are represented using schemas, which consist of declarations and predicates. Z uses syntactic sugar in order to allow the reader to focus more on the specification than the logic. As an example of this, there is an implicit logical AND that combines multiple predicate lines. The Z specification of a simple counter system is shown below to

demonstrate.

$$
\begin{array}{|l}
\hline
\textit{Counter} \\
count, max : \mathbb{N} \\
\hline
count \leq max \\
\hline
\end{array}
$$

The *Counter* schema shown above shows the state space of the counter. First it declares the observations, which is a name we use for the parts of the state that can be observed, *count* and *max* as natural numbers. The predicate section follows, which constrains the observation *count* to be less than or equal to the maximum *max* within this state space.

$$
\begin{array}{|l}
\hline
\textit{InitCounter} \\
Counter' \\
\hline
count' = 0 \\
max' = 100 \\
\hline
\end{array}
$$

The *InitCounter* operation schema is shown above. This schema first declares that it involves both the declaration and predicate parts of the primed *Counter′* state space and then initialises the primed observations $count'$ and max'. This priming convention is used to denote the next state of the system, so the *Counter′* space represents the next state of the *Counter* state space and the max' observation represents the next state of the *max* observation. As mentioned above, there is an implicit logical AND combining both initialisations, making it equivalent to $count' = 0 \wedge max' = 100$. If these values did not satisfy the predicate within the state space, i.e. $count \leq max$, the counter system would not be able to be initialised.

$$
\begin{array}{l}
\textit{Increment} \\
\hline
\textit{Counter} \\
\textit{Counter}' \\
\hline
count' = value + 1
\end{array}
$$

The above schema shows the *Increment* operation schema. It declares that the schema uses the *Counter* state space, as well as the primed *Counter'* state space. The primed $count'$ observation is defined as the sum of the current $count$ incremented by one every time the increment operation occurs, so long as the predicate within the *Counter* and *Counter'* state space is true.

Note that within the *Increment* schema, an error has been made by leaving the primed max' observation without a value. Within Z, if an observation within the state space has been left unconstrained without a value, then it may potentially be any value within its definition so long as it satisfies the predicate. Within the *Increment* schema, this means that the max' observation may change to any natural number so long as the predicate $count' \leq max'$ is satisfied. This can be used deliberately if there is an unknown factor within a system but in this case, the error can be fixed by adding the line $max' = max$ to the *Increment* schema's predicate as can be seen below.

$$
\begin{array}{l}
\textit{Increment} \\
\hline
\textit{Counter} \\
\textit{Counter}' \\
\hline
count' = value + 1 \\
max' = max
\end{array}
$$

2.2 μ-Charts

μ-Charts are a state diagram variant [13] with semantics given in Z [12] [14], involving states and transitions, and created with the assumption that sets of input signals will appear from the environment from time to time. One of the states is called the starting state and denotes the state the system begins in. Transitions occur only when sets of inputs appear and all resulting actions occur instantaneously alongside them. It is labelled with a pair, the guard and an action, written in the form $guard/action$.

The guard is a conditional expression, which usually involves input signals, but can also include local variables and feedback. If the guard is left empty, we consider this syntactic sugar for a $TRUE$ value, which means that it may occur, whatever the input signals are.

An action can consist of sending an output signal, changing a local variable or both. μ-Charts also include a number of other features that set them apart, such as composed and decomposed charts, as well as the aforementioned local variables, value carrying signals and feedback. Like the guard, the action can be left empty, which we consider syntactic sugar for no action occurring other than the transition to the new state. If the action is empty, we also do not include the slash.

This section will focus on sequential charts, decomposed charts, local variables, feedback and value carrying signals, as they are the features that are used within this paper. For more information on composed charts, see [12].

2.2.1 Sequential charts

Figure 2.1 shows a simple sequential μ-chart. It contains two states, the start state X and the state Y, and a transition from state X to state Y. The transition is labelled a/b. It transitions from state X to state Y when the input signal a is

received, while outputting the signal b.

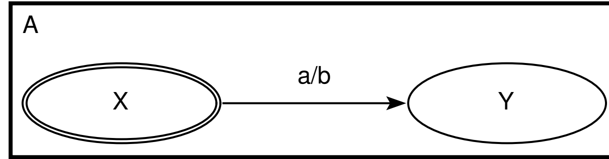


Figure 2.1: Simple sequential μ-chart

The semantics of μ-Charts consists of generic schemas and definitions that give the semantics of any μ-chart in Z. These definitions are used by machine translation tools, such as ZooM. The tool ZooM will be explained below but it is a tool that translates μ-Charts into the Z semantics. When it does this it creates numerous schemas to describe the chart, as well as higher level system schemas which are used to receive input and output from the environment. It is usually simple to recognise which part of the system a schema is associated with, as the chart name will be included within the schema name in subscript.

$$\mu_{State} ::= A \mid AX \mid AY$$

$$Signal ::= Sb \mid Sa$$

Seen above is the first detail of the semantics: the system type definitions. The first, μ_{State}, is a type which is defined, using the ::= symbol, to consist of all charts and states within the system. The states have the name of the chart they are part of prefixed to their state names. For example, the state Y appears as state AY because it is part of the chart A. In this case, the μ_{State} type consists of the chart A, as well as the states AX and AY. The second type, *Signal* is a set of the input and output signals used within the chart and, as they are signals, are denoted by a capital S before the signal name. In this case, the *Signal* type consists of the signals Sb and Sa.

$states_A : \mathbb{P}\, \mu_{State}$

$inputI_A : \mathbb{P}\, Signal$

$outputI_A : \mathbb{P}\, Signal$

$states_A = \{AX, AY\}$

$inputI_A = \{Sa\}$

$outputI_A = \{Sb\}$

Above is the first schema of chart A, showing chart A's global constants. These global constants are used throughout the system and are used as types. $states_A$ is a set of the states within chart A. It is has a power set type, denoted by the symbol $\mathbb{P}$, of the μ_{State} type previously defined. A power set is a set of all possible subsets of a type, as well as the set itself and including an empty set. For example, the power set of the μ_{State} type would consist of the subsets $\{\}$, $\{A\}$, $\{AX\}$, $\{AY\}$, $\{A, AX\}$, $\{A, AY\}$, $\{AX, AY\}$ and $\{A, AX, AY\}$. $states_A$ is then constrained to the $\{AX, AY\}$ subset, as these are the two states that are part of chart A.

$inputI_A$ and $outputI_A$ are similar. $inputI_A$ is a set of the input signals within chart A, while $outputI_A$ is a set of the output signals within chart A. They are each declared as having power set types of the $Signal$ type and are then constrained. $inputI_A$ is constrained to the signal Sa, as that is the only input signal received by chart A. $outputI_A$ is constrained to the Sb signal, as it is the output signal that can be sent by chart A.

$Chart_A$

$c_A : states_A$

The schema above shows the state space of chart A. It has a single observation, c_A, which represents the current state of chart A.

$Init_A$

$Chart_A$

$c_A = AX$

The $Init_A$ operation schema is shown next, which specifies how chart A is initialised. It declares that it includes the $Chart_A$ state space and then initialises the current state of chart A to AX.

AX

$Chart_A$

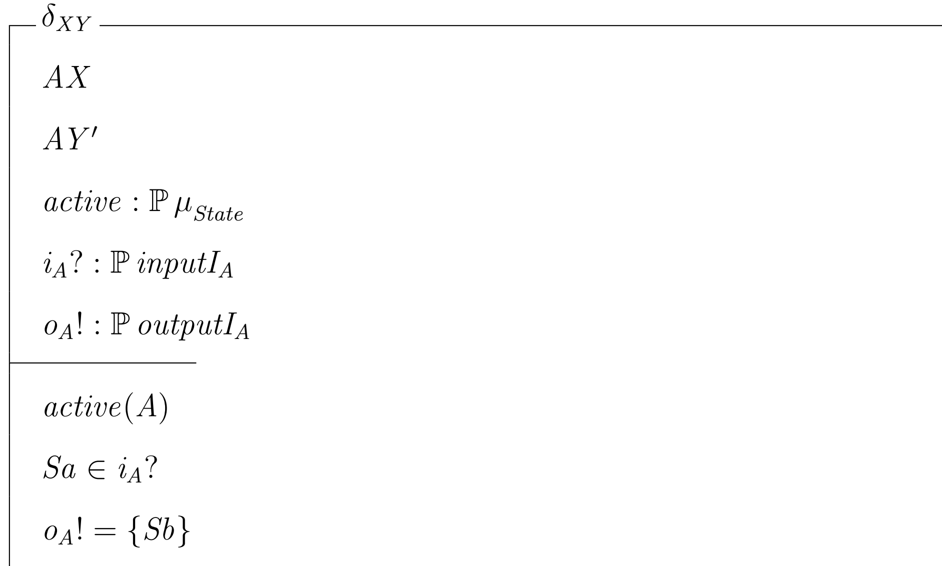
$c_A = AX$

AY

$Chart_A$

$c_A = AY$

The two schemas shown above are schemas representing the two states within chart A, AX and AY. Both declare that their respective states are the current state. The transition schemas, declared later, include these schemas. If a transition schema includes schema AX, the start state of that transition is state AX.



Above we see δ_{XY} operation schema, which is the schema for the transition labelled a/b within the example. The system transitions from state AX to state AY when the input signal Sa is received, outputting the signal Sb in the process. The lower case delta symbol, δ, is used to denote some operation schemas, in this case the transition from state AX to state AY. This symbol is then followed by $_{XY}$ because the schema transitions from state AX to state AY.

Within the declarations, we see that the AX schema is included. This means that the start state of this transition is the state AX. The primed AY' schema means that the next state of the chart after the transition occurs is state AY.

The next declaration is the observation *active*, which has a power set type of the μ_{State} type. Combined with the predicate, $active(A)$, this observation means that chart A is the active chart. This observation is used within the top level of the $ASys$ schema and we will detail this further within that section.

The last two declarations are the input signals, $i_A?$, and output signals, $o_A!$. The input observation will either be the signal Sa or there will be no signal. The guard of δ_{XY} within the example μ-chart was a, due to this the predicate includes the line $Sa \in i_A?$, which means that for transition δ_{XY} to occur, the input signal Sa must be received. The output has a power set type of $outputI_A$, which means

that we will either output no signals or the signal Sb. The action of δ_{XY} within the example μ-chart was b, so the predicate includes the line $o_A! = \{Sb\}$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\epsilon_A} \\
\hline
\Delta Chart_A \\
active : \mathbb{P}\, \mu_{State} \\
i_A? : \mathbb{P}\, inputI_A \\
o_A! : \mathbb{P}\, outputI_A \\
\hline
active(A) \\
c'_A = c_A \\
\neg\, (AX \land Sa \in i_A?) \\
o_A! = \{\} \\
\hline
\end{array}
$$

The above schema is the ϵ_A (epsilon) operation schema for chart A. The epsilon operation is intended to account for any steps within the chart when no transition occurs. As such, this schema represents the chart being idle, where it is active but no transitions occur.

The first declaration of the epsilon schema is $\Delta Chart_A$. The upper case delta, Δ, is used to denote a state change to $Chart_A$. If this is used, we know that there may be changes to the observations within the state space. In this case, we can see the predicate $c'_A = c_A$ below. This means that the current state of chart A, c_A, will remain the same within the next state of the system.

The next declaration is *active*, which is declared and constrained exactly as it was in the δ_{XY} operation schema.

After this, the input signals, $i_A?$, and output signals, $o_A!$, are declared as having power set types of type $inputI_A$ and $outputI_A$, respectively. Within the predicate, we see that the output signals are constrained to an empty set, so there cannot be any output signals being sent.

This schema may occur when no other transitions within the system may

occur. There is only one transition within this example μ-chart and that is the δ_{XY} transition. In order to ensure the epsilon operation schema occurs when that transition will not occur, we include the line $\neg\ (AX\ \wedge\ Sa\ \in\ i_A?)$ within the predicate. This means that the epsilon schema will only occur when δ_{XY} will not.

$$
\begin{array}{|l}
\hline
\textit{Iactive}_A \\
\Xi \textit{Chart}_A \\
\textit{active} : \mathbb{P}\, \mu_{State} \\
o_A! : \mathbb{P}\, \textit{outputI}_A \\
\hline
\neg\ \textit{active}(A) \\
o_A! = \{\} \\
\hline
\end{array}
$$

Above we see chart A's inactive operation schema, named $\textit{Iactive}_A$. This schema is intended to account for when the chart containing this transition is inactive.

The first declaration of the inactive schema is $\Xi \textit{Chart}_A$. The upper case Xi, Ξ, is used to denote when there are no changes being made to the state. As such, this means that the inactive schema makes no changes to the observations within the $\textit{Chart}_A$ state space and the current state of the chart will remain the same when the inactive schema occurs.

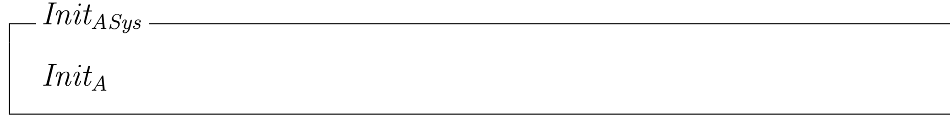
Following this, *active* is once again declared having a power set type of the μ_{State} type. Within the predicate, $\textit{active}(A)$ is negated. As the name of the inactive schema implies, this is because this operation schema will only occur when chart A is inactive and this predicate ensures that.

Finally, like the epsilon schema, the inactive schema declares and constrains the output signals, ensuring that it does not send any. It is worth noting that there are no input signals declared or constrained. This is because they are irrelevant to chart A while it is inactive.

$$\delta_A \mathrel{\widehat{=}} \delta_{XY} \vee \textit{Iactive}_A \vee \epsilon_A$$

The predicate above is the final chart A schema. It shows the main chart A operation schema, δ_A. This schema is intended to be the main operation schema for a chart, with every other operation schema, such as the transitions, epsilon and inactive schemas, available as possibilities within it. As such, it is defined as the disjunction of all of these operation schemas, in this case the δ_{XY}, $Iactive_A$ and ϵ_A.

Following the chart A schemas, we have the top level system schemas. In this case, the schemas are named $ASys$, with the suffix Sys added to the chart name. These are used to receive input and send output every time sets of inputs are received from the environment. It also ensures that the main chart is active.



The first $ASys$ schema is the $Init_{ASys}$ initialisation schema. This schema is used to initialise all charts within the system, which in this case is simply chart A, so it includes the $Init_A$ schema within it.

The second and final $ASys$ schema is the top level operation schema named simply $ASys$, which is the main operation schema of the entire system. Similar to the way the δ_A operation schema gathers the various operation schemas within its respective chart, this schema gathers all main operation schemas from the level below it. In this case, it is only δ_A.

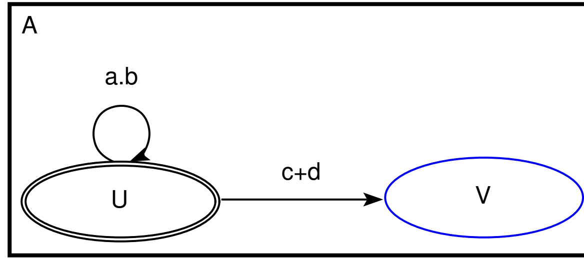
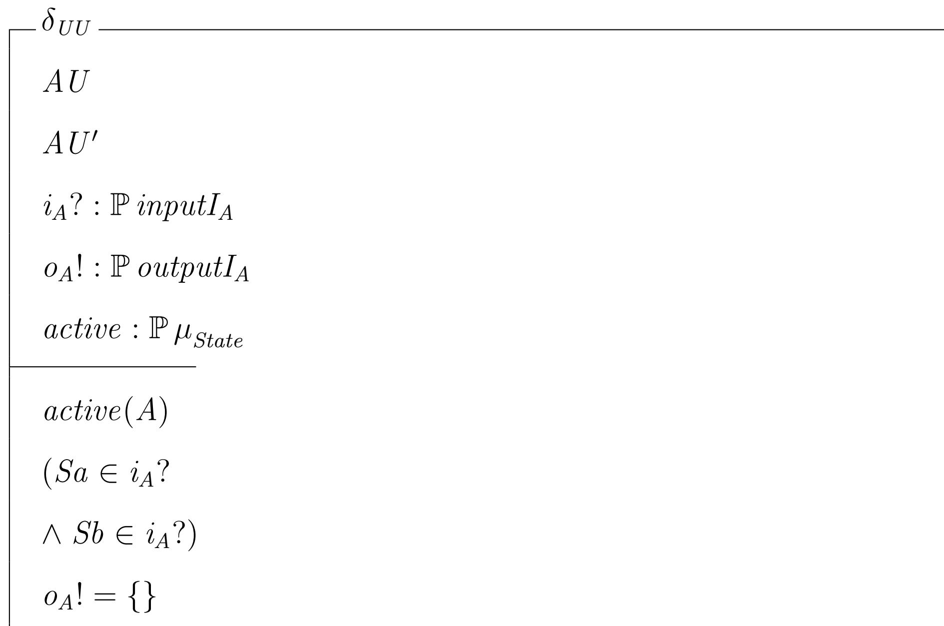


Figure 2.2: μ-chart with more complex transitions

In addition to the simple sequential chart seen in Figure 2.1, μ-Charts also provide support for more complex transitions, such as transitions that loop back to the same state and transitions that make use of more than one input signal. An example of this is shown in Figure 2.2.

This chart features two transitions, one of which is a loop transition which starts and ends on the same state. Within the guards of the transitions, we see that they use AND, denoted by the dot symbol, and OR, denoted by the plus symbol. An AND will be true only if both pieces of the guard are true, whereas an OR will be true if one or both pieces of the guard are true.



δ_{UU}

AU

AU'

$i_A? : \mathbb{P}\, inputI_A$

$o_A! : \mathbb{P}\, outputI_A$

$active : \mathbb{P}\, \mu_{State}$

$active(A)$

$(Sa \in i_A?$

$\wedge\ Sb \in i_A?)$

$o_A! = \{\}$

The semantics of these transitions are very similar to the transition operation

schema seen in the previous example, as can be seen in the semantics of the $a.b$ loop transition above in δ_{UU}. This operation schema features two points of difference in comparison to the δ_{XY} transition schema detailed earlier. The first is that due to this transition being a loop transition from state U, or in this case state AU due to the chart name being prefixed to the state name, back to the same state. As a result, the name of the schema, δ_{UU}, features the name of the state twice and the current and next state within the declarations is said to be the state AU.

The second point of difference is the predicate $(Sa \in i_A? \wedge Sb \in i_A?)$. Unlike μ-Charts, Z uses the $\wedge$ symbol as its logical AND and as such, this predicate can only be true if both the Sa and Sb signals are sent to the system as input signals. The δ_{UV} is not shown but uses Z's logical OR, $\vee$.

2.2.2 Decomposition operator

The decomposition operator is a feature of μ-Charts that consists of a chart that exists within another chart, as seen in Figure 2.3. We name these two charts the parent and child. The child chart is represented within the parent chart by a rectangular state, which features the same name as the child chart. We call this the decomposed state. In Figure 2.3, this can be observed in the parent chart, *Parent*, which features the decomposed state *Child*, and then the child chart, *Child*, which obviously features the same name as the decomposed state. Both the parent and child charts are initialised at their starting states: state X in the *Parent* chart and state Z in the *Child* chart. Input is shared by the entire system (i.e. both charts *Parent* and *Child*) but transitions will only occur when a chart is active and the *Child* chart is only active when the decomposed state *Child* is either the parent chart's current state or will be its next current state.

When a chart is inactive, the semantics of μ-Charts defines it so that the chart will retain its current state until it becomes active again and a transition

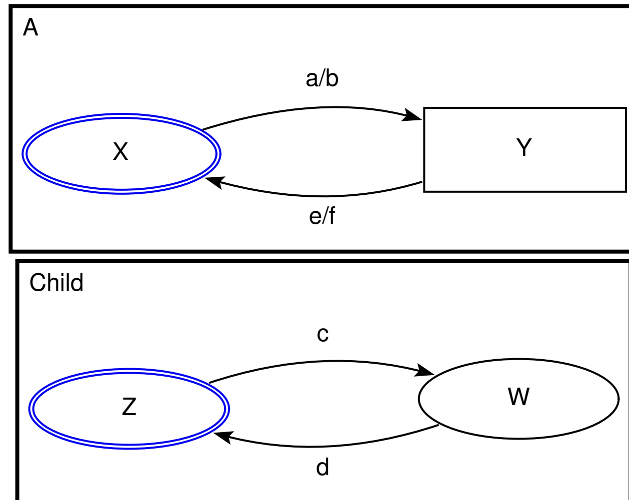


Figure 2.3: Decomposed μ-chart

may occur within it. Using Figure 2.3 as an example, if the current state of the *Parent* chart was the decomposed state *Child* and the current state of the *Child* chart was state W, the *Child* chart would become inactive if the e/f transition in the *Parent* chart occurred and it transitioned to state X. Even if the input signal d were then received, the *Child* chart would remain in state W until it became active again.

The Z semantics of decompositions are more complex than a sequential chart. Both the *Parent* and *Child* charts are translated in a similar fashion as chart A in the previous example but there are additional schemas added to the semantics to detail the relationship between those two charts. The name of these schemas within this example is *ParentChildDec*, which is named by combining the name of the parent and child charts together and then adding the *Dec* suffix to the end, indicating that these schemas are related to the decomposition.

$states_{ParentChildDec} : \mathbb{P}\, \mu_{State}$

$inputI_{ParentChildDec} : \mathbb{P}\, Signal$

$outputI_{ParentChildDec} : \mathbb{P}\, Signal$

$states_{ParentChildDec} = states_{Parent} \cup states_{Child}$

$inputI_{ParentChildDec} = inputI_{Parent} \cup inputI_{Child}$

$outputI_{ParentChildDec} = outputI_{Parent} \cup outputI_{Child}$

Above we see the first *ParentChildDec* schema. Like the chart schemas shown earlier, it shows the declaration and constraint of global constants. The three constants are very similar to the example shown previously, with the states, input and output all declared. The predicate, however, is very different. Here, instead of constraining them to specific values, we are using the union, ∪, to constrain each constant to the combination of the relevant *Parent* and *Child* constants. As such, if we assume $states_{Parent} = \{ParentX, ParentChild\}$ and $states_{Child} = \{ChildZ, ChildW\}$, $states_{ParentChildDec}$ would then consist of {*ParentX, Parent-Child*, *ChildZ*, *ChildW*}.

$Chart_{ParentChildDec}$

$Chart_{Parent}$

$Chart_{Child}$

$Init_{ParentChildDec}$

$Init_{Parent}$

$Init_{Child}$

The next two schemas are the $Chart_{ParentChildDec}$ and $Init_{ParentChildDec}$ schemas. These schemas are very simple in that they only include the equivalent *Parent* and *Child* chart schemas. The result of this is that $Init_{ParentChildDec}$ schema initialises both charts, while $Chart_{ParentChildDec}$ is the combined state space of both charts.

$\delta_{ParentChildDec}$

$$\Delta Chart_{ParentChildDec}$$
$$i_{ParentChildDec}? : \mathbb{P}\, inputI_{ParentChildDec}$$
$$o_{ParentChildDec}! : \mathbb{P}\, outputI_{ParentChildDec}$$
$$active : \mathbb{P}\, \mu_{State}$$

$$((ParentChild \lor ParentChild') \land active(Parent)) \Leftrightarrow active(Child)$$
$$\exists\, i_{Parent}?, i_{Child}?, o_{Parent}!, o_{Child}! : \mathbb{P}\, Signal \bullet$$
$$i_{Parent}? = i_{ParentChildDec}? \cap inputI_{Parent} \land$$
$$i_{Child}? = i_{ParentChildDec}? \cap inputI_{Child} \land$$
$$o_{ParentChildDec}! = o_{Parent}! \cup o_{Child}! \land$$
$$\delta_{Parent} \land \delta_{Child}$$

Finally, we see the last *ParentChildDec* schema above, the $\delta_{ParentChildDec}$ operation schema. This is the schema that details the relationship between the *Parent* and *Child* charts and, as a higher level schema than the δ_{Parent} or δ_{Child}, is then included within the system operation schema, which in this case is *ParentSys*.

Many of the declarations within this schema are similar to previous examples. First it declares that there may be state changes within this schema. Then it declares the input, output and active state.

The predicate, however, is very different. The first line concerns whether the *Child* chart is active. It uses an if and only if, $\Leftrightarrow$, to ensure that both sides have the same value, whether true or false. If one side is true while the other is false the predicate will be false and this operation schema will not be able to occur. As such, the right hand side of the if and only if operator, which states that the *Child* chart is active, can only be true if the left hand side is true. The left hand side will be true if both the *Parent* chart is active and the decomposed state *ParentChild* is either the current state or the next state of *Parent* chart. As the System schemas already define the *Parent* chart as always active, this means that

whether the *Child* chart is active depends entirely upon whether the decomposed state *ParentChild* is the current or next state. If it is not then the *Child* chart must be inactive.

The rest of the predicate within this schema is a conjunction. First it states that the input and output observations of the *Parent* and *Child* charts exist as having power set types of the *Signal* type. These can be considered the same as each of these charts operation schema's input and output observations. It follows this by using the intersection, $\cap$, to ensure that any input received by either chart must be part of the respective chart's input global constant. This essentially filters the input the $\delta_{ParentChildDec}$ operation schema receives down to the relevant charts. This is followed by the combination of the output received from both *Parent* and *Child* charts using the union to form the *ParentChildDec*'s output. Then finally, the delta operations schemas of both *Parent* and *Child* schemas are included.

2.2.3 Feedback operator

The feedback operator is a feature of μ-Charts that allows us to instantaneously feed a set of designated output signals back into the chart as input signals. As such, we can use this feature to send signals between multiple charts within a composed or decomposed chart.

The graphical representation of the feedback signal set is presented in a box located below the μ-Chart. This box is used to display the chart's feedback, hidden and filtered signals, the latter two of which are μ-Charts features detailed in [13], but for our purposes, we only make use of the feedback signals. A form of syntactic sugar is used to help simplify this. If there is only one set of signals within the box, this means that there are no hidden or filtered signals within this chart and the set must be the feedback signal set.

Within a decomposed chart, the relationship between the parent and child charts results in some unique rules in regards to the feedback operator. Both the

parent and child charts have their own sets of feedback signals but only the signals included within the parent chart's feedback signal set will be received as input by both charts. As such, the child chart will receive feedback signals from the parent chart even if these signals are not included within its own feedback signal set but in order for the child chart to send a feedback signal to the parent chart, this signal must be included within both the parent and child charts' feedback signal sets. An example of this is shown below in Figure 2.4.

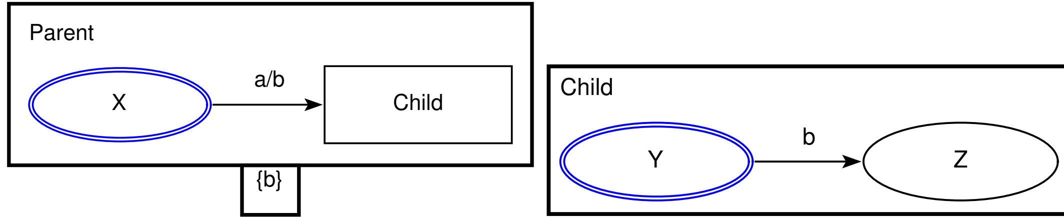


Figure 2.4: μ-chart with feedback

The feedback signal set located at the box at the bottom of the *Parent* chart contains the signal b, meaning that if the *Parent* chart outputs the signal b, it will feed this signal back into both the *Parent* and *Child* charts as an input signal. Using this example, if we were to receive the input signal a, the transition labelled a/b would occur. This transition would both make the decomposed chart active and output the signal b. This signal which would then be fed back into the system as the input signal b and would be received by the *Child* chart. The *Child* chart would transition to state Z due to this. All of these actions would occur immediately, within a single step.

Shown below are some excerpts from the semantics of this chart.

$$\Psi_{Parent} : \mathbb{P}\, Signal$$

$$\Psi_{Parent} = \{Sb\}$$

Seen above is the declaration of the *Parent* chart's feedback constant, which we represent using the Ψ symbol. This is then constrained to the signal Sb, as

this is the only feedback signal within the chart.

$$Sa \in i_{Parent}? \cup (o_{Parent}! \cap \Psi_{Parent})$$
$$o_{Parent}! = \{Sb\}$$

Above we see the two relevant pieces of predicate from the *Parent* chart's transition from state X to the decomposed state *Child*.

With the addition of feedback, the input signal predicate is extended to take the local chart's feedback signals into account. We determine which output signals have been fed back into the chart by getting the intersection of the output and feedback signal sets and then combine this with the input signal set to check whether the input signal *Sa* has been received. This extension to the predicate is redundant within a decomposition's parent chart due to the *ParentChildDec* operation schema predicate detailed below but in a sequential or child chart, this is how output signals are fed back into the chart.

In this same transition, we also include the signal *Sb* as output, which will later be fed back into the system.

$$Sb \in i_{Child}? \cup (o_{Child}! \cap \Psi_{Child})$$

In the *Child* chart, we see the above predicate in the transition from state Y to state Z. In this case, the feedback signal *Sb* will be received as an input signal.

$$\Psi_{ParentChildDec} : \mathbb{P}\, Signal$$
$$\Psi_{ParentChildDec} = \{Sb\}$$

The remaining changes are made within the *ParentChildDec* schemas, which define the relationship between the *Parent* and *Child* charts. We see the first change above, where the feedback constant is identical to the *Parent* chart's feedback constant. This is because the *Parent* chart is the main chart within the system, so only signals listed within its feedback set may be sent between the

two charts. If the *Child* chart had its own feedback signals, these would not be included within this feedback set.

$$i_{Parent}? = (i_{ParentChildDec}? \cup (o_{ParentChildDec}! \cap \Psi_{ParentChildDec})) \cap inputI_{Parent} \land$$
$$i_{Child}? = (i_{ParentChildDec}? \cup (o_{ParentChildDec}! \cap \Psi_{ParentChildDec})) \cap inputI_{Child}$$

Finally, within the *ParentChildDec* operation schema, the input signals of both the *Parent* and *Child* charts are defined as a combination of the input that has been received from the environment and the output signals that are meant to be fed back into the charts. This means that if the *Parent* chart outputs the signal *Sb*, this schema will ensure it will be received by both charts as an input signal.

2.2.4 Value carrying signals

Value carrying signals are signals that contain values within them. These signals can be used to help simplify charts, as related signals can be grouped together as possible values of a single value carrying signal, rather than be represented by multiple different signals. We can also send value carrying signals as output and use them in conjunction with the feedback operator and local variables.

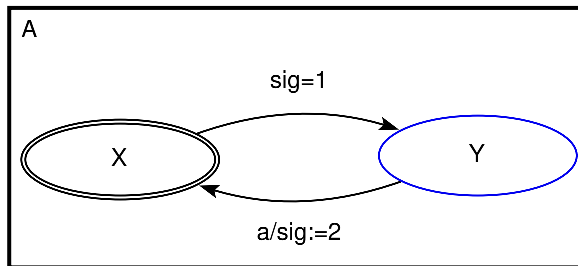


Figure 2.5: μ-chart with value carrying signals

Seen in Figure 2.5 is a chart containing the value carrying signal *sig*, which is used within both transitions. In this case, we want the values this signal carries to be of an integer type and so we make sure our use of it is restricted solely to integers. The first transition has a guard of $sig = 1$, so for this transition to occur,

sig must be received containing the value 1. The second transition has $sig := 2$ within its action, which means that when this transition occurs, it outputs *sig* with the value 2.

The semantics of value carrying signals is slightly different to a standard signal, which can be seen below in the system's *Signal* type declaration.

$$Signal ::= Sa \mid Ssig\langle\langle \mathbb{Z} \rangle\rangle$$

It is declared in a similar fashion as standard signals, with the S prefix added to the signal name, but is followed by its value type within double angle brackets. In this case, its value is of the integer type but this type depends on what values it has been set to within the μ-Chart.

$$inputI_A = \{Sa\} \cup \{n : \mathbb{Z} \bullet Ssig\, n\}$$
$$outputI_A = \{n : \mathbb{Z} \bullet Ssig\, n\}$$

The next addition is to the chart's input and output global constants. The value carrying signals in both are defined in the same way, with the integer n declared such that *Ssig* contains n. They are defined in a separate set than standard signals because of their differences.

$$Ssig\, 1 \in i_A?$$

Above we see the input predicate from the example's first transition, which means that for that transition to occur, the signal *Ssig* must be received containing the integer 1.

$$o_A! = \{Ssig\, 2\}$$

Last, we see the output predicate of the example's second transition. When this transition occurs, this means that the signal *Ssig* will being sent as output containing the integer 2.

2.2.5 Local variables

Local variables are used within μ-Charts to store a value which may then be used within a transition's guard or action. The use of this can be seen within Figure 2.6, where the local variable var is initialised in the top right corner to its default value $start$. This local variable is then used within both of the chart's transitions, demonstrating how local variables can be used within the guard and action. The first transition, labelled $a/var := stop$, occurs when the state X is the current state and the input signal a is received. The local variable var is then set to the value $stop$ within the action of this transition. The second transition, labelled $var = stop/var := start$, then occurs when the current state is Y and the local variable var is $stop$. As the local variable var was set to the value $stop$ within the previous transition, this means that the transition may occur straight after the previous transition occurred. Then within the action of this transition, the local variable var is set to the value $start$.

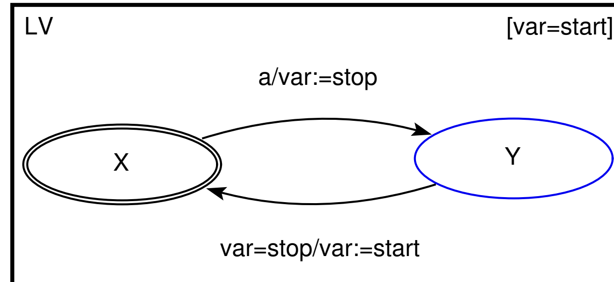


Figure 2.6: μ-chart with local variable

The inclusion of local variables to a chart adds a few new points of details to the Z semantics, which is detailed below.

$$FValvar ::= stop \mid start$$

We first see the local variable type definition, which defines a type $FValvar$ to either be the value $stop$ or the value $start$. This is named $FValvar$ by prefixing $FVal$ onto the local variable name, var.

$$\begin{array}{|l}
\hline
\textit{Variables}_{LV} \\
\textit{Vvar} : \textit{FValvar} \\
\hline
\end{array}$$

Next is the $Variables_{LV}$ schema, which declares the local variable $Vvar$ of type $FValvar$. This schema is used to declare any local variables to be used within the state space. Similar to the local variable type, the names are created by prefixing V onto the local variable name.

$$\begin{array}{|l}
\hline
\textit{Chart}_{LV} \\
c_{LV} : \textit{states}_{LV} \\
\textit{Variables}_{LV} \\
\hline
\end{array}$$

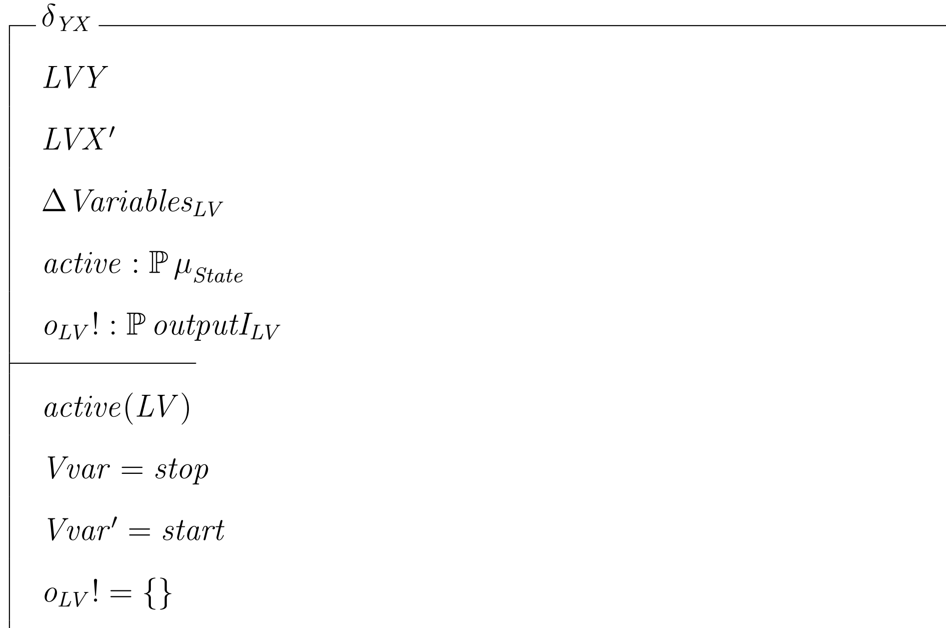
Above see the $Chart_{LV}$ schema, which is the state space of the chart. This declares both the current state and the $Variables_{LV}$ schema, which in turn adds all local variables declared within it to the state space.

$$\begin{array}{|l}
\hline
\textit{Init}_{LV} \\
\textit{Chart}_{LV} \\
\hline
c_{LV} = LVX \\
\textit{Vvar} = \textit{start} \\
\hline
\end{array}$$

Next is the $Init_{LV}$ schema, which as has been explained before, initialises the chart. The only difference from previous examples is that it initialises the local variable, $Vvar$, is initialised to the value $start$, as it was in Figure 2.6.

$$
\begin{array}{|l}
\hline
\delta_{XY} \\
LVX \\
LVY' \\
\Delta Variables_{LV} \\
i_{LV}? : \mathbb{P}\, Signal \\
active : \mathbb{P}\, \mu_{State} \\
o_{LV}! : \mathbb{P}\, outputI_{LV} \\
\hline
active(LV) \\
Sa \in i_{LV}? \\
Vvar' = stop \\
o_{LV}! = \{\} \\
\hline
\end{array}
$$

Above is the δ_{XY} operation schema, which is the semantics of Figure 2.6's first transition, labelled $a/var := stop$. There are two parts of this schema that are relevant to local variables. The first is that the $Variables_{LV}$ schema is declared using the upper case delta symbol, indicating that there will be changes to the local variable observations within it. The second is the line $Vvar' = stop$ within the predicate, which sets the primed local variable $Vvar'$ to the value *stop*, meaning that the local variable *Vvar* will be set to *stop* in the next state of the system. This is an accurate translation of the transition's label $a/var := stop$.



Finally we see the δ_{YX} operation schema and the Z semantics of Figure 2.6's second transition, labelled $var = stop/var := start$. Like the previous example, it indicates that there will be changes to the local variable observations within the $Variables_{LV}$ schema using the upper case delta symbol. Then within the predicate, it first ensures the current value of the local variable $Vvar$ is $stop$ and sets the primed local variable $Vvar'$ to the value $start$. So in order for the transition to occur, the current value of $Vvar$ must be $stop$ and then after the transition occurs, $Vvar$ will be set to the value $start$.

While not shown, an additional change is made to the ϵ_{LV} and $Iactive_{LV}$ schemas, adding the predicate $Vvar' = Vvar$. This ensures that the value of the local variable $Vvar$ remains unchanged if either of these operation schemas occur.

2.3 Tools

Within this section, we will talk about the tools AMuZed, ZooM and ProZ.

2.3.1 AMuZed and ZooM

AMuZed and ZooM are tools designed for use with μ-Charts and in conjunction with one another. AMuZed is a graphical editing tool designed with the intention to create, save, edit and print μ-charts. It was used to create the μ-charts that appear within this paper. ZooM is a tool that is used to convert μ-charts into Z using the semantics that defines μ-Charts, first by checking that the syntax of the chart is correct and then translating it into a LaTeX-formatted Z specification.

AMuZed and ZooM were both developed in Haskell in parallel with one another, so their source code is integrated together into a group of source files. Their graphical user interfaces were created using Tcl/Tk via an interface named TclHaskell. Unfortunately, support for TclHaskell has since been abandoned and it is no longer compatible with modern operating systems, so they can only be run from a virtual machine that has been specifically created to run them. Efforts to re-engineer both tools have been made but neither are at a point where they can fully replace them.

AMuZed is shown in use in Figure 2.7. The tool bar shown offers a number of options, such as creating a new chart, opening and saving charts with AMuZed's .muz file extension and outputting the chart as an image, and tools, such as creating new states, transitions and adding a local variable. These tools are used to build the μ-charts. The window shown on the right hand side displays the graphical model. In addition, each decomposition that exists within the model is shown in an additional window.

ZooM features a minimalist interface. It is comprised of an open file dialogue, which the user can use to select the .muz file they want to translate into its Z semantics, and an error dialogue, which will display any errors encountered during translation.

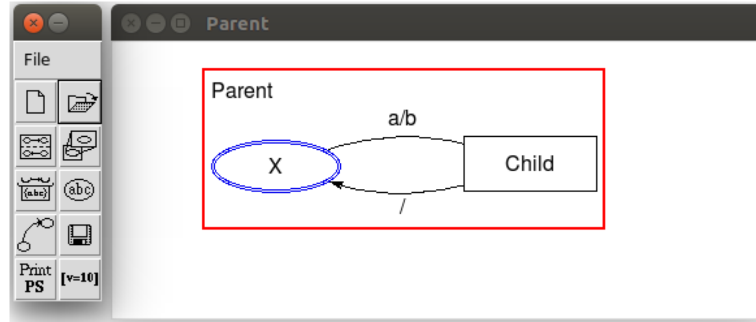


Figure 2.7: AMuZed user interface

2.3.2 ProZ

ProZ is an extension to the ProB Animator and Model Checker tool [10], which is an animator, graphical visualiser, constraint solver, simulator and model checker for B [1], a formal specification language similar to Z. ProZ uses the Fuzz type-checker [15] to typecheck and extract the formal content from Z specification files when they are loaded into ProB. It then parses this content and translates it into B [10], allowing ProB to simulate it as if it were a B specification.

Figure 2.8 shows the user interface of ProZ, which consists of a toolbar at the top and four panes. The top pane is a text editor we call the specification editor, which allows us to read and edit Z specifications. Changes that are made are not immediately simulated by ProZ, instead the file needs to be saved and reopened using an item in the File menu, in order for it to parse and simulate the specification. The bottom three panes, from left to right, are the State Properties, Enabled Operations and History panes. The State Properties pane lists the current values of the system's state observations and constants, providing the user with the current state of the system. The Enabled Operations pane provides a list of operations that are possible in the current state of the system. Some of these operations require inputs, in this case ProZ provides possible input observation values that the user can choose. This means that the same operation may appear within the Enabled Operations list multiple times. In μ-Charts the input signals

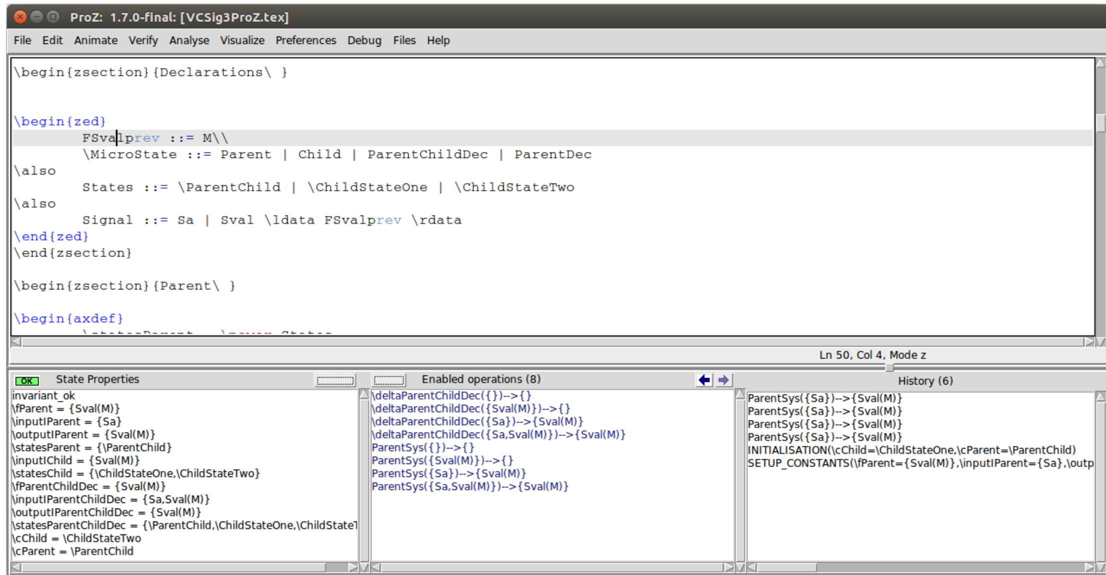


Figure 2.8: ProZ user interface

are received from the environment and may or may not be user controlled, so the interpretation in ProZ is slightly different from the μ-Charts interpretation.

The resulting output of each operation is also shown in curly brackets following an arrow. Each operation will be shown in a colour depending on their outcome, and we are primarily focused on the blue, green and black operations. Blue operations make no changes to the current state of the system, green operations lead to a new, unexplored state of the system and black operations lead to a different but previously explored state of the system. The History pane provides a list of the operations that have occurred to reach the current state of the system and can be used to move back to a previous state.

Using ProZ, we have the ability to simulate Z specifications and explore the effect operations have on the state of the system. This allows us to test whether specifications we have created work as intended or have any bugs, by extensively exploring each possible operation and ensuring the system's observations are as we intended.

2.4 Design Patterns

The use of design patterns to graphically model interactive systems was first introduced in the paper *Design Patterns for Models of Interactive Systems* by Bowen and Reeves [2]. Within this paper, Bowen and Reeves defined two patterns: the Callback and Binary Choice patterns.

2.4.1 Callback pattern

The Callback pattern was created to model the behaviour of a dialogue box that offers two choices, as well as the ability to cancel and return to the user's previous state. It uses a local variable to store the user's previous state while transitioning to the dialogue box and then uses that variable to determine which state it will return to if cancelled. An example of this pattern can be seen in Figure 2.9.

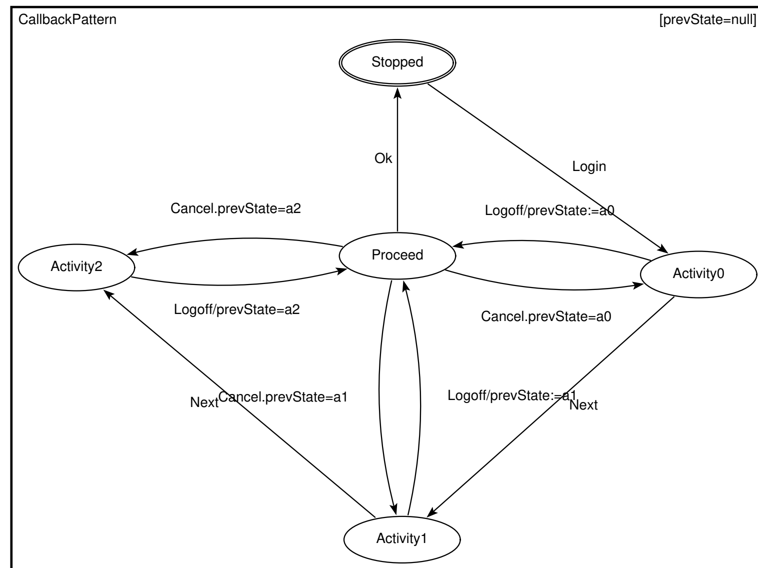


Figure 2.9: Callback pattern

This Callback pattern example is a model of a logout confirmation dialogue. The *Activity*0, *Activity*1 and *Activity*2 states represent possible windows within the interface, the *Stopped* state represents the logged out state within the system and the *Proceed* state represents the dialogue box where the user is asked to

confirm whether they want to log out. The *prevState* local variable is used to store a value representing the previous state, which is then used to determine which state it will return to if the logout dialogue is cancelled.

Normally, a design similar to the logout dialogue can lead to issues for a modeller, as it requires the logout dialogue to be available from every state within the system and be able to return to the previous state if the dialogue is cancelled. As such, a modeller may choose to model the logout dialogue individually for each state within the system but this is an unnecessarily complex solution which leads to a much larger state space. The Callback pattern is a much more suitable solution.

2.4.2 Binary Choice pattern

The Binary Choice pattern was created to model the behaviour of an interactive system where the user inputs something and the part of the system they end up is dependent on what they input. We model this within the pattern using a value carrying signal which is used to represent the system's behaviour having received the input.

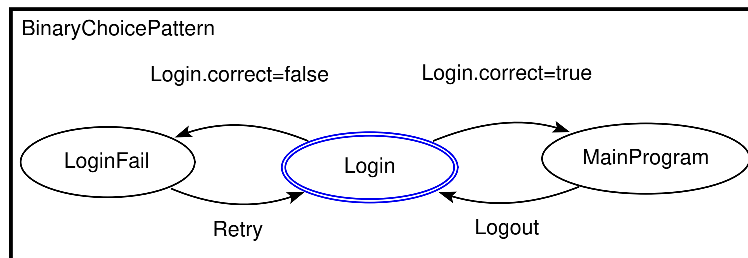


Figure 2.10: Binary Choice pattern

Figure 2.10 shows an example of the Binary Choice pattern within a model of a login window, where the *Login* state represents the login window. If the login details are correct, a signal *correct* will be received carrying the value *true* and will transition to the *MainProgram* state, which is self explanatory. If the login

details are false, *correct* will be received carrying the value *false* and the system will instead transition to the *LoginFail* state, where an error message may be received.

While there are other possible ways to model this system and this pattern may be more complex than some solutions, it accurately reflects both how the user interacts with the system and how the user interface behaves.

Chapter 3

Design Patterns

In this chapter we discuss finding design patterns and then lay out the Return Home and Return patterns.

3.1 Discovering Design Patterns

Modern interactive systems are often incredibly complex, which, in turn, makes modelling these systems incredibly complex as well. In order to manage this complexity, we use a technique called *abstraction* to model at a level of detail we find relevant, while suppressing the more complex details below this level. We call these levels of detail, the *levels of abstraction*.

During our research, we found that one of the most important parts of discovering design patterns is finding the appropriate level of abstraction to view systems at: one that ensures the model retains enough detail to capture all the important elements of a system's design, while avoiding adding excessive detail that may hamper discoveries.

One of the systems we first attempted to model in our work to discover design patterns was the PlayStation 4 user interface, named the PlayStation Dynamic Menu, and was based on the 2.50 version of it. This system was chosen because

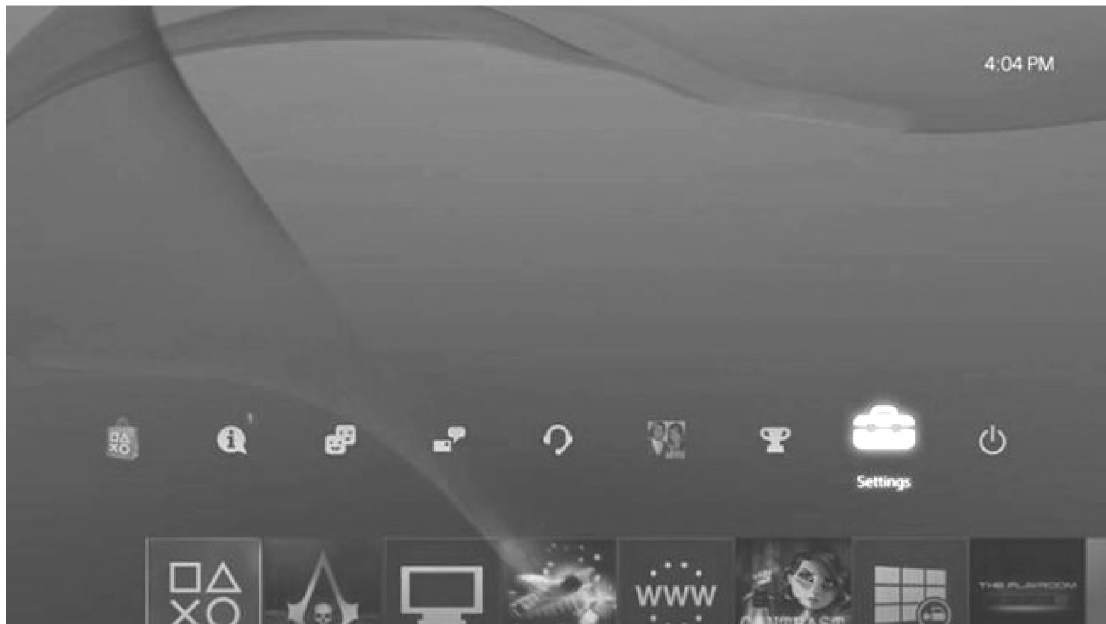


Figure 3.1: PS Dynamic Menu user interface

it is a well-regarded interface for a popular entertainment device, which made it interesting to analyse for design patterns.

Seen in Figure 3.1 is the home menu of the PlayStation Dynamic Menu. There are two levels of widgets within it. The bottom level consists of recently used game and media applications, as well as the Library on the far right of that menu, where all purchased applications are available to use or download.

The top level provides a variety of different widgets that will take the user to different parts of the interface. From left to right these are the Store, Notifications, Friends, Messages, Party, Profile, Trophies, Settings and Power settings. The Store is used to buy and download both game and media applications, as well as additional content and services that may be available. The Notifications screen provides news relevant to the user, such as whether there have been any friend or multiplayer invites to them, download progress, etc. The Friends screen provides a list of the user's current friends, which can be selected to view each player's profile, as well as a list of friend invites and players the user has recently played with in a multiplayer game. The Messages screen provides a list of ongoing conversations,

which once selected will display the full conversation, as well as providing the ability to create a new conversation with a friend of the player's choosing.

The Party screen provides a list of ongoing parties of players, where they can talk to one another via text or audio chat, invite friends and play together in multiplayer games. There is also a widget that allows the user to create a new party and invite their friends upon its creation. The Profile screen provides a variety of information on the user, such as their friends and trophies. The Trophies screen displays a list of games the user has played, as well as the in-game trophies they have been awarded for completing specific objectives. Within this screen, there is the Compare Trophies widget, which lets the user compare their trophies to the trophies their friends have collected. The Settings screen provides a large number of different settings that can be adjusted, such as network, audio, display, accessibility and storage settings. Lastly, the Power settings provides the user the ability to log out, put the system into rest mode, restart it or turn it off.

Figure 3.2 shows an attempt at modelling the PlayStation Dynamic Menu. As can be seen there are a large number of states that represent screens with significant inter-connectivity between different areas of the user interface. As this chart was intended to model the user interface and not potential video games or media apps that can be run on it, which have different user interfaces and behaviours, these are represented by the single *App* state and not delved into.

The starting state *Home* represents the main screen of the user interface, where all the features and applications are available for selection. From this state, other screens of the interface branch from it, such as Notifications, Friends List, Party List, etc. These screens, as well as their own sub-screens, may then provide the option to lead the user to a different screen or sub-screen within the interface, creating branches within the chart that represent these sections of the interface. Additionally, pressing the circle button on the controller will lead the user back to their prior screen. This functionality is difficult to model because of the numerous

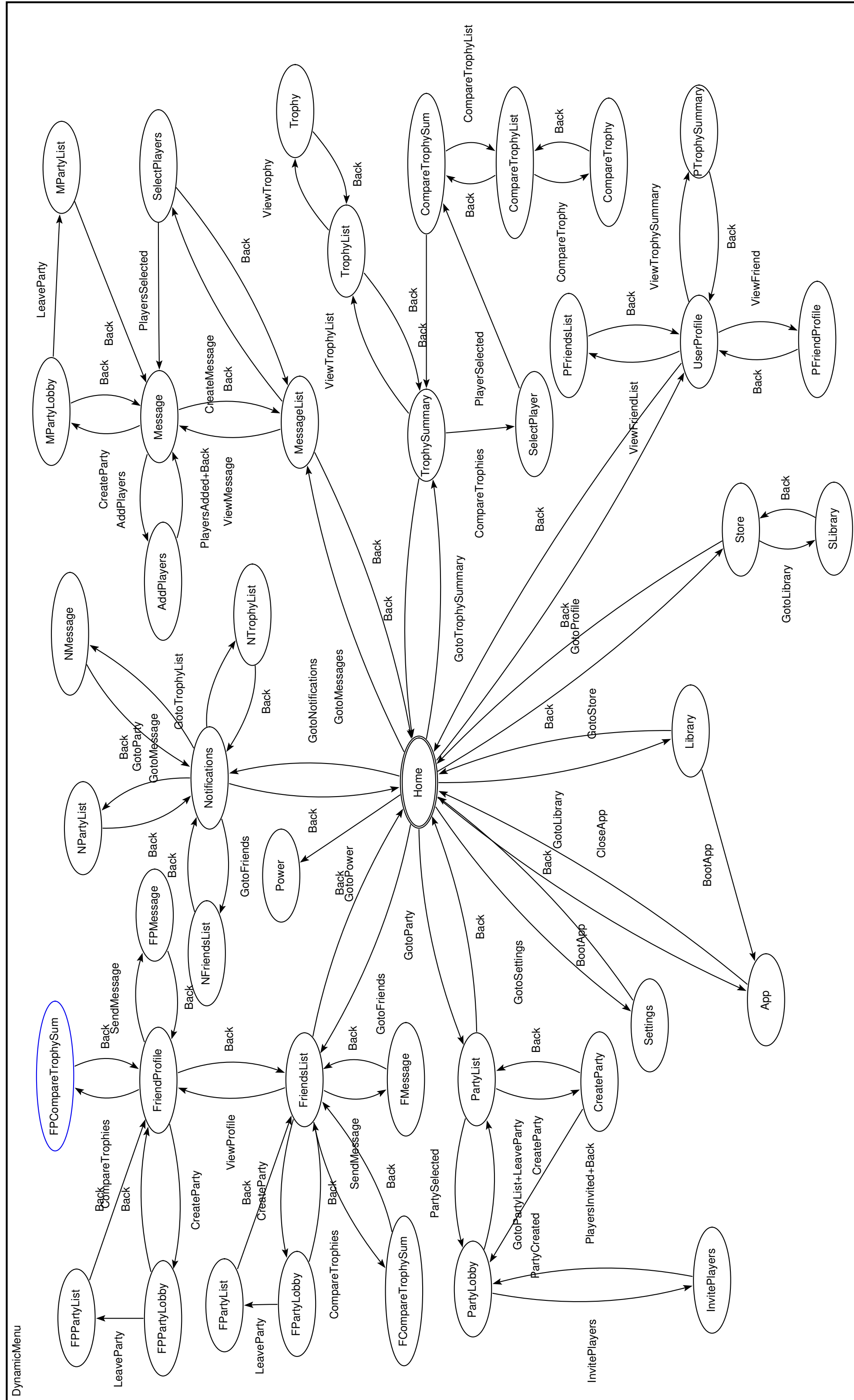


Figure 3.2: PS Dynamic Menu chart with states representing duplicate branches

transitions required to model every option, while taking into account the ability to go back to the prior screen.

In an effort to help simplify the chart, a form of syntactic sugar has been used in the form of duplicate states that represent other screens within the interface. These duplicate states are prefixed with an abbreviation of the prior state. For example such as *FPParty* is a duplicate the *Party* state where *FriendProfile* was its previous state. It is assumed that they behave identically to the original state, as well as the states that branch off of them, but with the *Back* input signal returning back to the appropriate prior state.

Another model of the interface can be seen in Figure 3.3. This chart is similar to the chart seen in Figure 3.2 but with a goal to remove the duplicate states by using a local variable to track the prior state and then using it in conjunction with input signals within transitions to ensure that when the user tries to return to their prior state, they move back to the appropriate state. This chart makes use of the local variable *PrevState*. It is used by setting its value to an abbreviation that represents the previous state, such as in the transition labelled $GotoFriendsList/PrevState := UP$.

The transition labelled $GotoMessages/PrevState := FL$ leads to the state *MessageList* if the input signal *GotoMessages* has been received and, as its action, it sets *PrevState* to the value *FL*, which represents the prior state *FriendsList*. This local variable is then used within the transition labelled $Back.PrevState = FL$, which will only occur if the *PrevState* local variable is set to the value *FL* and the input signal *Back* has been received.

This model removes 18 of the duplicate states seen within Figure 3.2 but adds a significant amount of complexity to the model, as can be seen within this chart by the number of transitions that cross over one another.

It could be argued that there would be fewer transitions crossing over one another if the states were rearranged, the chart was larger or even that the chart

Figure 3.3: PS Dynamic Menu chart with local variable storing previous state

was modelled in 3D. These are options but the fact that these suggestions had to be made emphasises how complex the chart is. The inter-connectivity of the various different states within it make creating a clear and concise chart very difficult.

The μ-Charts specification AMuZed is based on only allows a local variable to store one value at a time. Due to this, when we use the *PrevState* local variable to record the previous state, it loses the value it had previously stored. To avoid potential issues this may cause, we created an informal rule dictating that the *PrevState* variable should be set once on any possible branch of the chart, meaning that there should be no possible branch where the *PrevState* variable is set twice or more. Unfortunately, when we initially created Figure 3.3 the merging of branches resulted in a chart that was so complex, particularly surrounding the *PartyList* branch, that this issue occurred numerous times but was not immediately identified. This helps emphasise how useful and important reducing the complexity of μ-Charts is, while also suggesting that further additions to the μ-Charts specification AMuZed is based on would help reduce complexity.

In this case, the ability to use an array of values would have eliminated this issue but as the μ-Charts specification we use does not have this functionality, we had to find other solutions.

In an effort to resolve these issues, our initial plan was to eliminate the potential second variables by duplicating these branches, seen in Figure 3.4. The use of these duplicate states is not the same as the duplicates seen in Figure 3.2, as we did not want to use any syntactic sugar to cut corners with this chart. Instead, we needed to fully model all states and transitions within each duplicated branch. These duplicate states are denoted with an additional digit which indicates that they are duplicate states. For example, the first duplicate of the *PartyList* state will be named *PartyList*2, the second will be named *PartyList*3 and so on.

As the *PrevState* variable had already been set in the transitions to the

Figure 3.4: PS Dynamic Menu chart with local variable and duplicate states

Message, *FriendProfile* and *FriendList* states, this meant that their following transitions to the *PartyList* branch set the value of the *PrevState* variable for a second time. As a result, the *PartyList* branch then needed to be duplicated once for each of these states, leading to an additional 12 states being added to the chart, as the *PartyList* branch is four states that needed to be duplicated three times.

The next instance of this issue involved the *CompareTrophiesSum* sub-branch of the *Trophies* branch. The *FriendProfile*, *FriendsList* and *SelectPlayer* states all set the *PrevState* variable prior to their transitions to the *CompareTrophiesSum* state, so this was again resolved by duplicating this sub-branch for the *FriendProfile* and *FriendsList* states. This led to an additional six states being added to the chart, as the *CompareTrophiesSum* branch is three states that needed to be duplicated twice.

Finally, the third instance of this issue involved the *TrophyList* state, which both the *TrophySummary* and *Notifications* states transitioned to and set the *PrevState* value in doing so. While transitioning to the *TrophyList* state via the *Notifications* state only sets the value of the *PrevState* variable once, transitioning to it via the *TrophySummary* state sets it twice. To resolve this issue, we once again needed to duplicate these states for the *Notifications* state, adding an additional two states to the chart. Overall, these changes resulted in an additional 20 states being added to the chart, making it significantly larger than Figure 3.3 and significantly more complex.

Unfortunately, as we were solving these issues, we continued to encounter other instances of this issue. For example, transitioning from the *UserProfile* state to the *FriendList* state sets the value of the *PrevState* variable once, which can then be followed by transitioning to the *FriendProfile* state, where the variable is set for a second time, and then finally it can also be followed by transitioning to the *Message* state, where the variable is set for a third time. Continuing to use

this plan would have led to an additional 20 states being added to the chart by duplicating the states involved in that sequence alone. As a result of these continual issues, we decided to abandon this plan.

Our next plan to solve the issues seen in 3.3 was to use multiple local variables to help track prior states. The reason this plan was considered after the previous plan was because we believed using multiple local variables would make the chart considerably less legible upon an initial view, as a viewer would have to track which local variables were in use at any one time. However, as the previous plan would have eventually led to at least 40 new states being added to the chart, we believed the issues that may arise with multiple local variables were a necessary level of complication.

Using this plan, we determined that the best way to implement multiple local variables was to create a new local variable for each state that is transitioned into by more than one state. In doing so, we identified nine states that did this: the *PartyList*, *PartyLobby*, *Library*, *TrophySummary*, *TrophyList*, *CompareTrophy*, *Message*, *FriendList* and *FriendProfile* states. We then attempted to reduce the number of local variables needed by these states by figuring out whether some of the variables associated with states that are part of the same branch could be combined without causing too much complexity or additional issues to arise.

As an example of this, the *TrophySummary* and *TrophyList* states are both within the same branch and they both are transitioned into by two states. The *TrophySummary* state can be transitioned to from the *Home* and *UserProfile* states, while the *TrophyList* state can be transitioned to from the *TrophySummary* and *Notifications* states. Using a single local variable with these two states, which we call *TrophyPState*, standing for the Trophy Previous State, can be done with no issues and only requires changes to two transitions. The first change is to the transition from *TrophySummary* to *TrophyList*, which was previously labelled by $ViewTrophyList/PrevState := TS$. As both states now use the same

local variable, it no longer needs to set it to the abbreviation that represented the *TrophySummary* state. As such, the label for this transition is now simply *ViewTrophyList*.

The second change is to the transition from the *TrophyList* back to the *TrophySummary* state, which was previously labelled as $Back.PrevState = TS$. As the local variable is no longer set to the abbreviation that represents the *TrophySummary* State, we need to instead check whether the local variable is set to the abbreviation that represents one of the states that previously transitioned into the *TrophySummary* state. As the *Home* state is represented by H and the *UserProfile* state is represented by UP, this leads to the transition label $(Back.TrophyPState = H) + (Back.TrophyPState = UP)$, which can be shortened into a more concise $Back.(TrophyPState = H + TrophyPState = UP)$.

However, we chose not to combine the *FriendsList* and *FriendProfile* local variables, as we believed there were too many states that transitioned to those states and combining the local variables would lead to more complexity within the chart.

The result of these efforts can be seen in Figure 3.5. As it is largely based on Figure 3.3 but with additional local variables, it features the same number of states and transitions. The additional local variables add some complexity to the chart but in doing so, fixes the numerous issues that were rampant within the older chart.

Having modelled the PlayStation Dynamic Menu, we can draw two conclusions from it. The first is that the user interface can be quite simple when looking past the inter-connectivity between its various features. This is shown best by Figure 3.2, where all branches are centred around the Home menu and all features are consolidated within these different branches. A casual user of the device is likely to care little for the various enthusiast features it offers, while being most interested in major features, such as video games, media applications, settings and power

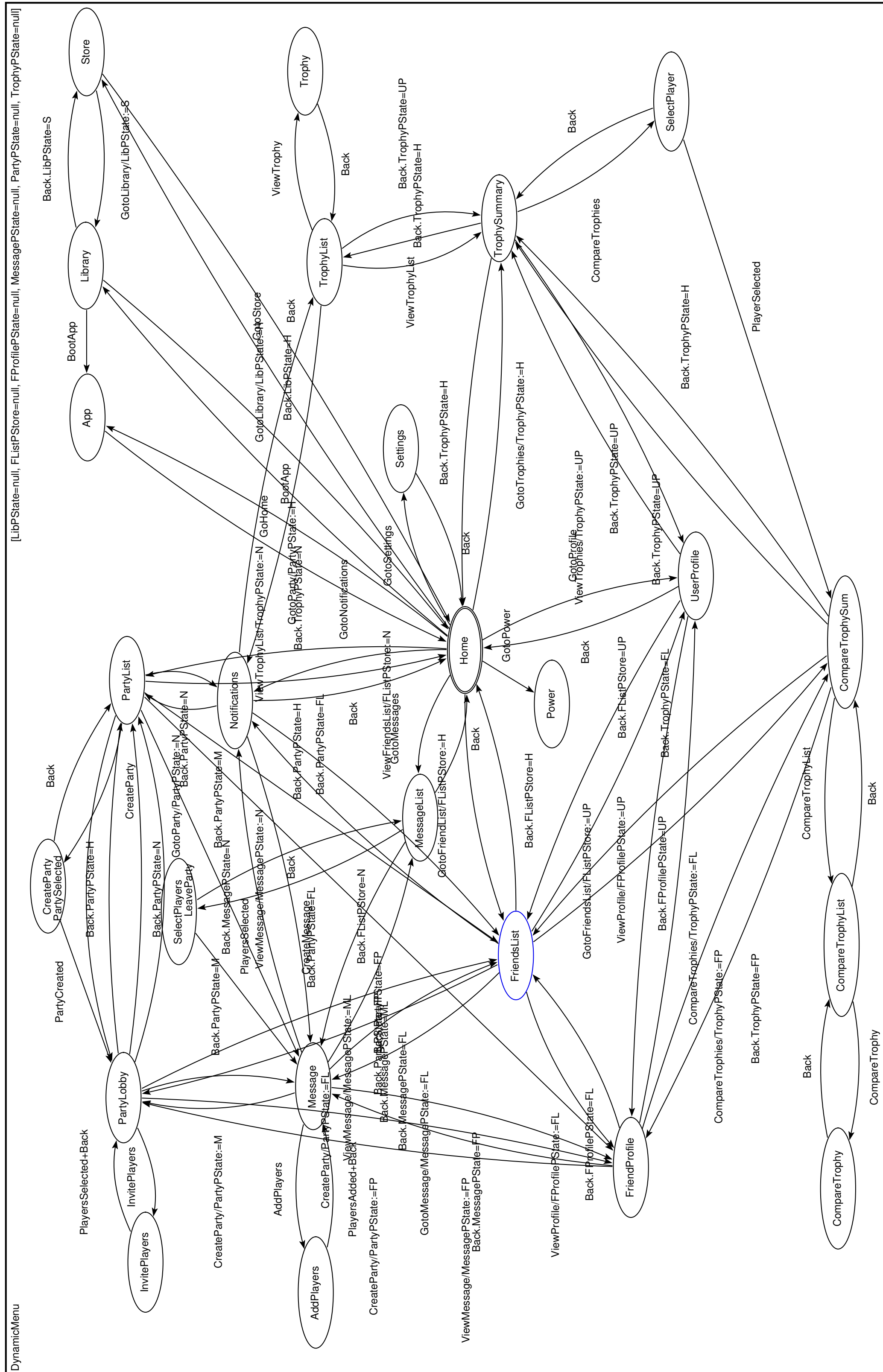


Figure 3.5: PS Dynamic Menu chart with multiple local variables

settings. As these major features are all immediately available from the *Home* menu, we believe it effectively caters to these users.

The second conclusion is that this user interface also appears very complex upon viewing these charts, particularly Figure 3.5, and is largely due to the inter-connectivity between its various different features. Upon analysis we found that this inter-connectivity, while over-complicating the model, did make sense in context. For example, when viewing a friend's profile or selecting a friend from a list, the user has the ability to interact with their friend in a variety of different ways, such as sending their friend a message and inviting their friend into a party. As such, we believe much of the inter-connectivity, and thus complexity, seen within the PlayStation Dynamic Menu were a result of the expectations their enthusiast audience would have of the PlayStation 4's feature set.

The combination of these two conclusions suggests to us that the designers of the PlayStation Dynamic Menu purposefully prioritised creating an interface that was simple on initial view but feature rich upon closer investigation. This is shown by the simple layout of the *Home* menu, with all major features immediately available from it and all other features grouped together into specific branches, such as Friends, Messages and Party. The complexity of the inter-connectivity between these branches was then a product of this design and the expectations of their enthusiast audience, as they then sought to provide more convenient ways for users to open other related branches directly from within a different branch if they chose to do so.

3.2 Return Home pattern

While modelling the PlayStation Dynamic Menu, we discovered a feature that we found added significant complexity to the model. The Dynamic Menu allows the user to return to the home screen with the press of the PS button on its controller,

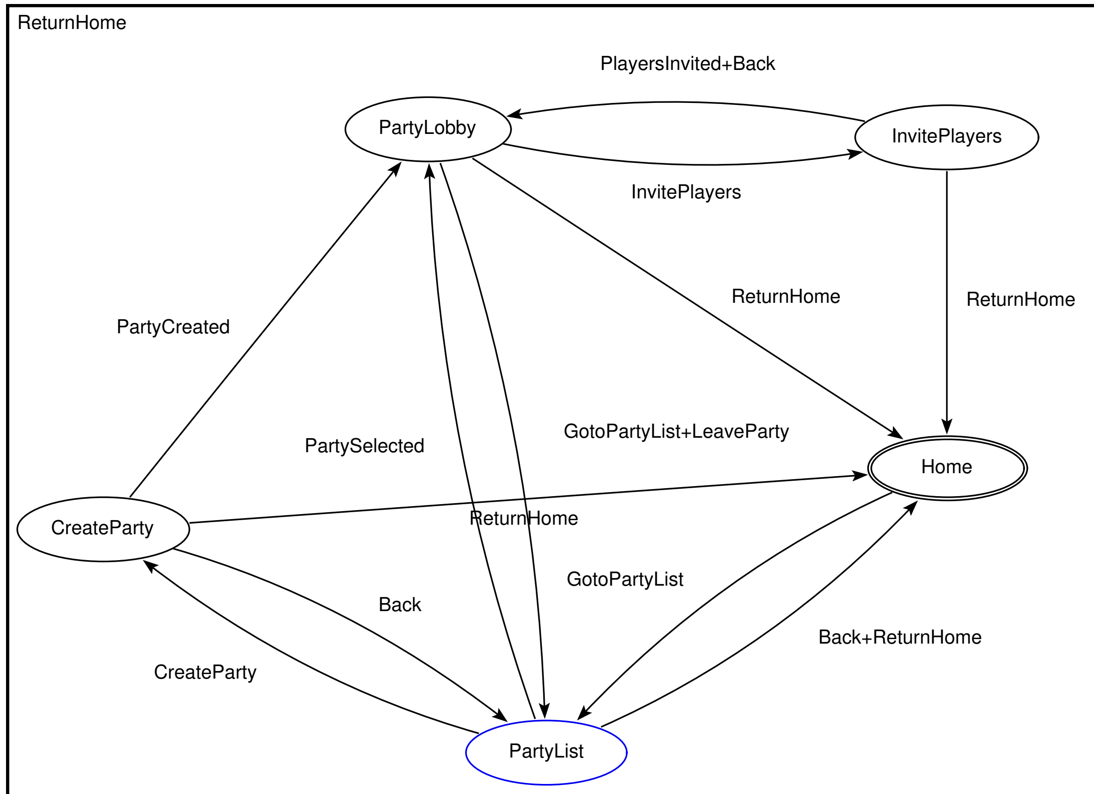


Figure 3.6: Party segment from PS Dynamic Menu chart with Return Home feature

similar to the show desktop shortcut in Windows or the Home button found on iOS and Android devices. We named this feature the Return Home function. Upon a second press of the PS button, the user will then be returned to their prior screen, which we named the Return Back function. Finally, the combination of the two functions was called the Return feature.

In order to model this feature correctly, the decision was made to first focus on the Return Home function and then later combine this with the Return Back function to capture the full behaviour of the feature.

Figure 3.6 shows an initial attempt at modelling the Return Home function using a small segment of the Dynamic Menu. The *Home* state is the starting state of the model and there are an additional four states representing screens within the Dynamic Menu, *PartyList*, *PartyLobby*, *CreateParty* and *InvitePlayers*. There

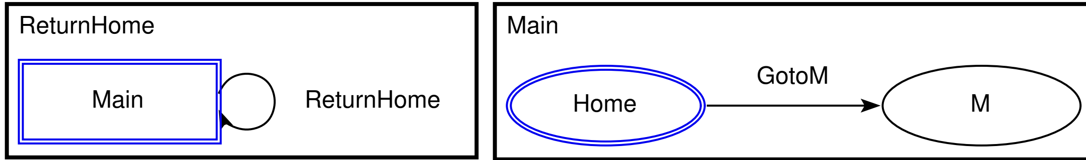


Figure 3.7: Simple Return Home pattern

are a number of transitions within this chart that are identical to the transitions within Figures 3.2, 3.3 and 3.4 but the transitions of note are the *ReturnHome* transitions. To model the Return Home function, one transition from each non-starting state to the starting state is required, though in the case of the transition from *PartyList* to *Home*, as there is already a transition between these two states, we can simply add another possible input to its guard.

The problem with this model was that there was a $y = x - 1$ relationship between the states and *ReturnHome* transitions, where y represents *ReturnHome* transitions and x represents states. Upon trying to use this with the full PlayStation Dynamic Menu chart seen in Figures 3.2, 3.3 and 3.4, which feature 33 states, 23 states and 35 states respectively, it quickly became very complex with the inclusion of additional transitions and/or guards from every non-starting state to the starting *Home* state. This is shown in Figure 3.8, with the Return Home function applied to the PlayStation Dynamic Menu. An additional 23 transitions have been added to the chart and another 10 transitions have had the *ReturnHome* input added to their guards. As a result of this, it was decided that we should find a more efficient way to represent this feature.

In order to find a concise design pattern that modelled the behaviour of the Return Home function, we needed a method that reset a chart to its starting state regardless of what state the current state was. Upon researching the logic and semantics of μ-Charts, we discovered a possible solution within the paper *The syntax and semantics of μ-Charts* [13]. This paper offered two different interpretations of how a decomposition may behave. In one behaviour, the child chart

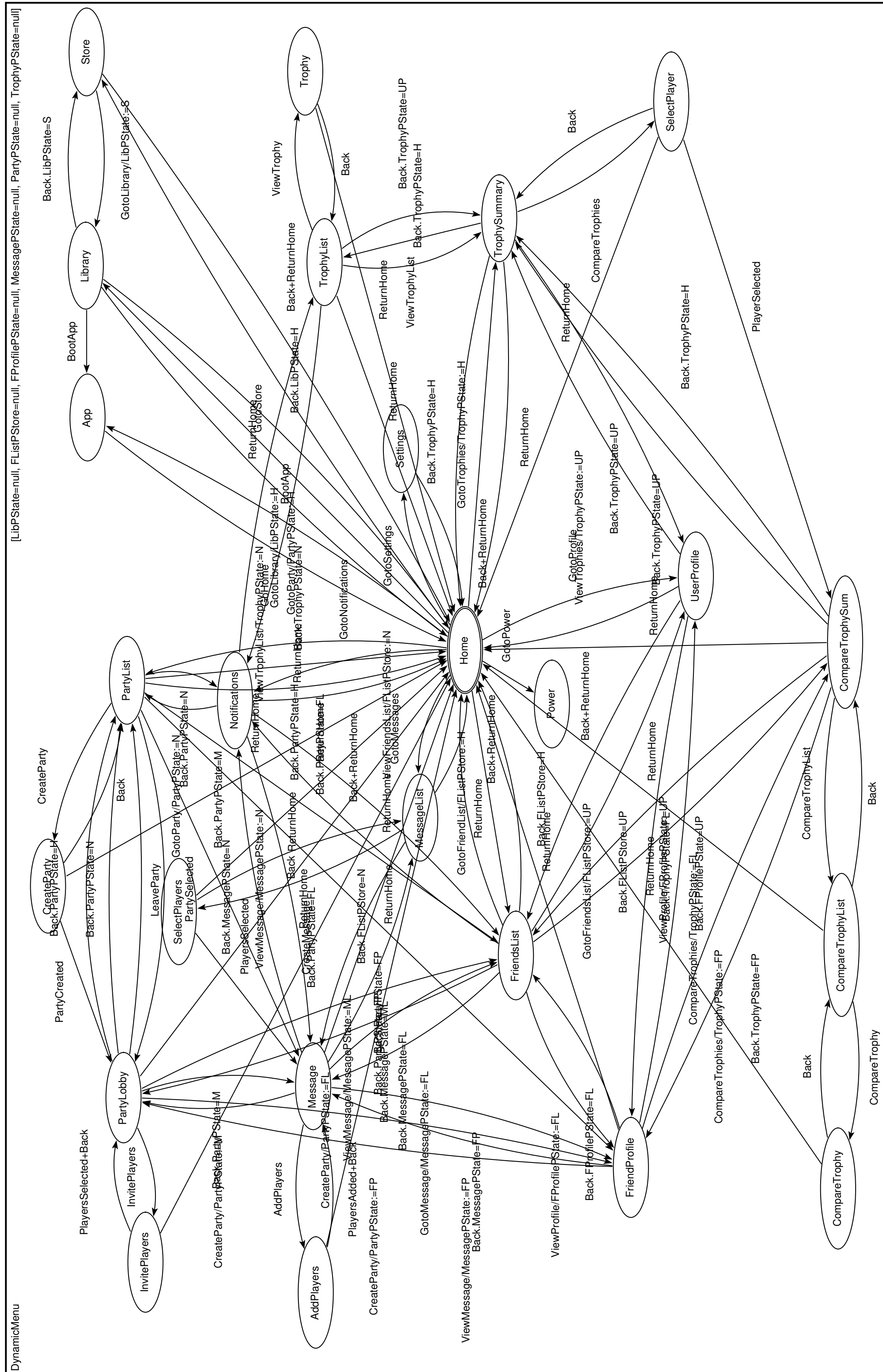


Figure 3.8: Full PS Dynamic Menu chart with Return Home function added

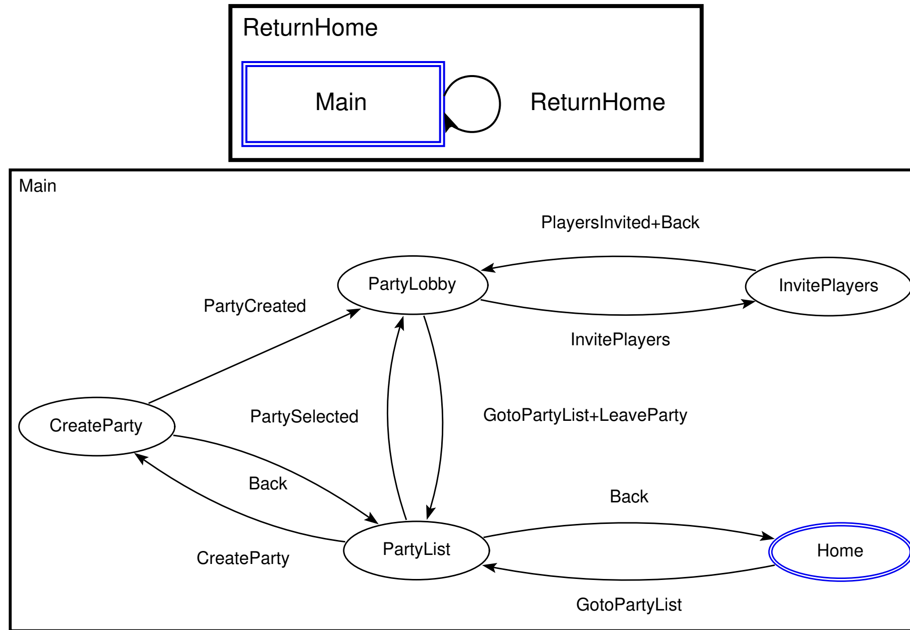


Figure 3.9: Return Home pattern applied to Party segment from PS Dynamic Menu chart

remembers and stays in the exact state it was in when it was last active in the parent chart. In the other behaviour, the child chart does not remember its state while it is inactive and as a result, is re-initialised every time it becomes active.

Using this interpretation, we created the Return Home pattern in its simplest form, shown in Figure 3.7. This chart contains a decomposition within it. Within the parent chart, *ReturnHome*, there is a single decomposed state with a single self-loop transition on it that receives the input *ReturnHome*. Within the child chart, *Main*, there are two states, the *Home* starting state, which represents the home screen of the system, while the state M represents any possible screens within the PlayStation Dynamic Menu. There is one transition between these two states, which receives the input *GotoM* in order to transition from the *Home* state to the state M.

This pattern works by assuming that all non-Return Home interactions within the user interface will be made within the child chart, *Main*. If a *ReturnHome* input signal is received at any point, the self-loop on the parent chart's decomposed

state will occur, resetting the child chart to its starting state *Home* no matter where it was previously within the interface.

Seen in Figure 3.9 is the Return Home pattern applied to the earlier example. The pattern has resulted in the removal of the three *ReturnHome* transitions from states *PartyLobby*, *CreateParty* and *InvitePlayers* to the starting state *Home*, as well as the *ReturnHome* input signal from the guard of the transition from state *PartyList* to the *Home* state. Applied to a small chart like this, the difference is relatively small but it is easy to recognise how much easier it would make modelling the Return Home functionality on much larger charts, such as the ones seen in Figures 3.2, 3.3 and 3.4, where instead of five states, there are over 20 or 30 states.

Finally in Figure 3.10 we see the Return Home pattern applied to the PlayStation Dynamic Menu. While the chart is still very complex, it is significantly less complex compared to Figure 3.8 as it no longer requires every state within the *Main* chart to transition to the *Home* state in order to model the Return Home function. As the chart was already very complex before the Return Home function was added to the model, this helps us a lot, as it means we do not need to add further complexity to the chart, aside from the addition of the decomposition. The Return Home function is there but modelled more abstractly, providing us a clearer look at the design of the Dynamic Menu interface.

3.3 Return pattern

Having modelled the Return Home function, we could then move on to model the Return Back function and then combine these two into a full model of the Return feature.

Figure 3.11 shows an initial attempt at modelling the Return Back function. Using the same segment of the Dynamic Menu seen in Figure 3.6, we see that the

Figure 3.10: Return Home pattern applied to full PS Dynamic Menu chart

Home state is the starting state and there are an additional four states within the chart, the *PartyList*, *PartyLobby*, *CreateParty* and *InvitePlayers* states. The majority of transitions also resemble Figure 3.6 but there are two key differences.

The first difference is that there are an additional three transitions, each from the states *PartyLobby*, *CreateParty* and *InvitePlayers* states to the starting state *Home*, as well as an additional guard within the transition from the *Home* state to the *PartyList* state. This is because with the Return Back function, we now need to be able to transition back to the previous state after a *ReturnHome* input signal has caused the chart to transition back to the *Home* state. This means that for every state that is added to the chart, an additional transition must also be added, alongside the *ReturnHome* transitions that were mentioned in the Return Home section. As a result of this, we go from having a $y = x - 1$ relationship in Figure 3.6, where x represents the number of states and y represents the number of transitions, to a $y = 2(x - 1)$ relationship. So for every extra state added to a chart featuring the Return Back function, two new transitions related to that function need to be added, adding a significant amount of complexity to it in the process.

The second difference is the use of the local variable *Return*, which can be seen initialised to the value *null* in the upper left corner of Figure 3.11. Similar to the *PrevState* local variable in the *Design Patterns* section, this variable is used to store an abbreviation representing a specific state within the chart, for example the state *PartyList* is represented by its abbreviation PLi.

This variable is used by the transitions which contain *ReturnHome* or *ReturnBack* within their guards. The *ReturnHome* transitions use the variable within the action to set it to the value that represents the previous state. For example, the transition labelled $(GotoHome + ReturnHome)/Return := PLi$ sets the *Return* variable to the value that represents the state *PartyList*, PLi, if the input signals *GotoHome* or *ReturnHome* are received. Note that the state *PartyList* will still

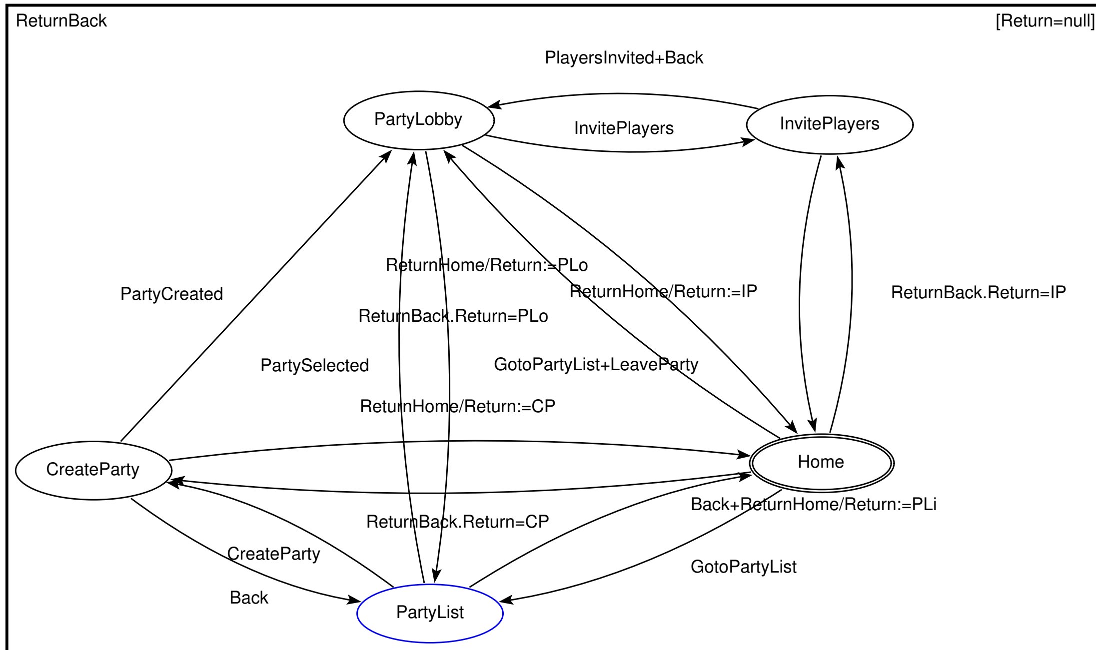


Figure 3.11: Party section from PS Dynamic Menu chart with both Return features

be recorded as the previous state even if the *GotoHome* input signal is received instead of a *ReturnHome* input signal. This behaviour is consistent with the behaviour of the PS Dynamic Menu. The *ReturnBack* transitions use the *Return* variable to specify which state the chart should return to if a *ReturnBack* input signal is received. For example, the transition labelled $ReturnBack.Return = PLo$ will only occur and transition to the state *PartyLobby* if both the *ReturnBack* input signal is received and the *Return* variable is set to the value *PLo*.

Having successfully modelled the Return Back function, our next goal was to combine both the Return Home and Return Back patterns in order to fully model the Return function and thus create the Return pattern. Figure 3.12 shows the resulting pattern.

Similar to the Return Home pattern shown in Figure 3.7, decomposition is used by the Return pattern in order to reset the chart to its starting state if a *ReturnHome* input signal is received. Additionally, it also resembles the Return

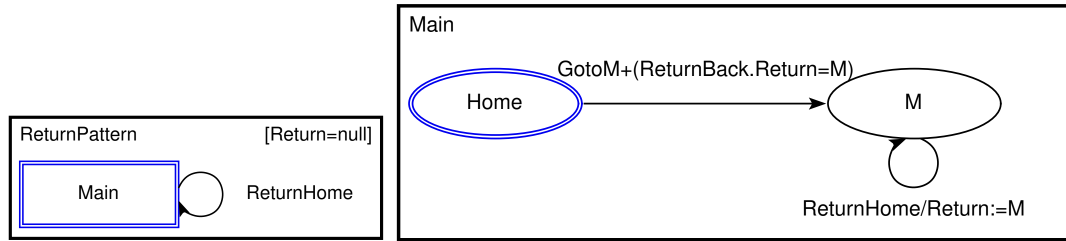


Figure 3.12: Simple Return pattern

chart shown in Figure 3.11, using local variables to store the previous state and then using these variables to determine which state the chart needs to return back to if a *ReturnBack* signal is received. A small difference seen within this chart is that the local variable is part of the parent chart. This is because our understanding of the relationship between the parent and child charts, defined in [11], gave the child chart access to the parent chart's local variables.

The most significant difference between the two charts is the self-loop transitions on every non-starting state within the child chart. These self-loop transitions occur when a *ReturnHome* input signal has been received and then within the action, sets the local variable to the numerical value that represents it.

Due to the way μ-Charts work, where a set of input signals will appear from the environment and all resulting actions occur simultaneously, we have the ability to make more than one transition within multiple charts occur simultaneously. So long as the resulting actions can happen in the same step, in that they do not conflict with one another, any potential non-determinism is nullified. As a result, if state M is the current state of the child chart, *Main*, and a *ReturnHome* input signal is received, both the *ReturnHome* transition within the parent chart, *ReturnBack*, and the $ReturnHome/Return := M$ self-loop transition within the child chart, *Main*, will occur. This will both set the *Return* local variable to the value M, representing state M, while also resetting the child chart *Main* to its starting state.

The requirement to have self-loop *ReturnHome* transitions on every non-

starting state and a *ReturnBack* transition from the starting state to every non-starting state means that the Return pattern does not reduce the total number of transitions within the chart, in fact it may increase that number. What it does do is reduce the number of transitions between the starting state and non-starting states, potentially halving it, significantly reducing the clutter and complexity that can occur with too many transitions within a chart.

Unfortunately, it was at this point we discovered an error in our understanding of how local variables work between parent and child charts. Due to an example used within one of our reference materials [11], we were under the impression we could use a parent chart's local variables within the child chart. However, upon further analysis of this material and the semantics of decompositions, we discovered that what led us to this assumption was instead a typographical error and that use of local variables was actually intended to be limited solely to its local chart. We believe this was done in order to help simplify the semantics, as giving both charts use of a local variable may have led to potential conflicts. For example, a transition within the parent chart and a transition within the child chart may have tried to set a variable at the same time. This created a significant issue with our Return pattern, forcing us to completely rethink it.

Due to this issue, we needed a new way to retain the previous state while the child chart was re-initialised. In an effort to find a solution, we then explored the use of the feedback operator and value carrying signals in order to send and then receive the previous state to and from the parent chart.

One feature of value carrying signals that stood out to us was the way we could use them with local variables. If we want to assign the value carried by a signal to a local variable or the reverse, we simply use an action such as $lvar := val$. This assigns the value of the value carrying signal val to the local variable $lvar$. Assigning the value of a local variable to a value carrying signal was exactly the same, using the action $val := lvar$. This feature led to the creation of Figure 3.13.

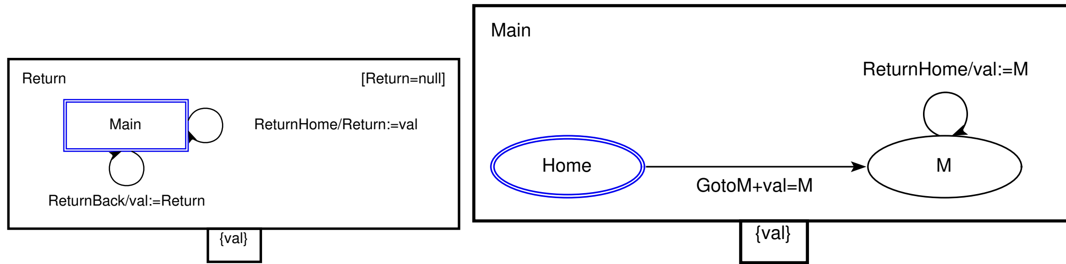


Figure 3.13: Revised Return pattern using feedback and value carrying signals

This chart greatly resembles the previous Return pattern but with the inclusion of value carrying signals and feedback. Both charts have the value carrying signal, *val*, applied as feedback, allowing them both to send the signal to the other. We use *val* to store the current state at the point the child chart is re-initialised, send it to the parent chart and store that value within the parent chart's *Return* local variable. Then upon a *ReturnBack* signal being received, we send this value back using *val* and the child chart receives the signal $val = M$ and transitions back to its previous state.

This chart had a major problem, however, in that it would re-initialise the child chart when the transition labelled $ReturnBack/val := Return$ occurred. So instead of transitioning back to its previous state, it would instead re-initialise again. With this we realised that due to the changes we had made with our interpretation, it was not possible to send signals from the parent chart back to the child chart without re-initialising it.

At this point, it became obvious that there were no possible solutions without some significant changes to μ-Charts or at least our interpretation of μ-Charts. In an effort to solve this issue, we came up with three possible solutions.

The first solution was to modify μ-Charts, as well as its semantics, to support the child chart having access to its parent chart's local variables. This change would have supported our initial Return pattern but it would have required a significant amount of work to figure out how to ensure there would be no conflicts

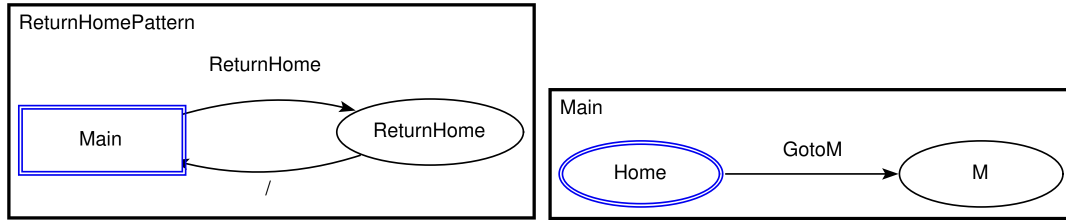


Figure 3.14: Final Return Home pattern

within the semantics and potentially would have required significant modifications. As such, we believed this solution was not one that should be explored, as we wanted to add features to μ-Charts that had been previously detailed, not change it to suit whatever our needs were at the time.

Our second solution was to change the semantics so that the local variables are not re-initialised to their initial values within the child chart. This change would have allowed us to store and retain the previous state within the child chart, avoiding the problems related to sending and receiving the previous state to and from the parent chart. While this would have been a simple solution, we did not believe it was appropriate, as it would essentially be changing the re-initialisation of the child chart into a partial re-initialisation solely to make our jobs easier.

Our final solution was to rethink how we determined when a re-initialisation was meant to occur. Our previous interpretation was that whenever the parent chart transitioned out of the decomposed state, even if it were a loop transition, the child chart would be re-initialised. However, if we were to instead focus on whether the decomposed state was the current state, we would find that a loop transition would not re-initialise the child chart, as the decomposed state would remain the current state after it occurred. The only time the child chart would be re-initialised was when the parent chart transitioned from the decomposed state to a completely different state. It was this interpretation that first led to a revision of the Return Home pattern.

Within Figure 3.14 we see our revised Return Home pattern, which has been

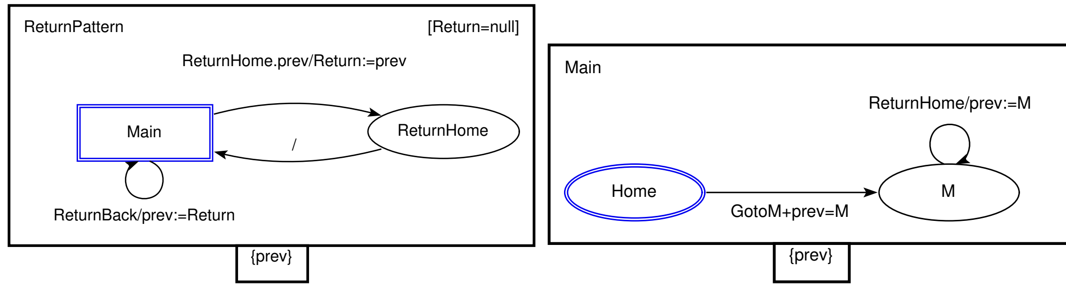


Figure 3.15: Final Return pattern

adapted to the new interpretation. This chart works almost identically to the previous Return Home pattern but instead of immediately re-initialising the child chart within a single step using a loop transition, we now re-initialise it by transitioning to the *ReturnHome* state when the *ReturnHome* signal is received and then transitioning back. As the transition back to the decomposed state lacks a guard, it is considered TRUE and will immediately occur in the next step, regardless of what input signals are received.

While the additional step within this Return Home pattern was more inconvenient than our initial Return Home pattern, we believed this change was necessary, as the Return pattern would clearly not work otherwise and we did not want to move forward while juggling two different interpretations.

Using this interpretation, we then proceeded to revise the the Return pattern, which can be seen in Figure 3.15. This combined the revised Return Home pattern with the use of the feedback signals and value carrying signals from Figure 3.13 to create a Return pattern that we believed accurately modelled the behaviour of the Return feature. The use of value carrying signals and feedback allowed us to send the child chart's previous state to the parent chart, which would save this state within a local variable. The child chart would then be re-initialised, as the parent chart transitioned to the *ReturnHome* state and then back to the decomposed state in the next step. At this point, if the *ReturnBack* signal were received, our new interpretation would allow us to send the previous state back

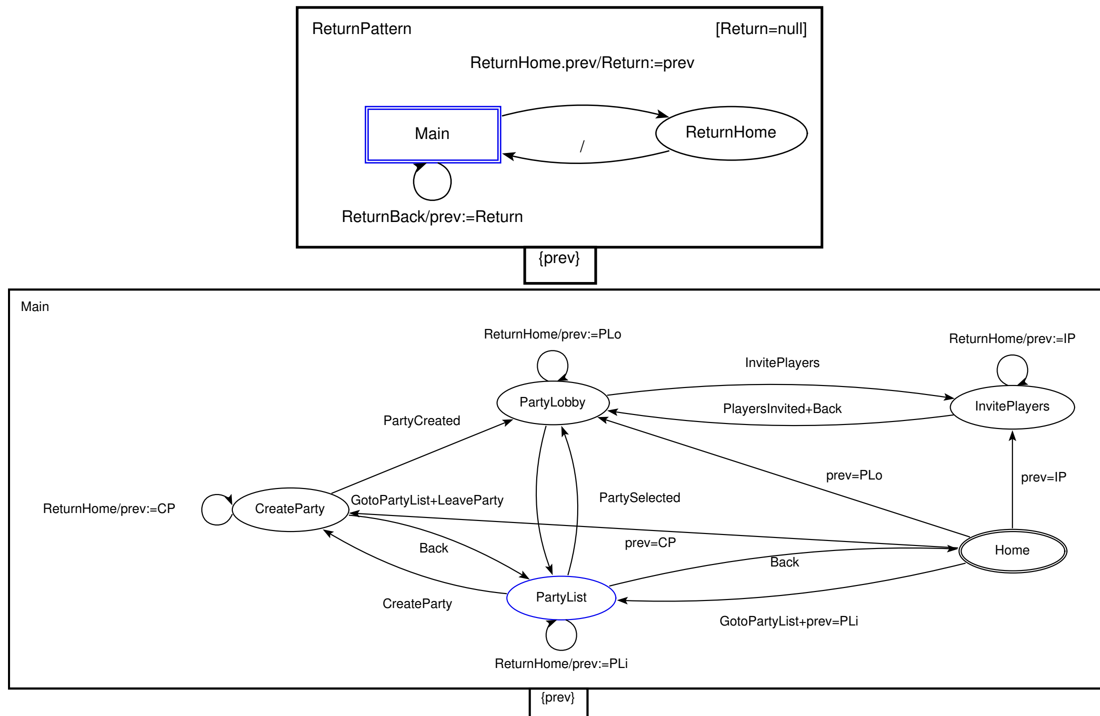


Figure 3.16: Final Return pattern applied to Party segment of PS Dynamic Menu chart

to the child chart within a value carrying signal without re-initialising the child chart. The child chart would then receive this signal and then transition to its previous state based on the value the signal carried.

Having produced these revised patterns, we believed that this solution was the appropriate move to make and decided to move forward with them.

In Figure 3.16 we see the revised Return pattern applied to the Party segment of the Dynamic Menu. We previously observed that there were 15 transitions when modelling the Return feature with this segment in Figure 3.11. Here we see that the Return pattern features 16 transitions, one more than the previous example, but it also reduces the visual complexity from the centre of the chart by removing three transitions to the *Home* state. These transitions are instead modelled as self-loop transitions on each of the states they start from, moving this visual complexity to the side of the chart, instead of the centre as it was originally.

As an example of this, we can see in Figure 3.11 that there are two transitions, labelled $ReturnHome/Return := CP$ and $ReturnBack.Return = CP$, that go to and from the *Home* state and *CreateParty* state. These two transitions then both cross over the two transitions between the states *PartyLobby* and *PartyList*. This creates more complexity within the centre of the chart, making it more difficult to recognise the structure of it upon first glance. In Figure 3.16, the transition labelled $ReturnHome/prev := CP$ is instead a self-loop transition going to and from the *CreateParty* state. This works in conjunction with the transition labelled *ReturnHome* within the parent chart, ensuring that it is logically identical to the equivalent transition within Figure 3.11 but is instead located to the left hand side of the chart, removing that complexity from the centre of the chart.

Chapter 4

Testing

In this chapter we discuss the testing of the Return Home and Return patterns, including the translation of the Return Home μ-Chart into Z using the ZooM tool and the testing of the Return Home Z schemas using the tool ProZ.

4.1 ZooM Modifications

Having created the Return Home and Return patterns, we then needed to test whether they behaved in the manner we believed they should. As the semantics of μ-Charts are given in Z, this meant that the best way to test both patterns was to translate them into Z and then test both Z specifications. For the first step of this plan, we decided to use the ZooM application.

ZooM is an application created by the University of Waikato to convert and translate μ-Charts into Z based on the semantics defined within [13]. It was developed in Haskell in conjunction with the AMuZed tool, using a Tcl/Tk interface named TclHaskell to create its graphical user interface. Unfortunately, as TclHaskell is no longer supported and has become incompatible with modern systems, ZooM was initially only accessible to us via a virtual machine that had been set up specifically to run both AMuZed and ZooM. This is an inconvenient solu-

tion, as it could only be used on certain computers at the University of Waikato and could not be modified to suit different purposes.

Our first task was to modify ZooM, removing the user interface and replacing it with a command line interface, while trying not to make too many unnecessary changes in an effort to reduce the risk of inadvertently causing unexpected behaviour.

Additionally, as both AMuZed and ZooM were developed in conjunction with one another, the source code of both tools was integrated together into a number of source files. So an additional goal was to remove any source files that were irrelevant to ZooM.

The ZooM user interface was simple, limited to open file and error dialogue boxes, so on face value it appeared it would be simple to strip the graphical user interface code from its source code. However, as AMuZed and ZooM's source code were integrated with each other, numerous functions were included and source files were referenced that ZooM never used. This led to a slow process of separating the source files that were imported into ZooM or a ZooM related source file from the ones that were not and then further filtering these files down by removing source files that were only used by functions that were not relevant to ZooM. This reduced the total number of source files from 15 files to nine files.

Once this was accomplished, we then moved onto removing TclHaskell from the source code. As most of the TclHaskell functions that were used within the ZooM source code were inherited from Haskell's own IO monad [8], replacing those function calls involved replacing them with the IO equivalent. For the TclHaskell functions that were not inherited by the IO monad, removing them involved trying to trace the function that called them back to ZooM. If those functions did not trace back to ZooM, the TclHaskell functions were replaced by placeholders, which would display an error message in the possible event that they were called at some point. However, if those functions did trace back to ZooM, we would have then

needed to rewrite ZooM's code so that it did not need to call said TclHaskell functions.

Due to ZooM's limited user interface, almost all of the TclHaskell function calls within its source code fell under the first two cases. They could be replaced with the IO monad's equivalent or a placeholder. The third case only occurred twice within its source code and were both located within the same source file, which dealt with building the open file and error dialogue boxes. Both functions were well coded and documented, so rewriting them to instead run from and print to the command line was then a simple task.

4.2 μ-Charts Semantics Modifications

Our next task with ZooM was to modify it so that it parsed and then correctly translated the additional interpretations of μ-Charts we were using into Z. Our first step towards this task was to figure out how these interpretations would be translated to Z.

4.2.1 Translating decomposition re-initialisation

$Iactive_{Child}$

$\Xi Chart_{Child}$

$active : \mathbb{P}\, \mu_{State}$

$o_{Child}! : \mathbb{P}\, outputI_{Child}$

$\neg\ active(Child)$

$o_{Child}! = \{\}$

Our first solution was a simple change to the child chart's inactive operation schema, seen above. As has been detailed previously, this operation schema occurs when the child chart is considered inactive. As such, using this schema to re-

initialise the child chart would have been a very simple and eloquent adjustment. We removed the declaration $\Xi Chart_{Child}$, as it ensured no state changes would be made within this schema. Replacing it would be the declaration $Init_{Child}$, which would declare that any observations and local variables within the state space of the child chart would be set to the initial values defined within the $Init_B$ schema. This schema can be seen below.

$$
\begin{array}{l}
\textit{Iactive}_{\textit{Child}} \\
\hline
\textit{Init}_{\textit{Child}} \\
\textit{active} : \mathbb{P}\, \mu_{\textit{State}} \\
o_{\textit{Child}}! : \mathbb{P}\, \textit{outputI}_{\textit{Child}} \\
\hline
\neg\ \textit{active}(\textit{Child}) \\
o_{\textit{Child}}! = \{\}
\end{array}
$$

Unfortunately, the relationship between the parent and child charts defined within the ParentChildDec section of the μ-Chart Z semantics prevented this solution from solving our problem. The issue is the predicate below from the $\delta_{ParentChildDec}$ schema:

$$((ParentChild \vee ParentChild') \wedge active(Parent)) \Leftrightarrow active(Child)$$

This line of predicate meant that the *Child* chart would be active so long as the decomposed state *ParentChild* was either the current state or the next state of the *Parent* chart. Due to this, if a loop transition were to occur on the decomposed state, such as the one used in the initial Return Home pattern, the *Child* chart would continue to stay active. This is because the decomposed state would be both the current and next state of the loop transition, making the left hand side of the if and only if operator true and thus the right hand side, which states that the child chart is active to also be true.

In addition to this, the *Child* chart would also stay active if the *Parent* chart

transitioned from the decomposed state to another state and then directly back to the decomposed state in the next step, as it does in our revised Return Home and Return patterns. This is because the decomposed state would be the current state while transitioning to the other state and then it would be the next state while transitioning back, so at no point would the left hand side of the predicate be false.

In order to make this predicate compatible with Return Home and Return patterns, we explored modifying it so that the child chart would only be active when the decomposed state was the current state. If the decomposed state was only the next state, the child chart would be considered inactive. This was a very small change, as seen below:

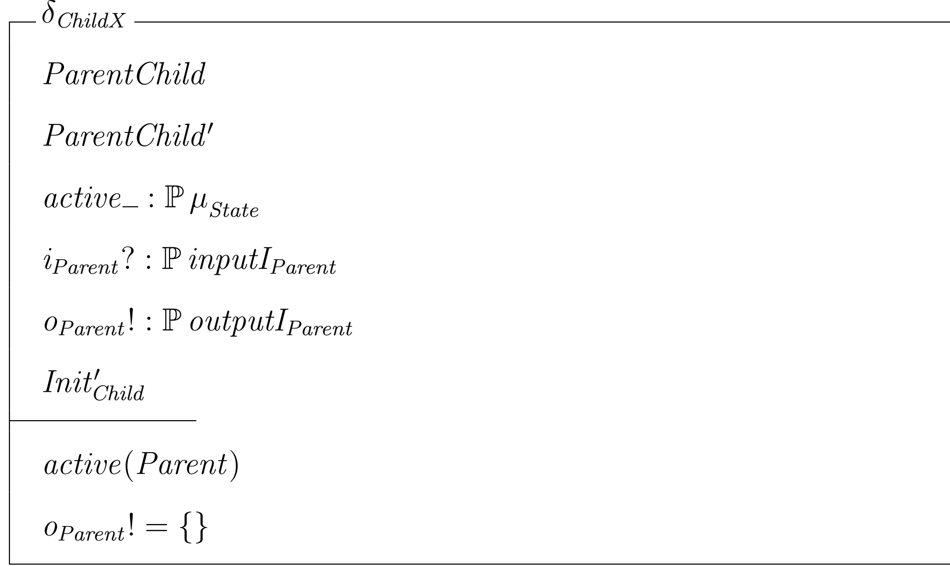
$$(\mathit{ParentChild} \wedge \mathit{active}(\mathit{Parent})) \Leftrightarrow \mathit{active}(\mathit{Child})$$

Without having tested it, we thought this modification accurately translated the behaviour we expected from our interpretation and as such, would have accurately translated our patterns. However, while we explored this idea, we came to the conclusion that this modification was making too significant a change to the semantics of μ-Charts. The predicate had been originally designed that way for a reason and we did not want to change this solely to suit our purposes. For this reason, we decided not to move forward with this solution.

Our second solution to translating re-initialisations in Z was significantly more complex. It involved re-initialising the child chart from within the parent chart's transition operation schemas. We then needed to ensure that this did not conflict with other actions within the child chart.

While we knew that we needed to re-initialise the child chart from within the parent chart's transition operation schemas, which of these schemas we would do this in was a decision we needed to make. Specifically, we needed to decide whether we would re-initialise the child chart when we transitioned out of the decomposed state or whether we would re-initialise it when we transitioned into

the decomposed state. We did not need to make this decision when first exploring the re-initialisation interpretation because the end result of both variants was the same in terms of input/output behaviour but on a semantics level, it was an important distinction to make. As a result, we decided to re-initialise the child chart when transitioning out of the decomposed state, as we believed this would better visually represent the state of the system when the child chart was inactive.



δ_{ChildX}

$ParentChild$

$ParentChild'$

$active_ : \mathbb{P}\, \mu_{State}$

$i_{Parent}? : \mathbb{P}\, inputI_{Parent}$

$o_{Parent}! : \mathbb{P}\, outputI_{Parent}$

$Init'_{Child}$

$active(Parent)$

$o_{Parent}! = \{\}$

Using this work, we created the *Parent* chart's transition schema from the decomposed state *Child* to state X shown above. As this operation transitions out of the decomposed state, we want it to re-initialise the *Child* chart. As such, the $Init'_{Child}$ schema has been included as a declaration. This inclusion means that when this transition occurs, the next state of the *Child* chart's state space observations will be re-initialised to the values defined in the $Init_{Child}$ schema.

$$\delta_{Child} \mathrel{\widehat{=}} \delta_{HomeM} \lor Iactive_{Child} \lor \epsilon_{Child}$$

$$\delta_{ChildNC} \mathrel{\widehat{=}} (\delta_{HomeM} \lor Iactive_{Child} \lor \epsilon_{Child}) \setminus (c_{Child})$$

Above we see the *Child* chart's main operation schema, δ_{Child}, and new operation schema, $\delta_{ChildNC}$. This new schema is needed because the relationship defined

between the *Parent* and *Child* charts requires both main operation schemas to occur at the same time. For this reason, if the re-initialisation occurs within the *Parent* chart, this will conflict with any state changes within the *Child* chart's operations.

To solve this issue, we create the $\delta_{ChildNC}$ operation schema. The additional NC within its name stands for non-conflicting, as this schema has been created to ensure that no conflict occurs when the *Child* chart is re-initialised. It does this by using the hide operator, \, to exclude any changes within its preceding operation schemas to the state space observations that follow it. In this case, any change to the current state within the δ_{XY}, $Iactive_{Child}$ or ϵ_{Child} is ignored. If there were more observations within the *Child* chart state space, we would also list those alongside the current state. This operation schema is then used in place of δ_{Child} only when a re-initialisation occurs.

$$
\begin{array}{l}
\delta_{ParentChildDec} \\
\hline
\Delta Chart_{ParentChildDec} \\
i_{ParentChildDec}? : \mathbb{P}\, inputI_{ParentChildDec} \\
active_ : \mathbb{P}\, \mu_{State} \\
o_{ParentChildDec}! : \mathbb{P}\, outputI_{ParentChildDec} \\
\hline
((ParentChild \lor ParentChild') \land active(Parent)) \Leftrightarrow active(Child) \\
\exists\, i_{Parent}?, i_{Child}?, o_{Parent}!, o_{Child}! : \mathbb{P}\, Signal \bullet \\
i_{Parent}? = i_{ParentChildDec}? \cap inputI_{Parent} \land \\
i_{Child}? = i_{ParentChildDec}? \cap inputI_{Child} \land \\
o_{ParentChildDec}! = o_{Parent}! \cup o_{Child}! \land \\
((\delta_{ChildX} \land \delta_{ChildNC}) \lor (\neg\, \delta_{ChildX} \land \delta_{Parent} \land \delta_{Child}))
\end{array}
$$

The final change we make is shown above. The point of interest is the final line within the predicate, which is where we determine whether we use the δ_{Child} or $\delta_{ChildNC}$ operation schemas. This predicate is a disjunction at its top-level. On

the right hand side, the main operation schemas of both charts are included like they normally would but only if the δ_{ChildX} transition operation does not occur. If it does occur, then the left hand side will occur instead, including the $\delta_{ChildNC}$ operation, which will ensure there are no conflicts with δ_{ChildX}'s re-initialisation of the *Child* chart.

If a chart features multiple transitions out of the decomposed state, then this predicate will look slightly different. Instead of the single transition operation δ_{ChildX}, there will be disjunctions of all possible transitions on both sides of the top-level disjunction within sets of brackets.

With these changes in mind, we made the changes to ZooM's code. As we did not want our changes to affect every decomposition that was translated by ZooM, we used the command line flag '-d' to ensure this interpretation was strictly optional.

4.2.2 AMuZed and ZooM bugs

Unfortunately, during our testing of AMuZed and ZooM, we encountered three bugs related to the use of local variables.

The first bug involved local variables within decompositions. If a user added a local variable to the parent chart in a decomposition, AMuZed would often not save this local variable within its resulting .muz file. This variable would not appear within ZooM's Z translation and any use of this variable would then be falsely translated as a value carrying signal.

The second bug we encountered involved local variables within the guard of transitions. If a local variable were used within the guard in disjunction with an input signal or another local variable, ZooM would then translate this relationship into a conjunction.

Finally, the third bug we encountered involved local variables and value carrying signals within the action of transitions and had two resulting translation

issues. If a local variable were set to the value stored within a value carrying signal, ZooM would fail to translate this action at all. Instead the semantics would make no mention of it

All three bugs were significant issues, capable of dramatically changing the translation we expected. It became apparent that this functionality had not been properly tested when AMuZed and ZooM were initially created. As we did not know where in the code base these bugs occurred, we decided to manually fix these issues ourselves. The AMuZed bug could be easily fixed by manually editing its saved .muz file and ensuring the local variables were all saved within the file. The guard bug could be fixed by manually editing the Z translation and changing the erroneous conjunction back into a disjunction.

We also manually edited the Z translation to fix the action bug but it was slightly more involved, as it required a more complicated predicate. For this we needed to add predicate similar to the following:

$$(\exists x : \mathbb{Z} \bullet \mathit{Ssig}\, x \in i_{\mathit{Parent}}? \cup (o_{\mathit{Parent}}! \cap \Psi_{\mathit{Parent}}) \wedge \mathit{Var}' = x)$$

This states that an integer x exists such that it is the signal *Sval*'s signal and has been received as input. We then set the next state of the local variable *Var* to this value.

4.3 ProZ

With the Return Home and Return patterns translated into their Z semantics, we then needed to test that they behaved the way we expected. For this, we turned to ProZ, a plugin for the verification tool ProB, which allows us to simulate, test and verify Z specifications.

4.3.1 ProZ formatting issues

Upon importing our translated Z semantics into ProZ, we were first met with numerous formatting errors. These included errors such as ProZ not supporting specifications that are spread across multiple files, LaTeX formatting tags ProZ conflicted with and operations that ProZ did not recognise. As a result of this, we created the following list of changes that needed to be applied to all ZooM translated Z specifications in order for them to be recognised by ProZ.

- Combine all .tex specification files into a single file
- Remove all lines starting with \Label
- Rewrite all instances of $active(chart)$ to $chart \in active$
- Delete all instances of _following *active* declaration
- Change top level System initialisation schema name to *Init*
- Expand and make explicit any value carrying signals within the feedback definition

While most of these points are simple formatting errors, the final point is slightly more complicated. When a value carrying signal is included within a feedback signal set, ZooM translates it in a format similar to the below.

$$\Psi_{Parent} = \{Ssig\}$$

While not incorrect, as the value carrying signal *Ssig* would have been previously defined, ProZ does not have the capability to know that these are related. For this reason, we need to instead expand and make explicit any value carrying signals within the feedback signal set, such as:

$$\Psi_{Parent} = \{n : \mathbb{Z} \bullet Ssig\ n\}$$

4.3.2 ProZ memory issues

Having resolved the formatting issues, we then found that the size and scope of our Z specifications, particularly decompositions, would often lead to ProZ running out of memory while simulating them. In an effort to prevent this from occurring, we then worked towards reducing the scope of the Z specifications by simplifying some of the semantics.

The first change we made to our Z specifications was to remove the types and observations that were not used within the system. As ZooM translates μ-Charts into a generic form of its Z semantics, it will often include all expected types and observations, even if they are not used within the system. As an example of this, the Return Home pattern does not produce any output but the output types and observations still appear within its semantics. As it is a waste of resources for ProZ to simulate these, we safely remove them and all inclusions of them while making sure not to affect the predicate.

One of the most memory intensive parts of the Z semantics we identified was the power sets. ProZ simulates these by exploring every possible subset and as such, systems that contain a large number of power sets can be very memory intensive. In order to simplify them, we identified two methods we could use.

The first was to split the μ_{State} type into two separate types: Charts and States. As the μ_{State} type included both chart and state elements but the observations of power set μ_{State} type were all limited to either the state elements or chart elements, we could safely split this type into two types without causing any issues. This significantly reduced the cardinality of each type and thus reduced the memory intensiveness of their power sets.

The second method we identified was to replace the power set declarations with the appropriate explicit definitions. For example, the explicit definition of $\delta_{ParentChildDec}$'s $\mathbb{P}\,\mu_{State}$ type is $\{\{\}, \{Parent\}, \{Child\}, \{Parent, Child\}\}$. This change significantly reduced the number of calculations made within the simula-

tion.

4.3.3 Testing the patterns

Having made these three changes, we then proceeded to test both the Return Home and Return patterns using ProZ. We found that both patterns behaved as intended. The Return Home pattern successfully re-initialised when the *ReturnHome* input signal was received, returning to the default *Home* state with no unintended side effects. The Return pattern had identical results when the *ReturnHome* input signal was received, while it also successfully returned to its previous state when the *ReturnBack* input signal was received following a re-initialisation.

Chapter 5

Conclusion and Future Work

In this chapter we provide an overview of the goals of our thesis, summarise our results and then provide directions for future work.

5.1 Overview of Project Goals

In this project, we have explored design patterns within graphical models, in an attempt to more elegantly model complex systems. We have then used this to explore μ-Charts and their semantics, which are defined in Z. Lastly, we evaluated the tools, AMuZed, ZooM and ProZ, which were used to create μ-Charts, translate μ-Charts into their Z semantics and test Z specifications, respectively.

5.2 Summary of Results

We started this project with a focus on design patterns within graphical models and worked towards this goal, modelling two interesting features we named the Return Home and Return Back functions. This resulted in the Return Home and Return patterns.

During this process, we found ourselves increasingly studying μ-Charts and its Z semantics, which greatly informed us throughout our project and became a

much larger part of it than previously expected. Not only did it help us create the aforementioned patterns, particularly by offering a different interpretation to how decomposed charts operate than the one that was usually detailed, but it also helped us translate these patterns into their Z semantics and solve a number of issues that we encountered. One such issue was the direct result of a typographical error within one of our resources but due to the work we had done and our understanding of the semantics, we quickly found a viable solution. We also used this understanding to investigate what changes we needed to make to the semantics in order to translate our interpretation of how decomposed charts operate.

In our use of the tools, we modified the μ-Chart translation tool, ZooM, in order to make it more easily accessible from the command line and have the ability to translate decompositions using the interpretation we used. We tested ZooM in order to ensure the changes we made had no unintended effect on the resulting translations. While we did not discover any errors that were caused by our changes, we did uncover a number of bugs related to the translation of local variables that existed prior to our work.

Finally, we created a number of steps that can be used in order to put ZooM translations into a format that ProZ would accept. We additionally provided a number of steps that could be used to reduce the memory imprint of the μ-Charts semantics within ProZ should anyone encounter them.

5.3 Future Work

While this thesis only explored two patterns, in future we believe there should be a continuation of building a library of design patterns. This library could then be used to help model systems and homogenise the way different people may model a system.

During our work, we used the tool, AMuZed. As AMuZed is reliant on the

Tcl/Tk interface, TclHaskell, which was abandoned and is no longer compatible with modern operating systems, it can only be run from a virtual machine that has been specifically created to operate it. While there is a new version of AMuZed that has been re-engineered with the goal to replace it, it is not in a state to do so. As such, we believe more work should be done in order to instead update the original version of AMuZed. In our research, we came across another Tcl/Tk interface for Haskell named HTk [6], which we believe could be used to replace TclHaskell.

We also believe more work should be made to test and fix the tool, ZooM. We encountered and detailed a number of bugs and translation issues during our research and due to this, we do not believe it has been tested extensively. For this reason, we believe that if it is to be continued to be used, it needs to be further tested, starting with the tests we have compiled.

Bibliography

[1] Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (August 1996)

[2] Bowen, J., Reeves, S.: Design patterns for models of interactive systems. In: Software Engineering Conference (ASWEC), 2015 24th Australasian. pp. 223–232. IEEE (2015)

[3] Bowen, J.P.: Formal specification and documentation using Z: A case study approach, vol. 66. International Thomson Computer Press London (1996)

[4] Harel, D.: Statecharts: A visual formalism for complex systems. Science of computer programming 8(3), 231–274 (1987)

[5] Johnson, R., Gamma, E., Helm, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Boston, Massachusetts: Addison-Wesley (1995)

[6] Lüth, C.: A short introduction to HTk–graphical user interfaces for haskell (2002), http://www.informatik.uni-bremen.de/htk/intro/intro.ps.gz

[7] Nazareth, D., Regensburger, F., Scholz, P.: Mini-statecharts: A lean version of statecharts. Mathematisches Institut und Institut für Informatik der Technischen Universität München (1996)

[8] O'Sullivan, B., Goerzen, J., Stewart, D.B.: Real world haskell: Code you can believe in. O'Reilly Media, Inc. (2008)

[9] Philipps, J., Scholz, P.: Compositional specification of embedded systems with statecharts. TAPSOFT'97: Theory and Practice of Software Development pp. 637–651 (1997)

[10] Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: Integrated formal methods. pp. 480–500. Springer (2007)

[11] Reeve, G., Reeves, S.: μ-charts and Z: Extending the translation. Tech. rep., Technical Report 00/11, Department of Computer Science, University of Waikato (2000)

[12] Reeve, G., Reeves, S.: μ-charts and Z: Hows, whys, and wherefores. In: Integrated Formal Methods. pp. 255–276. Springer (2000)

[13] Reeve, G., Reeves, S.: The syntax and semantics of μ-charts. Tech. rep., Technical Report 04/2004, Department of Computer Science, University of Waikato (2004)

[14] Reeve, G., Reeves, S.: Logic and refinement for charts. In: Proceedings of the 29th Australasian Computer Science Conference-Volume 48. pp. 13–23. Australian Computer Society, Inc. (2006)

[15] Spivey, J.M.: The fuzz manual. Computing Science Consultancy 34 (1992)

Appendix A

Initial Return Home Pattern Semantics

The Z semantics of the initial Return Home pattern displayed in Figure 3.7 are shown here, formatted to be compatible with ProZ.

$$\mu_{State} ::= ReturnHomePattern \mid Main \mid ReturnHomePatternMainDec \mid$$
$$ReturnHomePatternMain \mid MainHome \mid MainM$$

$$Signal ::= SGotoM \mid SReturnHome$$

$$states_{ReturnHomePattern} : \mathbb{P}\, \mu_{State}$$
$$inputI_{ReturnHomePattern} : \mathbb{P}\, Signal$$
$$states_{ReturnHomePattern} = \{ReturnHomePatternMain\}$$
$$inputI_{ReturnHomePattern} = \{SReturnHome\}$$

$$Chart_{ReturnHomePattern}$$
$$c_{ReturnHomePattern} : states_{ReturnHomePattern}$$

$$\textit{Init}_{\textit{ReturnHomePattern}}$$

$$\textit{Chart}_{\textit{ReturnHomePattern}}$$

$$c_{\textit{ReturnHomePattern}} = \textit{ReturnHomePatternMain}$$

$$\textit{ReturnHomePatternMain}$$

$$\textit{Chart}_{\textit{ReturnHomePattern}}$$

$$c_{\textit{ReturnHomePattern}} = \textit{ReturnHomePatternMain}$$

$$\delta_{\textit{MainMain}}$$

$$\textit{ReturnHomePatternMain}$$

$$\textit{ReturnHomePatternMain}'$$

$$i_{\textit{ReturnHomePattern}}? : \mathbb{P}\, \textit{Signal}$$

$$\textit{active} : \mathbb{P}\, \mu_{\textit{State}}$$

$$\textit{Init}'_{\textit{Main}}$$

$$\textit{ReturnHomePattern} \in \textit{active}$$

$$\textit{SReturnHome} \in i_{\textit{ReturnHomePattern}}?$$

$$\epsilon_{\textit{ReturnHomePattern}}$$

$$\Delta\, \textit{Chart}_{\textit{ReturnHomePattern}}$$

$$i_{\textit{ReturnHomePattern}}? : \mathbb{P}\, \textit{Signal}$$

$$\textit{active} : \mathbb{P}\, \mu_{\textit{State}}$$

$$\textit{ReturnHomePattern} \in \textit{active}$$

$$\neg\, (\textit{ReturnHomePatternMain} \land \textit{SReturnHome} \in i_{\textit{ReturnHomePattern}}?)$$

$$c'_{\textit{ReturnHomePattern}} = c_{\textit{ReturnHomePattern}}$$

$$\begin{array}{|l}\hline \textit{Iactive}_{ReturnHomePattern} \\ \Xi \textit{Chart}_{ReturnHomePattern} \\ active : \mathbb{P}\, \mu_{State} \\ \hline \neg\ ReturnHomePattern \in active \\ \hline \end{array}$$

$$\delta_{ReturnHomePattern} \mathrel{\widehat=} \delta_{MainMain} \lor Iactive_{ReturnHomePattern} \lor \epsilon_{ReturnHomePattern}$$

$$\begin{array}{|l} states_{Main} : \mathbb{P}\, \mu_{State} \\ inputI_{Main} : \mathbb{P}\, Signal \\ \hline states_{Main} = \{MainHome, MainM\} \\ inputI_{Main} = \{SGotoM\} \end{array}$$

$$\begin{array}{|l}\hline Chart_{Main} \\ c_{Main} : states_{Main} \\ \hline \end{array}$$

$$\begin{array}{|l}\hline Init_{Main} \\ Chart_{Main} \\ \hline c_{Main} = MainHome \\ \hline \end{array}$$

$$\begin{array}{|l}\hline MainHome \\ Chart_{Main} \\ \hline c_{Main} = MainHome \\ \hline \end{array}$$

$$\begin{array}{|l}\hline MainM \\ Chart_{Main} \\ \hline c_{Main} = MainM \\ \hline \end{array}$$

δ_{HomeM}

$MainHome$

$MainM'$

$i_{Main}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$Main \in active$

$SGotoM \in i_{Main}?$

ϵ_{Main}

$\Delta Chart_{Main}$

$i_{Main}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$Main \in active$

$\neg\, (MainHome \land SGotoM \in i_{Main}?)$

$c'_{Main} = c_{Main}$

$Iactive_{Main}$

$\Xi Chart_{Main}$

$active : \mathbb{P}\, \mu_{State}$

$\neg\, Main \in active$

$$\delta_{Main} \mathrel{\widehat=} \delta_{HomeM} \lor Iactive_{Main} \lor \epsilon_{Main}$$

$$\delta_{MainNC} \mathrel{\widehat=} (\delta_{HomeM} \lor Iactive_{Main} \lor \epsilon_{Main}) \setminus (c_{Main})$$

$$states_{ReturnHomePatternMainDec} : \mathbb{P}\,\mu_{State}$$

$$inputI_{ReturnHomePatternMainDec} : \mathbb{P}\,Signal$$

$$states_{ReturnHomePatternMainDec} = states_{ReturnHomePattern} \cup states_{Main}$$

$$inputI_{ReturnHomePatternMainDec} = inputI_{ReturnHomePattern} \cup inputI_{Main}$$

$Chart_{ReturnHomePatternMainDec}$

$$Chart_{ReturnHomePattern}$$

$$Chart_{Main}$$

$Init_{ReturnHomePatternMainDec}$

$$Init_{ReturnHomePattern}$$

$$Init_{Main}$$

$\delta_{ReturnHomePatternMainDec}$

$$\Delta Chart_{ReturnHomePatternMainDec}$$

$$i_{ReturnHomePatternMainDec}? : \mathbb{P}\,inputI_{ReturnHomePatternMainDec}$$

$$active : \mathbb{P}\,\mu_{State}$$

$$((ReturnHomePatternMain \lor ReturnHomePatternMain') \land$$

$$\quad ReturnHomePattern \in active) \Leftrightarrow Main \in active$$

$$\exists\, i_{ReturnHomePattern}?, i_{Main}? : \mathbb{P}\,Signal \bullet$$

$$i_{ReturnHomePattern}? = i_{ReturnHomePatternMainDec}?$$

$$\quad \cap inputI_{ReturnHomePattern} \land$$

$$i_{Main}? = i_{ReturnHomePatternMainDec}? \cap inputI_{Main} \land$$

$$((\delta_{MainMain} \land \delta_{MainNC}) \lor (\neg\, \delta_{MainMain} \land \delta_{ReturnHomePattern} \land \delta_{Main}))$$

$Init$

$$Init_{ReturnHomePatternMainDec}$$

$$
\begin{array}{l}
\textit{ReturnHomePatternSys} \\
\hline
\Delta Chart_{ReturnHomePatternMainDec} \\
i_{ReturnHomePatternMainDec}? : \mathbb{P}\, inputI_{ReturnHomePatternMainDec} \\
\hline
\exists\, active : \mathbb{P}\, \mu_{State} \bullet \\
\quad ReturnHomePattern \in active \land \delta_{ReturnHomePatternMainDec} \\
\hline
\end{array}
$$

Appendix B

Final Return Home Pattern Semantics

The Z semantics of the final Return Home pattern displayed in Figure 3.14 are shown here, formatted to be compatible with ProZ.

$$\mu_{State} ::= ReturnHomePattern \mid Main \mid ReturnHomePatternMainDec \mid ReturnHomePatternMain \mid ReturnHomePatternReturnHome \mid MainHome \mid MainM$$

$$Signal ::= SGotoM \mid SReturnHome$$

$$states_{ReturnHomePattern} : \mathbb{P}\,\mu_{State}$$

$$inputI_{ReturnHomePattern} : \mathbb{P}\,Signal$$

$$states_{ReturnHomePattern} = \{ReturnHomePatternMain, ReturnHomePatternReturnHome\}$$

$$inputI_{ReturnHomePattern} = \{SReturnHome\}$$

$Chart_{ReturnHomePattern}$

$$c_{ReturnHomePattern} : states_{ReturnHomePattern}$$

$$Init_{ReturnHomePattern}$$

$$Chart_{ReturnHomePattern}$$

$$c_{ReturnHomePattern} = ReturnHomePatternMain$$

$$ReturnHomePatternMain$$

$$Chart_{ReturnHomePattern}$$

$$c_{ReturnHomePattern} = ReturnHomePatternMain$$

$$ReturnHomePatternReturnHome$$

$$Chart_{ReturnHomePattern}$$

$$c_{ReturnHomePattern} = ReturnHomePatternReturnHome$$

$$\delta_{MainReturnHome}$$

$$ReturnHomePatternMain$$

$$ReturnHomePatternReturnHome'$$

$$i_{ReturnHomePattern}? : \mathbb{P}\, Signal$$

$$active : \mathbb{P}\, \mu_{State}$$

$$Init'_{Main}$$

$$ReturnHomePattern \in active$$

$$SReturnHome \in i_{ReturnHomePattern}?$$

$\delta_{ReturnHomeMain}$

$ReturnHomePatternReturnHome$

$ReturnHomePatternMain'$

$i_{ReturnHomePattern}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$ReturnHomePattern \in active$

$\epsilon_{ReturnHomePattern}$

$\Delta Chart_{ReturnHomePattern}$

$i_{ReturnHomePattern}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$ReturnHomePattern \in active$

$\neg\, (ReturnHomePatternMain \land SReturnHome \in i_{ReturnHomePattern}?)$

$c'_{ReturnHomePattern} = c_{ReturnHomePattern}$

$Iactive_{ReturnHomePattern}$

$\Xi Chart_{ReturnHomePattern}$

$active : \mathbb{P}\, \mu_{State}$

$\neg\, ReturnHomePattern \in active$

$$\delta_{ReturnHomePattern} \mathrel{\widehat=} \delta_{MainReturnHome} \lor \delta_{ReturnHomeMain} \lor Iactive_{ReturnHomePattern} \lor \epsilon_{ReturnHomePattern}$$

$states_{Main} : \mathbb{P}\, \mu_{State}$

$inputI_{Main} : \mathbb{P}\, Signal$

$states_{Main} = \{MainHome, MainM\}$

$inputI_{Main} = \{SGotoM\}$

$Chart_{Main}$

$c_{Main} : states_{Main}$

$Init_{Main}$

$Chart_{Main}$

$c_{Main} = MainHome$

$MainHome$

$Chart_{Main}$

$c_{Main} = MainHome$

$MainM$

$Chart_{Main}$

$c_{Main} = MainM$

δ_{HomeM}

$MainHome$

$MainM'$

$i_{Main}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$Main \in active$

$SGotoM \in i_{Main}?$

ϵ_{Main}

$$\Delta Chart_{Main}$$
$$i_{Main}? : \mathbb{P}\, Signal$$
$$active : \mathbb{P}\, \mu_{State}$$

$$Main \in active$$
$$\neg\,(MainHome \wedge SGotoM \in i_{Main}?)$$
$$c'_{Main} = c_{Main}$$

$Iactive_{Main}$

$$\Xi Chart_{Main}$$
$$active : \mathbb{P}\, \mu_{State}$$

$$\neg\; Main \in active$$

$$\delta_{Main} \mathrel{\widehat=} \delta_{HomeM} \\ \quad \vee\; Iactive_{Main} \vee \epsilon_{Main}$$

$$\delta_{MainNC} \mathrel{\widehat=} (\delta_{HomeM} \\ \quad \vee\; Iactive_{Main} \vee \epsilon_{Main}) \setminus (c_{Main})$$

$$states_{ReturnHomePatternMainDec} : \mathbb{P}\, \mu_{State}$$
$$inputI_{ReturnHomePatternMainDec} : \mathbb{P}\, Signal$$

$$states_{ReturnHomePatternMainDec} = states_{ReturnHomePattern} \cup states_{Main}$$
$$inputI_{ReturnHomePatternMainDec} = inputI_{ReturnHomePattern} \cup inputI_{Main}$$

$Chart_{ReturnHomePatternMainDec}$

$$Chart_{ReturnHomePattern}$$
$$Chart_{Main}$$

$Init_{ReturnHomePatternMainDec}$

$$Init_{ReturnHomePattern}$$

$$Init_{Main}$$

$\delta_{ReturnHomePatternMainDec}$

$$\Delta Chart_{ReturnHomePatternMainDec}$$

$$i_{ReturnHomePatternMainDec}? : \mathbb{P}\, inputI_{ReturnHomePatternMainDec}$$

$$active : \mathbb{P}\, \mu_{State}$$

$$((ReturnHomePatternMain \lor ReturnHomePatternMain') \land$$

$$\quad ReturnHomePattern \in active) \Leftrightarrow Main \in active$$

$$\exists\, i_{ReturnHomePattern}?, i_{Main}? : \mathbb{P}\, Signal \bullet$$

$$i_{ReturnHomePattern}? = i_{ReturnHomePatternMainDec}?$$

$$\quad \cap inputI_{ReturnHomePattern} \land$$

$$i_{Main}? = i_{ReturnHomePatternMainDec}? \cap inputI_{Main} \land$$

$$((\delta_{MainReturnHome} \land \delta_{MainNC})$$

$$\quad \lor (\neg\, \delta_{MainReturnHome} \land \delta_{ReturnHomePattern} \land \delta_{Main}))$$

$Init$

$$Init_{ReturnHomePatternMainDec}$$

$ReturnHomePatternSys$

$$\Delta Chart_{ReturnHomePatternMainDec}$$

$$i_{ReturnHomePatternMainDec}? : \mathbb{P}\, inputI_{ReturnHomePatternMainDec}$$

$$\exists\, active : \mathbb{P}\, \mu_{State} \bullet$$

$$\quad ReturnHomePattern \in active \land \delta_{ReturnHomePatternMainDec}$$

Appendix C

Return Pattern Semantics

The Z semantics of the final Return pattern displayed in Figure 3.15 are shown here, formatted to be compatible with ProZ.

$$\mu_{State} ::= ReturnPattern \mid Main \mid ReturnPatternMainDec \mid ReturnPatternMain \mid ReturnPatternReturnHome \mid MainHome \mid MainM$$

$$Signal ::= SGotoM \mid SReturnBack \mid SReturnHome \mid Sprev\langle\langle \mathbb{Z} \rangle\rangle$$

$$states_{ReturnPattern} : \mathbb{P}\,\mu_{State}$$

$$inputI_{ReturnPattern} : \mathbb{P}\,Signal$$

$$outputI_{ReturnPattern} : \mathbb{P}\,Signal$$

$$\Psi_{ReturnPattern} : \mathbb{P}\,Signal$$

$$states_{ReturnPattern} = \{ReturnPatternMain, ReturnPatternReturnHome\}$$

$$inputI_{ReturnPattern} = \{SReturnHome, SReturnBack\}$$

$$outputI_{ReturnPattern} = \{n : \mathbb{Z} \bullet Sprev\ n\}$$

$$\Psi_{ReturnPattern} = \{n : \mathbb{Z} \bullet Sprev\ n\}$$

$Variables_{ReturnPattern}$

$$VReturn : \mathbb{Z}$$

$Chart_{ReturnPattern}$

$$c_{ReturnPattern} : states_{ReturnPattern}$$

$$Variables_{ReturnPattern}$$

$Init_{ReturnPattern}$

$$Chart_{ReturnPattern}$$

$$c_{ReturnPattern} = ReturnPatternMain$$

$$VReturn = 0$$

$ReturnPatternMain$

$$Chart_{ReturnPattern}$$

$$c_{ReturnPattern} = ReturnPatternMain$$

$ReturnPatternReturnHome$

$$Chart_{ReturnPattern}$$

$$c_{ReturnPattern} = ReturnPatternReturnHome$$

$\delta_{MainReturnHome}$

$ReturnPatternMain$

$ReturnPatternReturnHome'$

$\Delta Variables_{ReturnPattern}$

$i_{ReturnPattern}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$o_{ReturnPattern}! : \mathbb{P}\, outputI_{ReturnPattern}$

$Init'_{Main}$

$ReturnPattern \in active$

$SReturnHome \in i_{ReturnPattern}? \cup (o_{ReturnPattern}! \cap \Psi_{ReturnPattern})$

$o_{ReturnPattern}! = \{\}$

$\delta_{ReturnHomeMain}$

$ReturnPatternReturnHome$

$ReturnPatternMain'$

$\Delta Variables_{ReturnPattern}$

$i_{ReturnPattern}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$o_{ReturnPattern}! : \mathbb{P}\, outputI_{ReturnPattern}$

$ReturnPattern \in active$

$o_{ReturnPattern}! = \{\}$

$VReturn' = VReturn$

$\delta_{MainMain}$

$ReturnPatternMain$

$ReturnPatternMain'$

$\Delta Variables_{ReturnPattern}$

$i_{ReturnPattern}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$o_{ReturnPattern}! : \mathbb{P}\, outputI_{ReturnPattern}$

$ReturnPattern \in active$

$SReturnBack \in i_{ReturnPattern}? \cup (o_{ReturnPattern}! \cap \Psi_{ReturnPattern})$

$o_{ReturnPattern}! = \{Sprev\ VReturn\}$

$VReturn' = VReturn$

$\epsilon_{ReturnPattern}$

$\Delta Chart_{ReturnPattern}$

$i_{ReturnPattern}?, o_{ReturnPattern}! : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$ReturnPattern \in active$

$\neg\, (ReturnPatternMain \land SReturnHome \in i_{ReturnPattern}?$
$\quad \cup (o_{ReturnPattern}! \cap \Psi_{ReturnPattern}))$

$\neg\, (ReturnPatternMain \land SReturnBack \in i_{ReturnPattern}?$
$\quad \cup (o_{ReturnPattern}! \cap \Psi_{ReturnPattern}))$

$c'_{ReturnPattern} = c_{ReturnPattern}$

$o_{ReturnPattern}! = \{\}$

$VReturn' = VReturn$

$Iactive_{ReturnPattern}$

$$\Xi Chart_{ReturnPattern}$$
$$active : \mathbb{P}\, \mu_{State}$$
$$o_{ReturnPattern}! : \mathbb{P}\, outputI_{ReturnPattern}$$

$$\neg\ ReturnPattern \in active$$
$$o_{ReturnPattern}! = \{\}$$
$$VReturn' = VReturn$$

$$\delta_{ReturnPattern} \mathrel{\widehat=} \delta_{MainReturnHome} \lor \delta_{ReturnHomeMain}$$
$$\lor\ \delta_{MainMain}$$
$$\lor\ Iactive_{ReturnPattern} \lor \epsilon_{ReturnPattern}$$

$$states_{Main} : \mathbb{P}\, \mu_{State}$$
$$inputI_{Main} : \mathbb{P}\, Signal$$
$$outputI_{Main} : \mathbb{P}\, Signal$$
$$\Psi_{Main} : \mathbb{P}\, Signal$$

$$states_{Main} = \{MainHome, MainM\}$$
$$inputI_{Main} = \{SGotoM, SReturnHome\} \cup \{n : \mathbb{Z} \bullet Sprev\ n\}$$
$$outputI_{Main} = \{n : \mathbb{Z} \bullet Sprev\ n\}$$
$$\Psi_{Main} = \{n : \mathbb{Z} \bullet Sprev\ n\}$$

$Chart_{Main}$

$$c_{Main} : states_{Main}$$

$Init_{Main}$

$$Chart_{Main}$$

$$c_{Main} = MainHome$$

$MainHome$

$Chart_{Main}$

$c_{Main} = MainHome$

$MainM$

$Chart_{Main}$

$c_{Main} = MainM$

δ_{HomeM}

$MainHome$

$MainM'$

$i_{Main}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$o_{Main}! : \mathbb{P}\, outputI_{Main}$

$Main \in active$

$(SGotoM \in i_{Main}? \cup (o_{Main}! \cap \Psi_{Main})$

$\vee\ Sprev\,1 \in i_{Main}? \cup (o_{Main}! \cap \Psi_{Main}))$

$o_{Main}! = \{\}$

δ_{MM}

$MainM$

$MainM'$

$i_{Main}? : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$o_{Main}! : \mathbb{P}\, outputI_{Main}$

$Main \in active$

$SReturnHome \in i_{Main}? \cup (o_{Main}! \cap \Psi_{Main})$

$o_{Main}! = \{Sprev\, 1\}$

ϵ_{Main}

$\Delta Chart_{Main}$

$i_{Main}?, o_{Main}! : \mathbb{P}\, Signal$

$active : \mathbb{P}\, \mu_{State}$

$Main \in active$

$\neg\, (\neg\, SGotoM \in i_{Main}? \cup (o_{Main}! \cap \Psi_{Main})$

$\wedge \neg\, (Sprev\, 1 \in i_{Main}? \cup (o_{Main}! \cap \Psi_{Main})))$

$\neg\, (MainM \wedge SReturnHome \in i_{Main}? \cup (o_{Main}! \cap \Psi_{Main}))$

$c'_{Main} = c_{Main}$

$o_{Main}! = \{\}$

$Iactive_{Main}$

$\Xi Chart_{Main}$

$active : \mathbb{P}\, \mu_{State}$

$o_{Main}! : \mathbb{P}\, outputI_{Main}$

$\neg\, Main \in active$

$o_{Main}! = \{\}$

$$\delta_{Main} \mathrel{\widehat=} \delta_{HomeM} \lor \delta_{MM}$$
$$\lor\ Iactive_{Main} \lor \epsilon_{Main}$$

$$\delta_{MainNC} \mathrel{\widehat=} (\delta_{HomeM}$$
$$\lor\ Iactive_{Main} \lor \epsilon_{Main}) \setminus (c_{Main})$$

$$states_{ReturnPatternMainDec} : \mathbb{P}\,\mu_{State}$$
$$inputI_{ReturnPatternMainDec} : \mathbb{P}\,Signal$$
$$outputI_{ReturnPatternMainDec} : \mathbb{P}\,Signal$$
$$\Psi_{ReturnPatternMainDec} : \mathbb{P}\,Signal$$

$$states_{ReturnPatternMainDec} = states_{ReturnPattern} \cup states_{Main}$$
$$inputI_{ReturnPatternMainDec} = inputI_{ReturnPattern} \cup inputI_{Main}$$
$$outputI_{ReturnPatternMainDec} = outputI_{ReturnPattern} \cup outputI_{Main}$$
$$\Psi_{ReturnPatternMainDec} = \{n : \mathbb{Z} \bullet Sprev\ n\}$$

$Chart_{ReturnPatternMainDec}$

$$Chart_{ReturnPattern}$$
$$Chart_{Main}$$

$Init_{ReturnPatternMainDec}$

$$Init_{ReturnPattern}$$
$$Init_{Main}$$

$\delta_{ReturnPatternMainDec}$

$$\Delta Chart_{ReturnPatternMainDec}$$
$$i_{ReturnPatternMainDec}? : \mathbb{P}\, inputI_{ReturnPatternMainDec}$$
$$active : \mathbb{P}\, \mu_{State}$$
$$o_{ReturnPatternMainDec}! : \mathbb{P}\, outputI_{ReturnPatternMainDec}$$

$$((ReturnPatternMain \lor ReturnPatternMain') \land$$
$$\quad ReturnPattern \in active) \Leftrightarrow Main \in active$$
$$\exists\, i_{ReturnPattern}?, i_{Main}?, o_{ReturnPattern}!, o_{Main}! : \mathbb{P}\, Signal \bullet$$
$$i_{ReturnPattern}? = (i_{ReturnPatternMainDec}? \cup (o_{ReturnPatternMainDec}!$$
$$\quad \cap \Psi_{ReturnPatternMainDec}))$$
$$\quad \cap inputI_{ReturnPattern} \land$$
$$i_{Main}? = (i_{ReturnPatternMainDec}? \cup (o_{ReturnPatternMainDec}!$$
$$\quad \cap \Psi_{ReturnPatternMainDec}))$$
$$\quad \cap inputI_{Main} \land$$
$$o_{ReturnPatternMainDec}! = o_{ReturnPattern}! \cup o_{Main}! \land$$
$$((\delta_{MainReturnHome} \land \delta_{MainNC})$$
$$\quad \lor (\lnot\, \delta_{MainReturnHome} \land \delta_{ReturnPattern} \land \delta_{Main}))$$

$Init$

$$Init_{ReturnPatternMainDec}$$

$ReturnPatternSys$

$$\Delta Chart_{ReturnPatternMainDec}$$
$$i_{ReturnPatternMainDec}? : \mathbb{P}\, inputI_{ReturnPatternMainDec}$$
$$o_{ReturnPatternMainDec}! : \mathbb{P}\, outputI_{ReturnPatternMainDec}$$

$$\exists\, active : \mathbb{P}\, \mu_{State} \bullet$$
$$\quad \delta_{ReturnPatternMainDec}$$