Design Patterns for Models of Interactive Systems

Judy Bowen
Department of Computer Science
University of Waikato
Hamilton, New Zealand
Email: jbowen@waikato.ac.nz

Steve Reeves
Department of Computer Science
University of Waikato
Hamilton, New Zealand
Email: stever@waikato.ac.nz

***Abstract*—Building models of safety-critical interactive systems (in healthcare, transport, avionics and finance, to name but a few) as part of the design process is essential. It is also advised for non-safety critical interactive systems if we want to be certain they will behave as intended in all circumstances. However, modelling interactive systems is also challenging. The levels of complexity in modern user interfaces and the wealth of interaction possibilities means that modelling at a suitable level of abstraction is crucial to ensure our models remain reasonably sized, readable, and therefore usable. The decisions we make about how to abstract the system to retain enough detail to be able to reason about it without running into known modelling problems (state-explosion, verbosity, unreadability) are complex, even for experienced modellers. We have identified a number of commonly seen problems in such models based on occurrences of common properties of interactive systems, and in order to help both experienced and novice modellers we propose model-patterns as a solution to this.**

I. Introduction

In software design and development, design patterns are reusable solutions that can be applied to commonly seen problems. Described comprehensively in [1], where more than twenty different patterns are explained and categorised, they are proposed as useful concepts to assist with code structuring (and, in the case of object-oriented programming, class structure and generation). They therefore provide a valuable resource both for teachers and learners since lessons for the unwary about pitfalls and how to spot them and avoid them, for example, can be built around them. But also they are important as they are abstractions, in that some details of a particular problem can be hidden in order to deal with complexity, and design and reasoning *etc.* can go on at a more abstract level. And as we all know, thanks to Paul Hudak, "abstraction, abstraction, abstraction" are the three most important ideas in programming [2] (and by extension computational systems design). What we abstract here is a pattern which is distinct from any particular context, and which can hence be used in many contexts.

Formal models of interactive systems are developed for a variety of reasons. They can be used prior to implementation as a way of verifying proposed behaviours of the intended systems to ensure they will behave correctly under all circumstances, or as the basis for generating tests and oracles which can then be used on the implemented system. Or they may be reverse-engineered from an existing system and used to support refactoring or again to ensure that the system is robust and correct. They may also be used to consider aspects of interaction, usability, functionality, flexibility *etc.* Our previous work on developing appropriate models for interactive systems has focussed on finding ways to combine models of the interface and interaction with models or specifications of functionality in order to be able to formally reason about interactive systems as a whole (and perform the sorts of tasks described above) [3], [4].

When we model interactive systems we need to remain at a high enough level of abstraction so that we are not overwhelmed by superfluous detail, but at the same time ensure we capture all of the crucial elements of the system's design so that we are subsequently able to reason about its correctness. The ability to correctly choose the amount of detail we need and the appropriate level of abstraction relies on both the skill of the modeller as well as the intended end-use of the model.

Our experience of modelling interactive systems, and teaching others to develop similar models, has shown that different people will model the same system differently. In some cases they will work at different levels of abstraction, or they may just view the problem in different ways. Over the years we have seen evidence of this as we have observed many people (both students and other researchers) generate different models to represent the same interactive systems.

Anecdotally we are aware that many people involved in modelling systems have their own patterns, which may also extend across teams that they are working within. One of our aims in this work is to make certain patterns explicit in the hope that a body of known patterns can be developed in a similar manner as for programming patterns. In this paper we describe two interactive systems we have been involved in modelling (one software and one interactive medical device) and show how these highlight particular design elements which we then propose modelling patterns for.

Whilst the sorts of patterns described in [1] and similar works are aimed at programmers and the programs they write, we believe that a similar approach can be taken for the creation of models of systems. That is, our patterns are aimed at modellers and the models they create. In this paper we identify several patterns which have been uncovered over the decade we have been modelling interactive systems. These patterns can be considered as guides to help the modeller with particular aspects of the model design. That is, they are not a solution in and of themselves but rather guide the modeller for particular parts of the system being modelled.

We have also observed that that there are particular patterns of interaction (in particular) that occur in the *systems* being modelled which are more likely to lead to differences (or

indeed errors) in the *model*. This has led us to identify corresponding patterns in the models themselves which enable us to describe a useful and consistent method of abstracting what are sometimes particularly complex (and we suggest sometimes overly complex) interactive systems. Similarly we have noted that the appearance of particular patterns in models may indicate problematic design elements in interactive systems.

We present a way of describing the patterns themselves in a formal language (something which is still under investigation). Leaving the patterns themselves informal, but giving lots of formal examples, is in some cases reasonable since humans are great at spotting patterns (it's their main skill). But having (possibly correctly) spotted a pattern, formal examples give a lead into how to actually use the pattern on some concrete problem, and the formality of the language can be used ultimately to check that the model, guided by the pattern, is in fact correct, as usual. Also formal patterns open the door to automatic (*i.e.* via algorithmic means) pattern recognition and also an algebra of patterns [5].

So, we start with the formal patterns in the expectation that from now on people will spot the sort of problem they are dealing with and simply instantiate our patterns with their target concrete modelling language (we give examples of this using both PIMs [3], [6] and emulink [7]). Thus the patterns we start with show how problems should be solved. We then go on to give reasons (via the concrete examples from which these patterns came) for why these patterns are as they are.

II. Organisation

We start in the next section by introducing our formal pattern language. We then give two examples of patterns, the callback and binary choice patterns, and show how we can describe them using this formalism. Next we give examples of these two patterns in real-world systems (our motivating examples) and introduce two modelling notations which can be used to create the types of models of interactive systems we are interested in. We then give examples of instantiating the patterns using these notations. Finally we introduce two further patterns, using a less formal description, as examples of other types of patterns that exist.

III. Formalising Patterns

Some of the standard ways in software engineering of writing down patterns (for example, as in the "Gang of Four" work [8]) might be suitable for many uses, but since our central concern is precision and correctness, we need something more.

We firstly need to be able to describe what the syntactic form of the thing being characterised by the pattern is, *i.e.* we need a grammar so that we can say "here is what a good solution looks like given your requirements" by giving a template which is concrete in some respects but abstract (*i.e.* hides details) in others, so that many different particular models are captured by the single description. This is what we mean by pattern, and it should become clear below exactly what this means.

So having captured all possible models (with some target concrete modelling language in mind) with a grammar, we need to write down constraints which say things like "all models in language X of this general form are within the pattern we have in mind" so that a pattern then is really a set of possible models, usually a subset, characterised by constraints in first-order logic, of all the models in the language generated (or recognised) by the grammar.

Happily, all this already exists, so the hard work has been done, *e.g.* [5], and we merely use this standard way of giving formal patterns.

First, a grammar for models. In the cases we are considering in this paper we are concerned with the interactivity of systems, and the general idea is that these will be modelled with *abstract state machines* of various sorts (concrete examples are given later as instantiations of our patterns). The basic grammar is given in Figure 1. We have simplified this by not saying what the structure (if any) of nodes are, we have used a placeholder non-terminal symbol which would lead to a further grammar which defines the syntactic structures (if any) of nodes. Later, as we will see, these might be complicated structures like presentation models, or they might simply be strings (*i.e.* a name or label) in the case of the charts associated with emu link models (we briefly describe these two modelling approaches later).

So, our machines have nodes, edges connecting nodes, labels for edges (consisting of guards and actions) and local variables (which allow information to be stored and accessed by any label, as we will see). To complete this account we do of course need grammar rules proceeding from the non-terminals Localvars, Guards and Actions, but we omit these for brevity. It might be helpful to regard our machines (and their grammars) as abstract syntax for the various modelling languages that we (concretely) build or models in.

Also note that these grammars are slightly different from the usual BNF in that we have added names to the various parts on the right-hand sides of productions (*e.g.* $nodes : Node^+$). These names allow us to put conditions in first-order logic on the structures that these grammars generate (or recognise). Finally, as is usual in BNF grammars, the $+$ superscript denotes a non-empty collection, a $*$ denotes a possibly empty collection and items in $[...]$ are optional.

We also need to introduce a shorthand predicate for a transition, given by an arrow, two nodes and a label (of the form "guard, action"), which is defined as follows:

$$\begin{aligned}\forall\, n0, n1 \in nodes \quad \bullet \\ & (n0 \xrightarrow{g,a} n1 \Leftrightarrow \\ & \exists\, e : edges \bullet from.e = n0 \wedge to.e = n1 \wedge \\ & g \in guard.e \wedge a \in action.e)\end{aligned}$$

That is, $n0 \xrightarrow{g,a} n1$ is an edge with source $n0$, target $n1$ and g is part (perhaps all of) its guard and a is part (perhaps all of) its action.

A. Callback Pattern

This is our first example of a pattern. Consider a system which has many different windows, and in each of these it is possible to choose to log-out of the system using a 'logout' feature. If a user chooses to 'logout' the system does not immediately perform this action, but rather provides the user

$$
\begin{array}{lll}
\mathit{Machine} & ::= & \mathit{nodes} : \mathit{Node}^{+}, \mathit{edges} : \mathit{Edge}^{*}, [\mathit{localvars} : \mathit{Localvars}^{*}] \\
\mathit{Edge} & ::= & \mathit{from} : \mathit{Node}, \mathit{to} : \mathit{Node}, [\mathit{guard} : \mathit{Guard}], \\
 & & [\mathit{action} : \mathit{Action}] \\
\mathit{Node} & ::= & x : X \\
X ::= & \ldots\ldots &
\end{array}
$$

Fig. 1. A part of the grammar for Machines

with a dialogue where they can choose either to cancel this action—in which case they will be returned to the window where they selected the 'logout' option—or continue to log-out, in which case they leave the system and are returned to the initial 'login' screen. Later we will see an example system with exactly this form and we will see how, without a pattern, the solution that a modeller develops might (quite reasonably) not be a good one.

More generally we see that most interactive systems provide ways for a user to navigate through a variety of different windows and dialogues which provide different aspects of the system's behaviours. Physical interactive devices (medical infusion pumps for example) provide different modes which have a similar effect. We need to be able to describe this navigation within our models in order to understand pertinent properties such as reachability, potential for deadlock *etc.* There are often particular windows or dialogues that exist which a user can access from any part of the system. The 'logout' example given above is a common example of this. In systems which require a user to log-in before use and which provides the ability to log-out at any point during use, it is usual (and indeed good practice) for a confirmation to be requested prior to the log-out occurring, so a typical interaction scenario is:

> User selects 'Log-out'
> Confirmation window is presented ("Are you sure you want to log-out?")
> User selects "ok" to continue with log-out or "cancel" to return to their previous activity

It is the ability to "cancel" the activity which causes the problem for the modeller, as it requires an explicit representation of the ability to access the log-out function for every state of the system. If this is abstracted to a single representation of the "log-out" feature then we lose the ability to understand (via the model) what happens when the user selects 'cancel'. That is, it is not enough to know that we have reached the log-out state we must also know where we were immediately prior in order to correctly model the cancel. However, explicitly modelling every state/log-out pair is unnecessarily verbose, and the callback pattern is presented as a more suitable solution.

We now illustrate our use of formal patterns with the Callback example. What we present is a pattern because it picks out a whole set of concrete Machines from the set of Machines defined by the grammar in Figure 1, not just one of them. The idea is that any Machine in the set picked out conforms to the pattern.

We follow [5]'s conventions, and add our own convention that identifiers that start with capital letters are to be taken as placeholders, essentially bound variables in the logical sense, and so, uniformly, they can stand for any concrete value.

Callback is any *Machine* (as defined by the grammar in Figure 1) which satisfies the following constraints. First, $\mathit{Callback} \in \mathit{Machine}$, where we understand that *Machine* is the set generated by the given grammar (in the usual way we think of a grammar as either generating a set of sentences, the language, or of recognising all and only the elements of the language).

Then, referring to Figure 2, the first section of the constraints introduces names for various parts of the Machines we are interested in. So, we first we have the parts or *components* of the pattern. But of course many other Machines (rather trivially) conform to this pattern too, because this at the moment covers *any* five node Machine with a local variable! So, we then put constraints on these parts so that exactly the right set of Machines is characterised for the Callback case. This is the role of the *conditions*.

The developer, when confronted with the problem of the callback kind, can look at this pattern and by providing concrete values for the components they can build a Machine of our suggested elegant form.

This text stands for a whole set of Machines once the parameters are made concrete. The important part, as ever with things containing parameters and with formal things in general, is the *shape* of the Machines that are picked out by the pattern. The actual names used for nodes, local variables *etc.* is of course not important, though the use of the same name for the same parameter (*e.g.* everywhere that *LV* appears in the pattern it must be replaced by the same actually local variable throughout) has to happen as usual since it is all part of the structure of the pattern we are trying to define. Recall the comment above about the components being like logical bound variables.

Also note that though we have given this example with an incomplete grammar for Machines, it is not much work to complete the grammar (*i.e.* adding the detail we have subsumed under "X" and its productions) tailored for some concrete target modelling language (*e.g.* as we shall see for the charts in PVSio-web or the PIMs in the PM/PIM models) and thus be able to interpret these patterns as abstractions of those sorts of model.

As ever with formalisation, we are expressing the *form* of the objects we are dealing with, that is the shape of the structure or the way various parts connect together.

B. Binary choice pattern

Another common occurrence in interactive systems is where a user invokes some behaviour (via a series of inter-

$Callback \in Machine$

Components
$Stop, Confirm?, Activity0, Activity1, Activity2 \in Node$
$LV \in Localvars$
$A0, A1, A2, OK, Cancel, Logout, Login \in Label$

Conditions
$Confirm? \xrightarrow{OK} Stop, Stop \xrightarrow{Login,} Activity0,$
$Activity0 \xrightarrow{Logout, LV:=A0} Confirm?, Confirm? \xrightarrow{Cancel.LV=A0,} Activity0,$
$Activity1 \xrightarrow{Logout, LV:=A1} Confirm?, Confirm? \xrightarrow{Cancel.LV=A1,} Activity1,$
$Activity2 \xrightarrow{Logout, LV:=A2} Confirm?, Confirm? \xrightarrow{Cancel.LV=A2,} Activity2 \in Edge$

Fig. 2. The formal pattern for Callback

action steps) but the outcome (the mode or part of the system they end up in) is dependent on the specific information they have entered. For example, in a login window a user might enter their username and password and click an "OK" button but the result of this is dependent on the values entered in the username and password text fields rather than their actions. They might end up being presented with an error dialogue informing them they have been unsuccessful, or they might end up in the main part of the system (indicating successful login) or they may stay in the login state.

What we are dealing with here are two different activities. The first is the actions of the user which are independent of the final result, the determination of which is a distinct second activity. For example they enter a username and a password and click an 'OK' button. If we model just their behaviour (which is independent of input values) the resulting model suggests non-determinism as we can see in the first model of figure 3. However, if we separated out the two distinct paths we need to artificially enhance the modelled UI behaviours (we say 'artificial' to mean that we change to level of abstraction purely to solve the modelling problem). So the second model in figure 3 has removed the non-determinism but has introduced separate user behaviours ('oklogin' and 'badlogin') which do not actually reflect how the user interacts with the system.

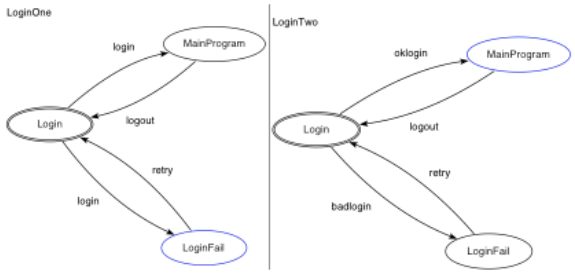

Fig. 3. Login model with non-determinism and artificial UI behaviour

What actually happens in such systems is that there is some underlying functional behaviour which determines which of the paths is taken, so what we need is a pattern which describes this ability to incorporate functional behaviours into the UI model to represent this.

Figure 4 shows the formal pattern for this, insofar as we can capture it formally. The reason for this proviso is that we also have to add a comment to say that "Correct" is a local variable *which is set by some underlying functionality*. The need for this proviso is in fact formally evident since nowhere in the pattern is "Correct" set (*cf.* Figure 2 and the "LV" which *is* set within the pattern).

C. Motivating Examples

Over the past ten years we have modelled a large number of interactive software systems using the presentation model approach described in the next section. We started with small proof-of-concept example systems as we developed our modelling theories and then progressed to real-world systems, and more recently to modal medical devices [9]. There is, of course, a huge variety in the nature of interactive systems in terms of number of windows and dialogues, modality, complexity of widgets and interaction approaches *etc.* When a number of different people with different levels of experience are involved in modelling the same system we see a large divergence in approach and in some cases uncertainty of how to create the most useful model (where this may depend on the level of abstraction required and the purpose of the model). Despite the diversity within different interactive systems we often observe similarities in their general design and the appearance of common features. Some of these can cause particular problems with modelling and in this paper we describe examples of these.

To provide useful examples for this paper we introduce a large-scale interactive software application as well as an interactive medical device we have been involved with modelling and provide examples of the particular problems which have formed the basis and motivation for the work described here (although we have, of course, seen these issues in other systems too). We describe these next and give brief details of some of the issues that arose during their modelling phase.

1) Gallagher Command Centre: The Gallagher Command Centre is a commercial software tool developed by the Gallagher Group Ltd.[1] used for building control and security to manage automation of doors, alarms *etc.*. There are two pieces of the software: one which enables users to monitor buildings

[1] www.gallagher.co

$$Binary \in Machine$$

$$\underline{Components}$$
$$Login, MainProg, LoginFail \in Node$$
$$Correct \in Localvars$$
$$LO, LI, Reset \in Label$$

$$\underline{Conditions}$$
$$Login \xrightarrow{LI.Correct=true,} MainProg, MainProg \xrightarrow{LO,} Login,$$
$$Login \xrightarrow{LI.Correct=false,} LoginFail, LoginFail \xrightarrow{Reset,} Login \in Edge$$

Fig. 4. The formal pattern for Binary Choice

and their set-ups (Command Centre Classic); and another where a super-user can control and change those settings and control user-access (Command Centre Premier). Both of these versions of the Command Centre have been (and are still being) modelled as part of ongoing work between ourselves and Gallagher.

The Command Centre software has a large number of windows that users can navigate to, as well as a number of settings which can be defined and changed which affect behaviours across multiple windows and widgets. There are a number of constant menus that sit at the top of all of the different windows whose behaviour differs slightly depending on which of the windows is being displayed.

Figure 5 shows one of the windows of the command centre software in the Alarm Panel Viewer mode. The major cause of complexity within the Command Centre model is the non-determinism of the Classic version. This is due to the fact that the Premier version can modify the behaviour of, and user access to, the Classic version while it is running. Our initial approach to modelling the Classic version, therefore, was to assume a 'super-user' (who had access to everything with no external changes being made) with the intention of then combining this with the model of Premier to produce a single model for both.

Figure 6 shows one of the intermediate PIMs for the Command Centre Classic model which gives some idea of both the complexity of the interactions as described above, and also the difficulty in creating a readable, visual model (even for just part of the system) because of that complexity.

Here we started using composition and value-carrying signals (both described in the next section) to try and simplify the descriptions of complex behaviour. It was these experiments which led to the simpler pattern we describe in the next section.

2) Bodyguard Pain Management Pump: The CME Bodyguard Pain Management Pump is an interactive modal medical device used to deliver pain medication to patients in a medical setting. We have modelled this device as part of our ongoing work with the Waikato Hospital on modelling medical devices and their contexts of use. The device delivers medication in pre-defined 'shots' (known as a bolus) rather than by continuous infusion. The interface consists of a number of soft-keys and a display along with a push-button which the user uses to initiate bolus delivery.

The software enables the medical practitioner to define parameters for the bolus delivery (size of bolus, volume permitted over a set period of time *etc.*) based on the medication being delivered.

Each of these systems presented particular modelling challenges and suggested that there were common themes (in these, and other systems we had encountered) which might be tackled in systematic and consistent ways.

IV. Instantiating the Patterns—Background

A. The Models

Our aim, motivated by the fact that we are currently working with safety-critical systems, is to present concrete examples in "full" languages, *i.e.* languages with not only formal syntax and semantics but logics too. Otherwise, how else are we to do proofs?

Thus our examples will be given using two different approaches (in order to show the concrete differences but also, hopefully, the abstract similarities). The first of these is the PVS approach (which includes the pvsio, pviso-web and emulink work from the CHI+MED group [10]–[12]) and the second is the presentation model approach (which uses Z, μCharts, presentation models, PIMs and PMR) [3], [9], [13]. Describing both of these approaches in full is beyond the scope of this paper (and best left to the relevant references), but we provide a brief overview here to support the readability of the rest of the paper.

1) PVS: Our use of PVS [14] in this paper is in the background as it provides the foundation for what is our real focus here, namely the charts which we use to model the interactive part of systems. Suffice to say that PVS provides us with a language (think functional with dependent types) and a theorem-prover/proof assistant to act as the "compiler" for programs written in the language (as with more modern languages like Agda [15] and Idris [16], type checking in such a strong type system needs theorem-proving).

2) PVSio: This is a PVS package that basically adds imperative features to the PVS language [17], thereby making more natural stateful things like side effects and input/out (hence its name). This is useful in that it allows a (perhaps) more natural setting in which to perform experiments with a PVS model, and also lends itself to being driven in an input-state-response way, which is useful when investigating models (rather than "merely" proving things about them). This then suggests the final part of the system we use: PVSio-web.

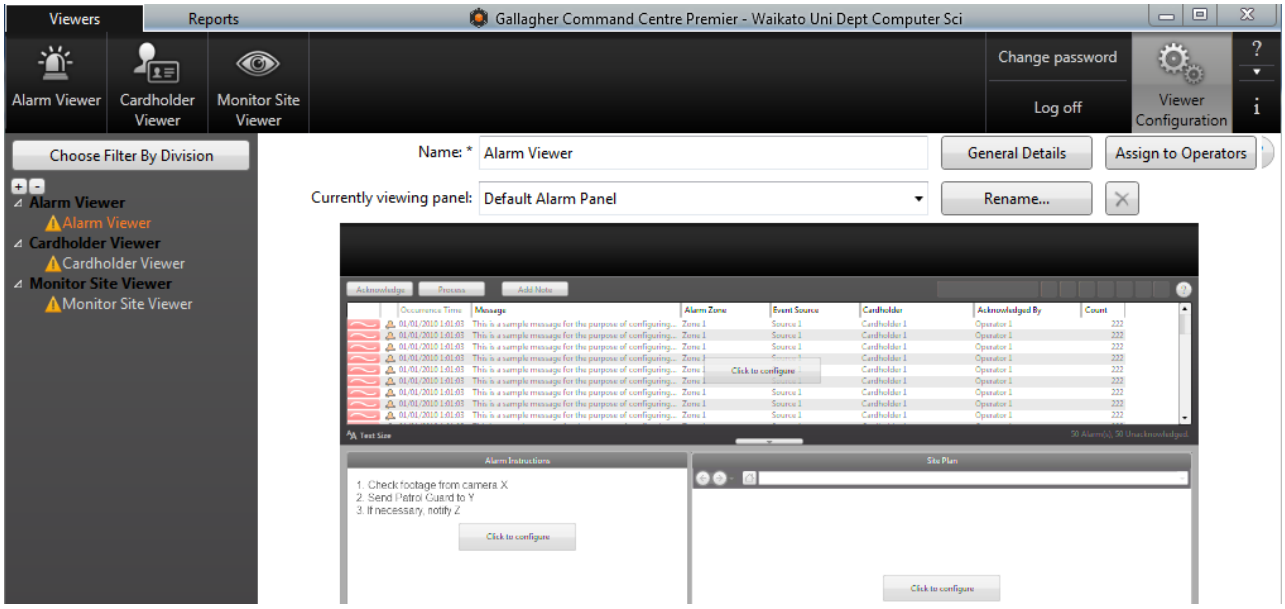

Fig. 5. Gallagher Command Centre

3) PVSio-web: This, for us, provides the counterpart to μCharts and PIMs. It is a web-based graphical "front end" to PVS [7] (via PVSio) which allows visualisation via charts and also (though we will not be using this feature in this paper) via mock-ups of interactive systems by allowing a pictorial representation of a system to have areas of a picture made "sensitive" and linked to handlers within the PVS model. This allows a modeller to demonstrate a model in a fairly realistic way, since we could have a picture of the device being modelled and sensitise areas of the picture to behave as though we had an actual device to play with.

4) Presentation Models: Presentation models describe an interface and its interactivity (either an actual implemented interface or a design artefact such as a prototype) by way of its component widgets. Each separate window or dialogue of a UI - or each unique mode of an interactive device - is described in a presentation model and then these are collected together to form the complete UI (or device) presentation model. Each widget is described as a tuple consisting of an identifier, a category (which denotes the nature of interaction) and a set of behaviours associated with this widget.

5) PIMs: While the presentation model of a device describes all possible behaviours of that device (in all of its given modes), it says nothing about the availability of those behaviours, *i.e.* it cannot be used to determine whether or not a user can ever access the described behaviours or whether the system contains undesirable properties such as deadlock. The presentation model is therefore used as a component within a Presentation Interaction Model (PIM) which *can* be used to consider such properties.

The PIM is based on a finite-state description, where each state represents a mode or window (or rather its associated presentation model). This abstraction enables the development of PIMs of systems and devices which avoid a state space explosion (as the number of states is linked to the number of different windows or modes rather than behaviours, which are 'hidden' in the presentation model). The visual representation of the PIM is given using the μCharts visual notation [18].

6) Z Specification and PMR: The functional behaviour of the systems we model is described in a formal specification using Z [19]. A relation is created between each functional behaviour (associated with a widget) in the presentation model and an operation described in the Z specification (this is the presentation model relation, or PMR) which then gives meaning to these functional behaviours.

B. Instantiating Callback

Recall that this pattern suggests itself when we wish to prevent certain kinds of single-state duplication within a model. If there is a single state which can be reached from many other states, but from which the system can only ever return to the state it was reached from, then the *Callback* pattern is the appropriate solution.

Note how Callback "fits" the pattern given earlier in Figure 2: *X* is *login*, *LV* is *prev*, *A*0 is *ao*, *Cancel* is *cancel*, and so on.

As a μchart the 'logout' example looks like Figure 7 and the emulink/PVSio looks like Figure 8. Note the similarity between these two charts (apart from the way we have laid them out, of course!). This similarity (*i.e.* the structure and the shape and the way parts fit together) is of course the essence of what we try to capture by the idea of a pattern.

C. Instantiating Binary Choice

Again we can see how the single pattern for binary choice from Figure 4 can be instantiated to give both a PIM version (Figure 9) and an emulink version (Figure 10).

V. Patterns not yet Formalised

Some sorts of pattern—those where the shape or form of design can be captured (as in the two examples earlier)—are

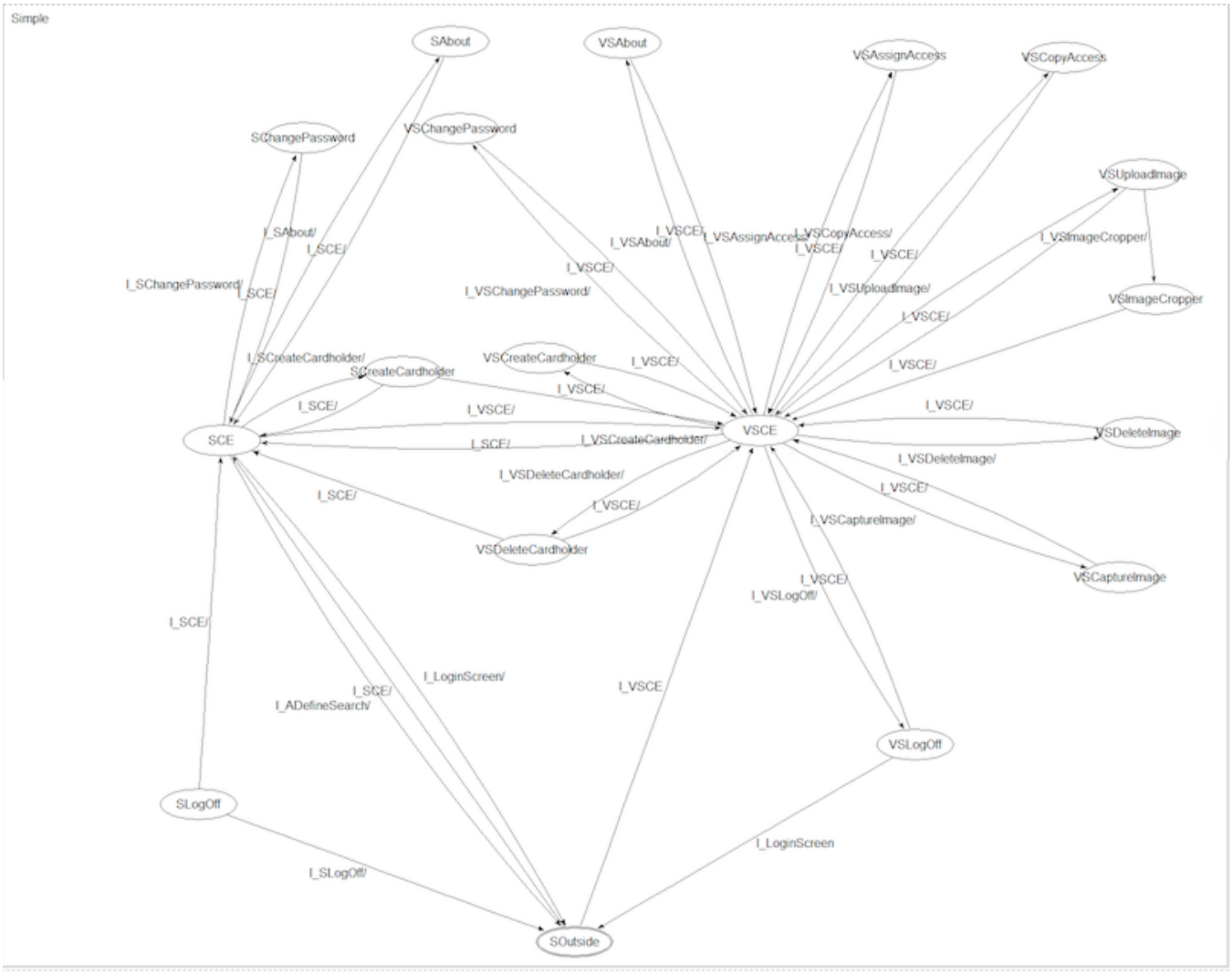

Fig. 6. PIM of Top-level of Command Centre

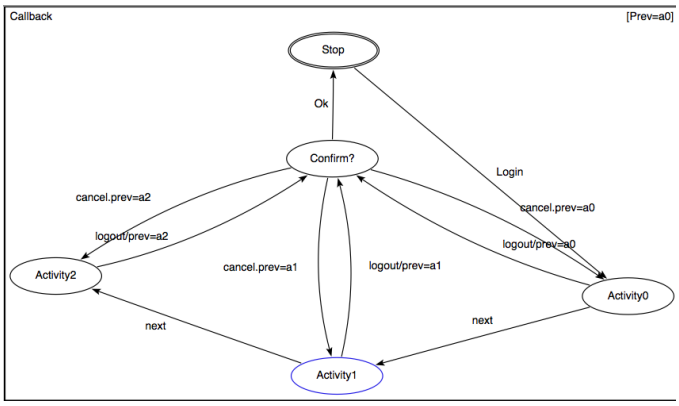

Fig. 7. Callback example using a (extended) PIM

amenable to formalisation. Some examples, like the two which follow, may have to stay informal because they require not so much conformance to certain shapes (or forms) but to the occurrences of certain relationships. Of course, we might (we hope) be wrong, but currently it is not clear how to capture formally what these two examples express. Nevertheless, they do express patterns in the sense of being guidance about how to write a design when in a certain situation.

We use a more standard textual approach to describing these - so we introduce the pattern, the problem it addresses, describe the pattern and then give an example of its use.

A. *Iterator Pattern*

This is identified when we are describing widgets such as numeric entry keys, where the behaviour of each widget is identical, but for the fact there is a value parameter associated with each of the keys. The interface to the CME pain management pump has a keypad for digits in the range of 0 to 9, and with the exception of the keys with the values 2 and 0 (which have multiple functions) each of these keys behaves in exactly the same way.

1) The Problem: We would typically model each of these numeric widgets separately, but with the same defined behaviour. In presentation models, for example, each independent widget of the device is described within its own triple in

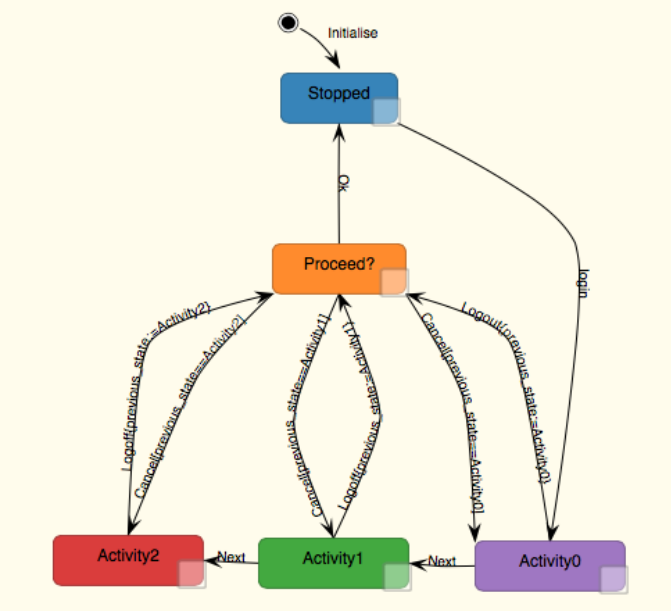

Fig. 8. Callback example using PVS *etc.*

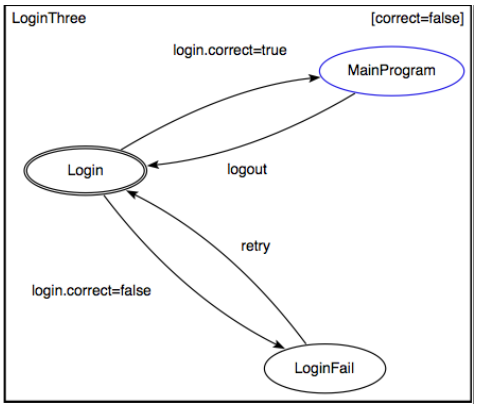

Fig. 9. Login FSM Enhanced with System Behaviour

the presentation model along with its category and behaviour. However, there are two problems with this approach. Firstly, we have unnecessary repetition, which makes the modelling a more time-consuming process and the models themselves more verbose. Secondly, the description does not accurately capture the fact that the behaviour of each of these keys is in fact subtly different due to the value parameter associated with it.

2) The Pattern: The Iterator pattern provides a solution to the occurrence of multiple widgets with identical behaviour apart from a parameter value. Note that it is not constrained to numeric keys as per the example above, it is equally applicable to other groups of widgets which behave in a similar manner. The pattern abstracts the parameter out from the interaction model enabling the keys to be modelled as a single widget, but includes the parameter within the functional behaviour description. We give some samples of this next.

3) Examples of Use: Staying with the numeric key example from the CME Pain Management Pump, we can describe these within each of the pump modes in a presentation model as

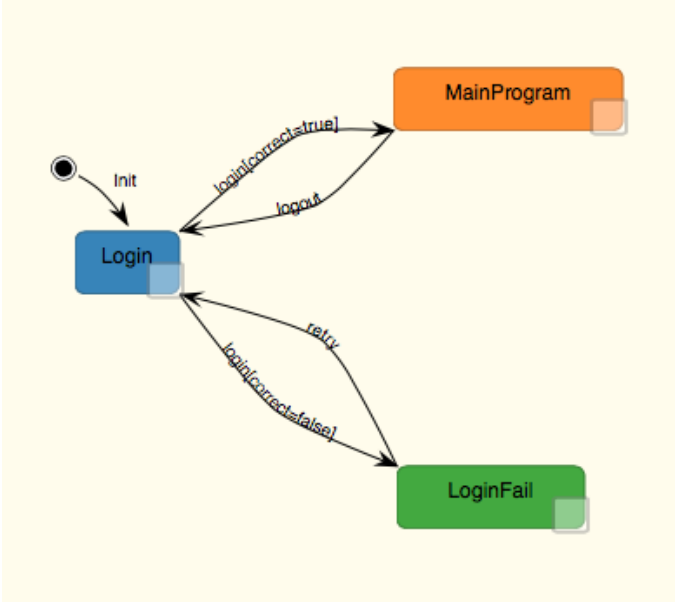

Fig. 10. Binary pattern (login example) in emulink

follows:

```
SetUpMode is
 0Key, ActionControl,(S\_DisplayVal,S\_Dec)
 2Key, ActionControl,(S\_DisplayVal,S\_Inc)
 numKey, ActionControl,(S\_DisplayVal)
```

The '0' and'2' keys are modelled separately as they have additional behaviours, but the rest of the numeric keys are abstracted into a single description called 'numKey' which have a behaviour called "S_DisplayVal". In the system specification the operation which describes this behaviour is then as follows:

SetDisplayValue

$\Delta PumpSystem$

$i? : \mathbb{N}$

$displayedValue' = i?$

So for each widget with the associated behaviour we have an input parameter which creates the unique behaviour for that key.

B. Update Pattern

Interface widgets do not exist in isolation from each other or from the underlying system functionality. It is often the case that interacting with a widget has an effect not only on the underlying systems, but also on other parts of the UI (for example the display). The UI values and system values exist independently from each but we need to ensure that these two values remain consistent with each other at all times.

1) The Problem: Our interface models can easily represent dependencies between different widgets through describing shared behaviours. Similarly, dealing with values that increment or decrement in a functional specification is a straightforward task. However, displays of interactive systems may be constrained to smaller/different values than the underlying

variables represented in code. So for example a 3 digit display may be used on the interface but a 32-bit signed integer used for the variable in code. We want to be certain that the two values remain consistent with each other, particularly at boundary points (so when the value displayed is at its maximum value of 999 we should be certain that an increment has the same effect on both values). We therefore need a consistent way to relate these in the models which allows us to perform the necessary proofs that this is always the case.

2) The Pattern: The update pattern requires that any values displayed as part of the user interface are explicitly included in the functional description. In a Z specification, for example we include a separate observation for displayed values and specify their relationship to other system observations which enables them to be explicitly linked. This also us to ensure consistency across different parts of the model and be certain that linked values remain the same.

3) Examples of Use: The PCA pump has a display screen which supports the display of both text and values. The maximum value that can be displayed is 9999 and there are five different value parameters that can be displayed, each with their own range of allowable values, for example:

Total infused volume 0 .. 9999 ml
Bag volume 0.1 .. 1000 ml
KVO rate 0 .. 5 ml/h

The PModel for the VTBIConfirm mode of the pump describes the 'Display' widget as follows:

(Display, MValResponder, (S_VTBIDisplay))

While we can see here that there is an S-behaviour called S-VTBIDisplay there is no indication that there is a shared value relating to this. In the Z specification however, we include this as a predicate of the related operation as follows:

OutputVTBIOperation

$$\Xi PCAPumpSystem$$
$$display! : VALUE$$

$$display! = currentVTBI$$

and also add as a predicate to the PCAPumpSystem that any observations which relate to values which may be displayed on the device interface are always the same as their corresponding values.

VI. Conclusions and Future Work

The patterns described here have emerged from modelling particular types of systems. It is likely (we hope!) that there will be other patterns which can be uncovered in similar ways, as we model more, and different, types of systems. In order to make progress with this work we intend to now look from the other direction—so rather than 'discovering' patterns during the process of modelling, we can examine known patterns from software development and see if they are relevant within modelling, and if so begin experimentation to investigate this further.

We also want to approach the problem of identifying patterns from another angle by looking at specification-level patterns that others have recognised in the past and seeing if we can adapt them for our work here. One obvious example is the promotion pattern and its associated Z formulation as given in [20].

Our aim is that the patterns described here, and future patterns, will prove useful not only for our own work, but also for others modelling interactive systems.

One outcome of having started the process of giving the patterns formally is that we can start to build, in the style and following the methods of [21], an algebra of patterns. We would first need to develop pattern composition operations (Bayley and Zhu [21] have six in the area of pattern-oriented software design, and some of those may be useful for us, but we would expect there to be new ways of composing that make conceptual sense in the area of interactive system design which differ from theirs). Having developed some operations we can then follow the path set out in [21] and prove a set of algebraic laws that our operations obey, prove properties of the algebra (completeness, normalisation *etc.*) and show how existing models can be expressed in this way, and reasoned about. This is, of course, looking a long way ahead, but allowing people to design their own patterns and then be able (assuming their patterns have certain properties which make them well-behaved with respect to the pattern operators) to build larger patterns from smaller ones would be a big step forward in making the design of expressive but formally-tractable interactive systems a reality.

VII. Acknowledgments

Thanks to Jeffrey Brown for initial work with Gallagher Command Centre and the Callback patterns. Also to Gallaghers Ltd. for a licence for, and support for, their Command system. Also thanks to Paolo Masci for his help with pvsio-web and for general discussions about the form of this paper.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] P. Hudak, *The Haskell School of Expression*. Cambridge University Press, 2000.

[3] J. Bowen and S. Reeves, "Formal models for user interface design artefacts," *Innovations in Systems and Software Engineering*, vol. 4, no. 2, pp. 125–141, 2008.

[4] ——, "UI-design driven model-based testing," *Innovations in Systems and Software Engineering*, vol. 9, no. 3, pp. 201 – 215, 2013.

[5] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *Journal of Systems and Software*, vol. 83, no. 2, pp. 209 – 221, 2010.

[6] "PIMed, http://sourceforge.net/projects/pims1/?source=directory."

[7] "PVSio-web, https://github.com/thehogfather/pvsio-web."

[8] "Design Patterns, http://en.wikipedia.org/wiki/design_patterns."

[9] J. Bowen and S. Reeves, "Modelling safety properties of interactive medical systems," in *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '13. New York, NY, USA: ACM, 2013, pp. 91–100.

[10] Engineering and Physical Sciences Research Council, "CHI+MED: Multidisciplinary computer-human interaction research for the design and safe use of interactive medical devices, EPSRC reference: EP/G059063/1," 2011. [Online]. Available: http://gow.epsrc.ac.uk/ViewGrant.aspx?GrantRef=EP/G059063/1

[11] P. Masci, Y. Zhang, P. L. Jones, P. Oladimeji, E. D'Urso, C. Bernardeschi, P. Curzon, and H. Thimbleby, "Combining pvsio with stateflow," in *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston,TX, USA, April 29 - May 1, 2014. Proceedings*, 2014, pp. 209–214.

[12] P. Masci, Y. Zhang, P. L. Jones, P. Curzon, and H. W. Thimbleby, "Formal verification of medical device user interfaces using PVS," in *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, 2014, pp. 200–214.

[13] J. Bowen and S. Reeves, "A simplified Z semantics for presentation interaction models," in *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, 2014, pp. 148–162.

[14] "PVS, http://pvs.csl.sri.com."

[15] "Agda, http://en.wikipedia.org/wiki/agda_(programming_language)."

[16] "Idris, http://www.idris-lang.org."

[17] "PVSio, http://shemesh.larc.nasa.gov/people/cam/pvsio/."

[18] G. Reeve, "A refinement theory for μcharts," Ph.D. dissertation, The University of Waikato, 2005.

[19] ISO/IEC 13568, *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics*, 1st ed., ser. Prentice-Hall International series in computer science. ISO/IEC, 2002.

[20] S. Stepney, F. Polack, and I. Toyn, "Patterns to guide practical refactoring: examples targetting promotion in z," in *ZB2003: Third International Conference of B and Z Users, Turku, Finland*, ser. LNCS, D. Bert, J. P. Bowen, S. King, and M. Walden, Eds., vol. 2651. Springer, 2003.

[21] H. Zhu and I. Bayley, "An algebra of design patterns," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 23:1–23:35, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2491509.2491517