Modelling a Framework for Plugins

Robert Chatley
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ
rbc@doc.ic.ac.uk

Susan Eisenbach
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ
sue@doc.ic.ac.uk

Jeff Magee
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ
jnm@doc.ic.ac.uk

ABSTRACT

Using plugins as a mechanism for extending applications to provide extra functionality is appealing, but current implementations are limited in scope. We have designed a framework to allow the construction of flexible and complex systems from plugin components. In this paper we describe how the use of modelling techniques helped in the exploration of design issues and refine our ideas before implementing them. We present both an informal model and a formal specification produced using Alloy. Alloy's associated tools allowed us to analyse the plugin system's behaviour statically.

Keywords

plugins, components, modelling, specification

1. INTRODUCTION

Maintenance is a very important part of the software development process. Almost all software will need to go through some form of evolution over the course of its lifetime to keep pace with changes in requirements and to fix bugs and problems with the software as they are discovered.

Traditionally, performing upgrades, fixes or reconfigurations on a software system has required either recompilation of the source code or at least stopping and restarting the system. High availability and safety critical systems have high costs and risks associated with shutting them down for any period of time [14]. In other situations, although continuous availability may not be safety or business critical, it is simply inconvenient to interrupt the execution of a piece of software in order to perform an upgrade.

Unanticipated software evolution tries to allow for the evolution of systems in response to changes in requirements that were not known at the initial design time. There have been a number of attempts at solving these problems at the levels of evolving methods and classes [5, 7], components [11] and services [15]. In this paper we consider an approach to software evolution at the architectural level, in terms of *plugin* components.

We believe that it is possible to engineer a generalised and flexible plugin architecture which will allow applications to be extended dynamically at runtime. Here we present a model of how components may be assembled in such an architecture based on the interfaces that they present. This model will be used at run-time by a plugin framework to determine the connections that can and should be made between plugins (our implementation of such a framework is detailed in [3]).

The benefits of building software out of a number of modules have long been recognised. Encapsulating certain functionality in modules and exposing an interface evolved into component oriented software development [2]. Components can be combined to create systems. An important difference between plugin based architectures and other component based architectures is that plugins are optional rather than required components. The system should run regardless of whether or not plugin components have been added, but offer varying degrees of functionality depending on what plugins are present. Plugins can be used to address the following issues:

- the need to extend the functionality of a system,
- the decomposition of large systems so that only the software required in a particular situation is loaded,
- the upgrading of long-running applications without restarting,
- incorporating extensions developed by third parties.

Plugins have previously been used to address each of these different situations individually, but the architectures designed have generally been quite specifically targeted and therefore limited. In existing systems, either there are constraints on what can be added, or creating extensions requires a lot of work on the behalf of the developer, for example writing architectural definitions that describe how components can be combined [13]. We believe that it is possible to engineer a more generalised and flexible plugin architecture not requiring the connections between components to be explicitly stated.

Here we describe how formal specification techniques helped us in developing a generalised plugin model that can be used to deal with any of the situations described above. Unlike other plugin models (for example that used by Eclipe [13]), in our model components are matched purely based on information that is available from the code, rather than using meta-data such as an IDL description. In the remainder of the paper we present our model both informally,

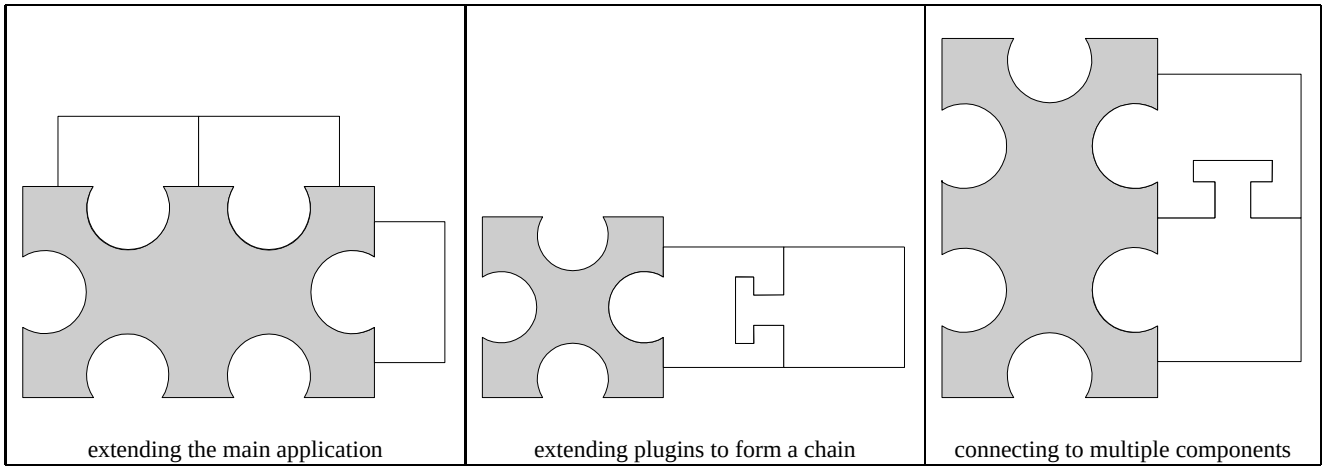

Figure 1: Some possible configurations of plugins

based on a familiar analogy, and formally using the specification language Alloy.

2. AN INFORMAL MODEL

We think of the way that components fit together in a plugin architecture as being similar to the way that pieces of a jigsaw puzzle fit together. As long as a jigsaw piece has the right shaped peg, it can connect to another piece that has a corresponding hole.

The main application provides a number of holes, into which components providing extra functionality can plug. Plugins are optional components containing collections of classes and interfaces. The holes represent an interface known to the main application, and the pegs represent classes in the plugin components that implement this interface. The interface defines the signatures of methods in the class. If an application has an interface that allows other components to extend it, and a plugin contains a class that implements this interface, a connection can be made between them. The peg will fit into the hole. This situation, adding components to a central application, is shown in the first example Figure in 1.

Thinking about plugins in this way, it becomes clear that some other more sophisticated configurations would be possible if we allow plugin components to have holes as well as pegs, *i.e.* if we allow plugins to extend other plugins rather than only allowing them to extend the main application. We can then have chains of plugins as shown in the middle example in Figure 1. An example of this situation might be if the main application were a word processor, which was extended by plugging in a graphics editor, and this graphics editor was in turn extended by plugging in a new drawing tool.

It is possible that a component has several holes and pegs of different shapes (probably the most common situation in traditional jigsaw puzzles). This can lead to more complicated configurations of components, such as those shown in the rightmost example in Figure 1. Such a configuration might be useful in a situation where the main application was, say, an integrated development environment, the first plugin was a help browser, and the second a debugging tool. The debugging tool plugs into the the main application, but also into the help browser so that it can contribute help relevant to debugging. In this way the help browser can display help provided by all of the different tools in the IDE, with the help being stored locally in each of the separate tools. It is clear that we cannot represent all possible configurations of plugins using these simple planar jigsaw representations, but they provide a useful metaphor for thinking about what might be possible.

If we think once again about the first case, then it seems that we should be able to keep on adding plugins to the application as long as they implement the right interface, but there might be cases where we want to put limits on the number of plugins that can be attached. This might be the case when each plugin that is added consumes a resource held by the main application, of which a limited quantity is available. Cardinality constraints can also be employed to constrain the shapes that the configuration can take.

To see the effect of using cardinalities, consider a main application which accepts a certain type of plugin, without a restriction on how many plugins can be added. If three compatible plugins are added, all three will be loaded and connected to the system. If, however, we change the cardinality of the interface to be ≤ 2, *i.e.* any number up to a maximum of two, after two plugins have been added, a third cannot be. It might be possible to remove plugin 1 or 2, and to replace it with plugin 3, but it is not possible to plug in all three at the same time. In practice though it seems that the two cardinalities used most often will probably be ≤ 1 and "any number".

Revisiting the chaining patterns that we saw earlier (see the second example in Figure 1), but employing cardinalities, we can chain together a number of different components of the same type, by having each provide and accept one peg of the same shape (limiting the number of pegs accepted requires a cardinality constraint - see Figure 2). This is almost like a Decorator pattern [6] for components. A decorator conforms to the interface of the component it decorates so that it adds functionality but its presence is transparent to the component's clients. Such a situation might be useful if, for instance, we wanted to chain together video filters, each of which took a video stream as an input and provided another stream as an output. Each filter could perform a different transformation (for example converting the image to black and white, or inverting it) but the components could be combined in any order, regardless of the number in the chain. Plugins would allow this configuration to be changed dynamically over time.

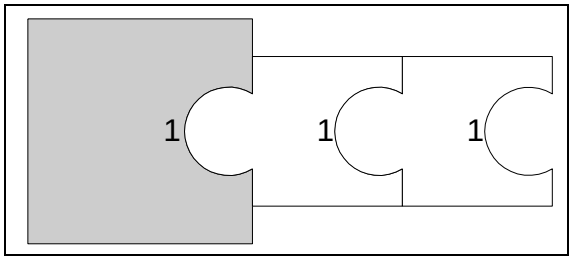

Figure 2: Chaining with cardinality constraints

It is our aim to provide the described plugin architectures in self-assembling systems [8]. It should be possible to introduce new components over time. For each additional component the system should make connections to join it to the existing system in accordance with its accepted and provided interfaces. It should not be necessary for the user or developer to provide extra information about how or where the component should be connected, as they may not have total information about the current configuration, or they may just want to delegate responsibility for managing the configuration to the system itself. The plugin framework should be able to assemble the components according to the types of the classes they contain.

Figure 3 shows a possible configuration of a video replay application. The main application displays video streams which are supplied by plugin components. The mixer component mixes two video streams into one, so can be used to add subtitles to a film. In the figure a mixer and a set of subtitles have been added to the application, and a film source is about to be added. The film source could connect either to the mixer or directly to the video player. In the first case, the subtitles will be applied to the film, in the second case the film and the subtitles will be displayed separately. We would like to be able to ensure that the behaviour desired by the provider of the film component is implemented or at very least to predict what will happen in this case. We need to know that the same thing will happen if the same components are combined on different occasions.

It is desirable that the behaviour of self-assembling systems can be made to be deterministic: it should be possible to determine what connections will be made when a certain component is added to a certain configuration. To ensure that this is the case, provision needs to be made for defining a strategy to decide between different possible bindings in a predictable way. The technique we use for this is to allow strategies for deciding between different possible bindings to be provided in the form of preference functions written by plugin developers.

3. A FORMAL MODEL

Before implementing a framework to support applications that are extensible with plugins, we developed a formal specification for the system in Alloy [9]. Alloy is a lightweight notation that supports the description of systems that have relational structures. The systems that we wish to describe are concerned with sets of linked components, so Alloy is a particularly appropriate language. The notation allows us to write any first-order logical expression plus transitive closure. In addition to providing language constructs that fit our domain, Alloy has the advantage that specifications are able to be analysed automatically. Analysis is supported by, the Alloy

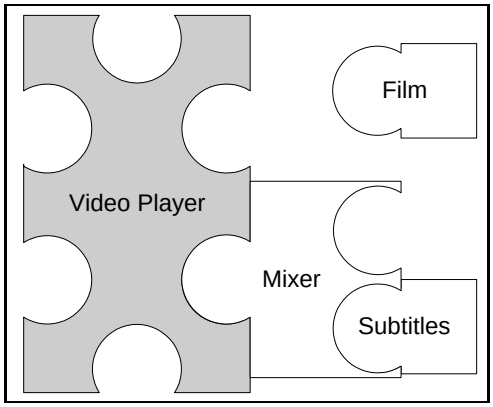

Figure 3: Non-determinism

Constrain Analyser (ACA) [4]. This tool allows us to check our Alloy models for consistency and to generate example situations which we may not have considered. Uncovering the possibility of such unexpected behaviour early in the development process allows us to refine the specification to deal with it, rather than having to do much more expensive maintenance, as would be the case if problems were discovered after implementation.

Using Alloy allows us to represent formally the way in which plugin components can fit together, and what happens when a new component is added to the system. In the case that we have written inconsistent constraints, the analyser will report that it could not generate an example that satisfies the constraints that we have specified.

The ACA tool provides a visualiser which will display example structures graphically. This representation is easy to interpret. We can see how the components have been joined together to form a system. The figures in this paper were generated by this visualisation tool (with minor hand editing of labels to make the examples easier to understand). The visualisation tool is quite flexible, allowing us to omit parts of the model and to show labels either within an object or with an arrow from the object. In Figure 4 we have used both techniques, purely for clarity.

In the text of this paper we present the model in first order logic for readability, and again in Alloy in the Appendix.

One of the ideas described in the previous section is that the number of each type of plugin component allowed may be explicitly defined. This is quite a complicated property and so we first model plugins without it and then extend the model to include cardinality constraints.

3.1 A basic model

The artifacts we model could be created by a compiler for an object oriented language with name equivalence. So they could be created by a Java or C# compiler.

Classes are defined in terms of the interfaces they implement and whether or not they are abstract. The type interface $\mathcal{I}$ is atomic.[1] We use the notation '?' to mean abstract may optionally be present

[1]For a declared type $\mathcal{T}$, $t \in \mathcal{T}$ and $t : \mathcal{T}$ will be used interchangeably.

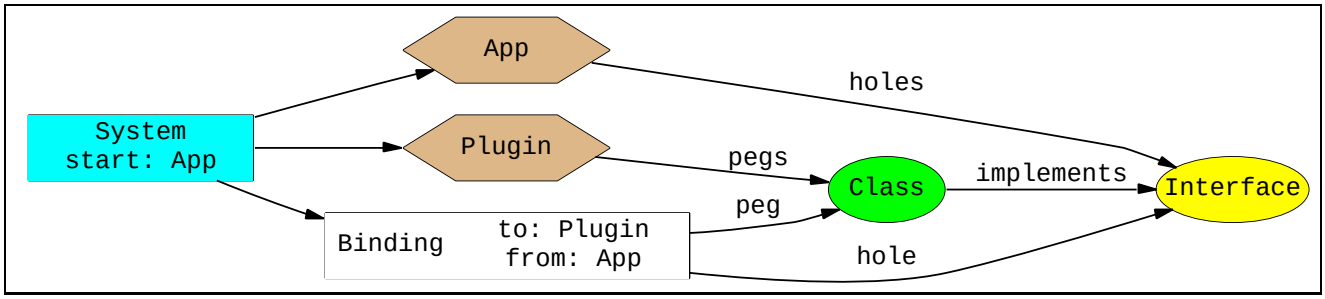

Figure 4: One component added

in a given class. As they have already been successfully compiled we know that classes must implement the interfaces they say they implement.

DEFINITION 1. *A class* $cl : \mathcal{CL}$ *is defined as:*

$$cl = \{\mathsf{implements} : \mathcal{P}(\mathcal{I}), \mathsf{abstract?} : String\}$$

Components $\mathcal{C}$ are just sets of classes and sets of interfaces. The classes constitute what the component provides and the interfaces are what the component can accept.[2]

DEFINITION 2. *A component* $c : \mathcal{C}$ *is defined as:*

$$c = \{\mathsf{pegs} : \mathcal{P}(\mathcal{CL}), \mathsf{holes} : \mathcal{P}(\mathcal{I})\}$$

Components need to be connected or bound together. Bindings $\mathcal{B}$ connect classes to interfaces. The component containing the interface has to be different from the component containing the class, so that components cannot plug in to themselves. The need for this constraint was not originally apparent. Considering examples produced by the analyser that did not follow this constraint caused us to add it to the model (see Section 4 for more discussion).

DEFINITION 3. *A binding* $b : \mathcal{B}$ *is defined as:*

$$b = \{\mathsf{peg} : \mathcal{CL}, \mathsf{to} : \mathcal{C}, \mathsf{hole} : \mathcal{I}, \mathsf{from} : \mathcal{C}\}$$

such that:

$$(\mathsf{to} \neq \mathsf{from}) \land (\mathsf{peg} \in \mathsf{to.pegs}) \land (\mathsf{hole} \in \mathsf{from.holes})$$

A System $\mathcal{S}$ consists of a set of components, a set of bindings between interfaces and classes of the components and a special component, designated start. The start component must contain at least one interface or there would be no way of ever extending a system containing it as the first component. All other components must contain some classes in order that they can provide some extra functionality to the system. An interface cannot be bound to a given class more than once (the same class in a different component is taken to be a different class).

[2]In the implementation of this model, components also include sets of resources, but these would add nothing to our model so we have omitted them.

DEFINITION 4. *A system* $s : \mathcal{S}$ *is defined as:*

$$s = \{\mathsf{comps} : \mathcal{P}(\mathcal{C}), \mathsf{bindings} : \mathcal{P}(\mathcal{B}), \mathsf{start} : \mathcal{C}\}$$

such that:

$$\begin{array}{l}
\mathsf{start} \in \mathsf{comps} \\
\exists b : \mathcal{B}.(\mathsf{start} = b.\mathsf{from}) \lor (\#\mathsf{comps} = 1) \\
\forall c \in \mathsf{comps}.(c.\mathsf{pegs} \neq \emptyset \lor c = \mathsf{start}) \\
\forall b_1, b_2 : \mathcal{B}.(((b_1.\mathsf{from} = b_2.\mathsf{from}) \land (b_1.\mathsf{hole} = b_2.\mathsf{hole}) \land \\
\quad (b_1.\mathsf{to} = b_2.\mathsf{to}) \land (b_1.\mathsf{peg} = b_2.\mathsf{peg})) \Rightarrow (b_1 = b_2))
\end{array}$$

Figure 4 shows a system with an application and a single plugin. In this system the starting component is App, which has a single interface with one method header. Plugin is added and a binding is formed from App to Plugin because Plugin contains Class, which implements Interface.

Classes and interfaces cannot exist in isolation. Every class and every interface is associated with a component. Similarly, bindings and components are always associated with systems and all components (with the possible exception of when a system contains exactly one component) are bound to other components. These constraints were not thought about explicitly before we started modelling our proposed systems. Each property has to be built into any framework that implements our model so that we create systems that behave in the way predicted by our model.

PROPERTY 1 (NO ORPHANS IN ANY $s : \mathcal{S}$).

$$\begin{array}{l}
\forall i : \mathcal{I}.\exists c : \mathcal{C}.(i \in c.\mathsf{holes}) \\
\forall cl : \mathcal{CL}.\exists c : \mathcal{C}.(cl \in c.\mathsf{pegs}) \\
\forall b : \mathcal{B}.\exists s : \mathcal{S}.(b \in s.\mathsf{bindings}) \\
\forall c : s.\mathsf{comps}.(\exists b : \mathcal{B}.c = b.\mathsf{to} \lor c = b.\mathsf{from}) \\
\qquad \lor (\#s.\mathsf{comps} = 1)
\end{array}$$

The addition of a plugin component to an existing system needs to be modelled. A component can only be added if it has a class that is not abstract that implements an interface in the existing system. But before we look at a function to add a new component to a system, we first will need to test whether two components with an associated interface and class can be bound at all.

DEFINITION 5 (canBind).

$$\begin{array}{ll}
\mathsf{canBind} & \subseteq \mathcal{CL} \times \mathcal{C} \times \mathcal{I} \times \mathcal{C} \\
\mathsf{canBind} & (cl, c', i, c) \iff (i \in c.\mathsf{holes}) \land (cl \in c'.\mathsf{pegs}.) \land \\
 & \quad (c' \neq c) \land (\mathsf{abstract} \notin cl) \land i \in cl.\mathsf{implements}
\end{array}$$

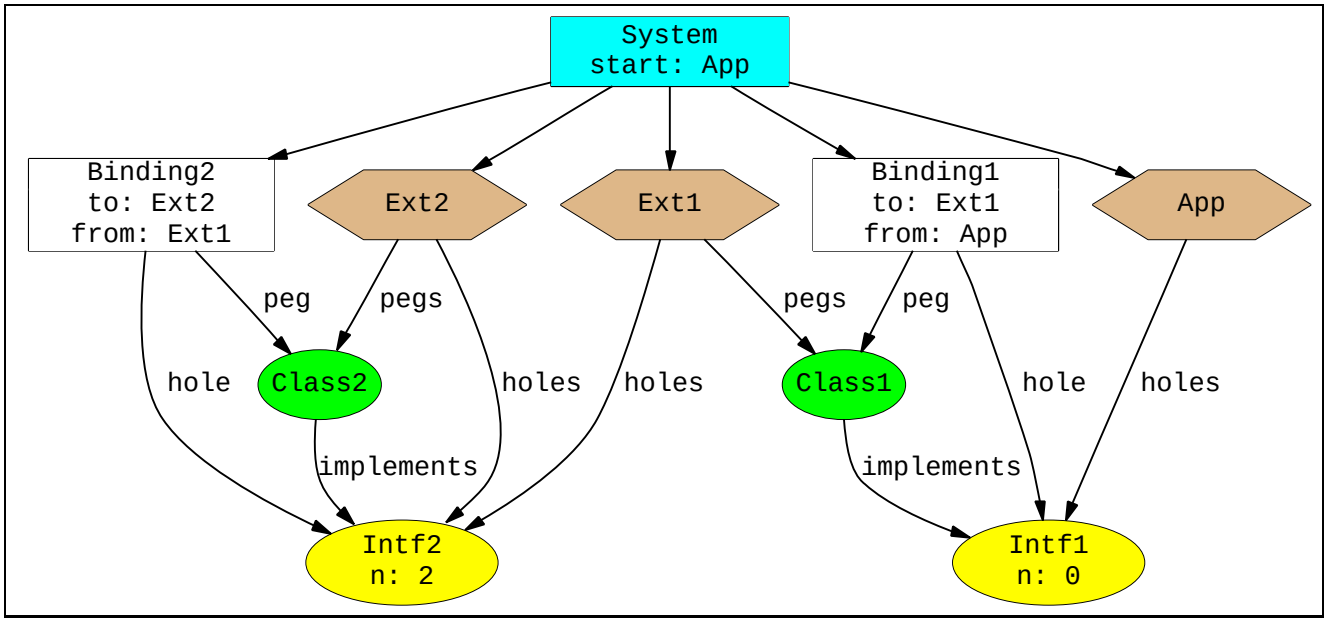

Figure 5: Several components forming a system

If a component can be bound to another component in a system, then it can be added to that system. Otherwise trying to add such a component will have no effect on the system.

DEFINITION 6 (ADDITION FUNCTIONS).

A function $\mathsf{add} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S}$ *is an addition function iff*

$$\begin{array}{l}\mathsf{add}(s, c) = s' \Rightarrow \\ (\exists c' \in s.\mathsf{comps}.\exists i : \mathcal{I}.\exists cl : \mathcal{CL}.\mathsf{canBind}(cl, c, i, c') \\ \quad \Rightarrow s' = \{s.\mathsf{comps} \cup \{c\}, s.\mathsf{bindings} \cup \{cl, c, i, c'\}, s.\mathsf{start}\}) \\ \qquad \vee \\ (\forall c' \in s.\mathsf{comps}.\neg\mathsf{canBind}(\_, c', \_, c) \Rightarrow s' = s)\end{array}$$

If there is more than one candidate for c' then there could be a set of possible addition functions each capable of performing the appropriate operation. In order for the system to be deterministic, so that behaviour holds no surprises, we need to choose a single addition function and use it. In section 3.3 we show how to do this.

3.2 Extending the Model with Cardinality Constraints

In the model described so far, the number of plugins that can be bound to a particular interface simultaneously is not prescribed. A given interface may have any number of classes bound to it. This is not always what is required. Sometimes the number of classes that can be bound to an interface is fixed. Perhaps for a specific interface only one class should be bound to it. At the other extreme an interface may let any number of classes be bound to it. For this model the numbers will be defined to be the natural numbers (including 0) extended with an infinite number.[3]

[3]In a finite Alloy model a natural number larger than the scope for which the model is analysed will have the same effect on binding as an infinite number would.

DEFINITION 7. *The numbers* $\mathcal{N}$ *are defined as:*

$$\mathcal{N} = \mathcal{N}at \cup \{\infty\}$$

Interfaces need to be extended with the number of classes that can be bound to them.

DEFINITION 8. *A numinterface* $ni : \mathcal{NI}$ *is defined as:*

$$ni = \{i : \mathcal{I}, n : \mathcal{N}\}$$

Numinterfaces need to replace interfaces throughout the definitions and in the NO ORPHANS property. More importantly, the definition of add needs to be changed to take the numbering into account. Firstly, a class can only be bound to an interface if the number associated with that interface is not zero. Secondly, when a new component is added, the number associated with the relevant interface should be decremented.

DEFINITION 9 (dec).

$$\mathsf{dec} : (\mathcal{C}, \mathcal{NI}) \longrightarrow \mathcal{C}$$

$$\mathsf{dec}(c, (i, n)) = \begin{cases} \{ \; c.\mathsf{pegs}, \\ \quad \{c.\mathsf{holes} \setminus (i, n) \\ \quad \cup (i, n-1)\} & \\ \} & \textbf{if } n \neq 0 \\ c & \textbf{if } n = 0 \end{cases}$$

DEFINITION 10 (ADDITION FUNCTIONS).

A function $\mathsf{add} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S}$ *is an addition function iff*

$$
\begin{array}{l}
\mathsf{add}(s, c) = s' \Rightarrow \\
(\exists c' \in s.\mathsf{comps}.\exists(i, n) : \mathcal{NI}.\exists cl : \mathcal{CL}.\mathsf{canBind}(cl, c, i, c') \\
\quad \Rightarrow s' = \{ \quad s.\mathsf{comps} \setminus \{c'\} \cup \{\mathsf{dec}(c', (i, n))\} \cup \{c\}, \\
\qquad\quad\; s.\mathsf{bindings} \cup \{cl, c, i, c'\}, \\
\qquad\quad\; s.\mathsf{start}\}) \\
\qquad\quad \vee \\
(\forall c' \in s.\mathsf{comps}.\neg\mathsf{canBind}(\_, c', \_, c) \Rightarrow s' = s)
\end{array}
$$

Figure 5 was produced by the Alloy model in the Appendix. This is the Alloy version of our model including numbers. The figure shows an application extended by a chain of components, as in the second example in Figure 1. Where n : 0 appears in the diagram it means that no more classes can be bound to this interface and n : 2 means that two more classes can be bound to this interface.

3.3 Removing the Nondeterminism

The final step in producing a model that is suitable for implementation is to remove the nondeterminism caused by not having a unique addition function. We need somehow to only bind to the *best* component if there is a choice of several components to which the new plugin could be bound.

Only the designer of the plugin will know, given two components that it is possible to plug in to, which would be the best choice. We need a function prefer, which the plugin developer can implement saying for every suitable pair of components, which of the two components should be bound to. If the developer does not care (it does not matter which component a plugin is connected to) then they do not need to specify a prefer function, and the binding will happen non-deterministically as in the previous case.

DEFINITION 11 (prefer).

$$
\begin{array}{l}
\mathsf{prefer} : (\mathcal{C}, \mathcal{C}, \mathcal{C}) \longrightarrow \mathcal{C} \\
\mathsf{prefer}(addc, this, that) = \left\{ \begin{array}{ll} this & \text{developer's choice} \\ that & \text{developer's choice} \end{array} \right.
\end{array}
$$

such that for each $addc \in \mathcal{C}$ prefer *induces a total order on the binding candidates.*

We next find the set of components that a given component could possibly be bound to.

DEFINITION 12 (match).

$$
\begin{array}{l}
\mathsf{match} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C}) \\
\mathsf{match}(s, c) = \{c' \mid c' \in s.\mathsf{comps}.\exists cl : \mathcal{CL}.\exists(i, n) : \mathcal{NI}. \\
\qquad\qquad\qquad \mathsf{canBind}(cl, c, i, c')\}
\end{array}
$$

Given prefer we can find the best component, amongst all those that are possible (match), to bind the new plugin to.

DEFINITION 13 (best).

$$
\begin{array}{l}
\mathsf{best} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{C} \\
\mathsf{best}(s, c) = c' \Longrightarrow \quad \forall c'' \in \mathsf{match}(s, c). \\
\qquad\qquad\qquad\qquad (c'' \neq c') \Longrightarrow (c' = \mathsf{prefer}(c, c', c''))
\end{array}
$$

Given best, we can now rewrite add so that it is deterministic. If there is no component to bind to the new component then the system without the plugin is returned. If there is one or more components that can be bound then the best one is chosen.

DEFINITION 14 (add).

$$
\begin{array}{l}
\mathsf{add} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S} \\
\mathsf{add}(s, c) = \left\{ \begin{array}{ll} \{ \quad s.\mathsf{comps} \setminus \{c'\} \cup \{\mathsf{dec}(c', (i, n))\} \cup \{c\}\}, & \\ \qquad s.\mathsf{bindings} \cup \{cl, c, i, c'\}, & \\ \qquad s.\mathsf{start} & \\ \} & \textbf{if } pre \\ & \\ s & \textbf{otherwise} \end{array} \right. \\
\textbf{where } pre = \quad \exists c' \in s.\mathsf{comps}.\exists(i, n) : \mathcal{NI}.\exists cl : \mathcal{CL}. \\
\qquad\qquad ((c' \neq c) \wedge (n \neq 0) \wedge \\
\qquad\qquad\quad \mathsf{canBind}(cl, c, i, c') \wedge (c' = \mathsf{best}(s, c))
\end{array}
$$

4. DISCUSSION

By writing an Alloy specification incrementally, and using the ACA tool to generate examples of the system's behaviour at each stage, several situations were uncovered where we had not constrained the specification strictly enough, resulting in undesirable behaviour.

Initially we had not explicitly stated that plugins cannot fill their own holes. The analyser produced an example where one component had a hole and also a matching peg which was bound to the hole. This sparked a discussion as to whether such behaviour was desirable or not. As the intention of holes is that they provide extension points where other components can be bound, we added a constraint to the model that the two components connected by a binding must not be the same component. In this way, working with a formal specification and an analysis tool led us to discuss issues that we had not considered when working with our informal model.

Another situation that came up early on, was one in which several separate groups of components were produced. Each group was connected internally, but not connected to the other groups. As execution starts in the first component, only those components that are transitively connected to the starting component will extend the base application. We therefore amended the NO ORPHANS property, so that there can be no components in the system that do not have a transitive link back to the start component.

In the model we have presented here, we have assumed that the language in which plugins are implemented will be in the style of Java or C# where the interfaces implemented by a class are explicitly named, and matched by name. Therefore in the model a class can just contain a set of interfaces which it implements, rather than us modelling all of the methods in the class and the interface. We assume that the code in plugins has passed through a compiler and so any class that says it implements an interface does in fact define the necessary methods.

If we wanted to model the implementation of plugins in a language with structural typing, where implemented interfaces are not explicitly named, but classes and interfaces are matched based on the methods that they contain, we could simply change the definitions of classes and interfaces, and write a property `implements` to check one against the other. Otherwise the behaviour of the model and the system should be unaffected.

5. RELATED WORK

There are several systems currently in existence that use plugin components as an extension mechanism. Java Applets [1] allow code to be downloaded dynamically and run in a Java-enabled web browser. The system is not particularly flexible, as all applets have to have be derived from a particular superclass, and the system cannot be used for extending applications in general.

The Eclipse platform for IDEs [13] uses plugins to allow for the addition of extra functionality. However, plugins are only detected on start-up and cannot be added to the system while it is running.

The work described by Mayer on Lightweight Application Development [12] involves a technique for using plugins with a variety of applications, but only deals with connecting extensions directly to the main application, rather than the more complex configurations that we consider.

The PluggableComponent [16] architecture features a registry to manage the different types of PluggableComponent. The registry is used by a configuration tool to provide a list of available components that administrators can use to configure their applications, so configuration is human driven, where our approach aims at automatic configuration without total knowledge of the system. As with Applets, all PluggableComponents are derived from the PluggableComponent base class, limiting flexibility of what can be used as a plugin.

There have been various attempts at formalising component based systems, for instance Jackson and Sullivan's modelling of COM in Alloy [10]. The PACC group at the SEI have been working on Prediction Enabled Component Technologies (PECT [17]). Their work aims to enable the prediction of properties of compositions of components such as latency, and to constrain the assembly of systems to configurations where certain properties hold.

6. CONCLUSIONS

We have presented a model for a system of plugin components. Developing and formalising the model caused us to consider several issues relating to what sorts of behaviours and configurations of plugins should and should not be allowed. Using the Alloy analyser helped us by allowing us to visualise different configurations that could occur with our current model. This helped us to make design decisions and refine the model further.

We have implemented a framework in Java that uses the model described here, and used it to build several applications that can be configured and extended using plugin technology. Details of the implementation can be found in [3].

In [14] Oreizy *et al* identify three types of architectural change that are desirable at runtime: component addition, component removal and component replacement. In the future we hope to extend the model presented here to cover all of these cases and to implement such a system.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the support of the European Union under grant STATUS (IST-2001-32298). We would also like to acknowledge the SLURP group at Imperial College London, especially Sophia Drossopoulou and Matthew Smith for their help with the formal model, and Matthew again for his help in producing the diagrams that appear in this paper.

8. REFERENCES

[1] Applets. Technical report, Sun Microsystems, Inc., java.sun.com/applets/, 1995-2003.

[2] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, 1997.

[3] R. Chatley, S. Eisenbach, and J. Magee. Painless Plugins. Technical report, Imperial College London, www.doc.ic.ac.uk/˜ rbc/writings/pp.pdf, 2003.

[4] D. Jackson, I. Schechter, and I. Shlyakhter. *Alcoa: the Alloy Constraint Analyzer*, pages 730–733. ACM Press, Limerick, Ireland, May 2000.

[5] M. Dmitriev. HotSwap Client Tool. Technical report, Sun Microsystems, Inc., www.experimentalstuff.com/Technologies/ HotSwapTool/index.html, 2002-2003.

[6] E. Gamma, R. Helm, R. Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[7] P. S. G. Bierman, M. Hicks and G. Stoyle. Formalising dynamic software updating. In *Second International Workshop on Unanticipated Software Evolution at ETAPS '03*, 2003.

[8] D. Garlan, J. Kramer, and A. Wolf, editors. *Proc. of the First ACM SIFGOSFT Workshop on Self-Healing Systems*. ACM Press, November 2002.

[9] D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, M.I.T., sdg.lcs.mit.edu/ dng/, February 2002.

[10] D. Jackson and K. Sullivan. COM Revisited: Tool Assisted Modelling and Analysis of Software Structures. In *In proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, 2000.

[11] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE TSE*, 16(11):1293–1306, November 1990.

[12] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development, 2002.

[13] Object Technology International, Inc. Eclipse Platform Technical Overview. Technical report, IBM, www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.

[14] P. Oriezy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, 1998.

[15] M. Oriol. Luckyj: an asynchronous evolution platform for component-based applications. In *Second International Workshop on Unanticipated Software Evolution at ETAPS '03*, 2003.

[16] M. Völter. Pluggable Component - A Pattern for Interactive System Configuration. In *EuroPLoP '99*, 1999.

[17] K. C. Wallnau. A technology for predictable assembly from certifiable components (pacc). Technical report, Software Engineering Insitute, http://www.sei.cmu.edu/pacc/publications.html, 2003.

Appendix: The Model in Alloy

```
module Plugins

open std/ord

sig String {}
sig Number {}
sig Interface{}

sig Class {
   implements : set Interface,
   abstract : option String
}

sig NumInterface extends Interface{
   n : Number
}

sig Component {
   pegs  : set Class,
   holes : set NumInterface
}

sig Binding {
   hole : NumInterface,
   from : Component,
   peg  : Class,
   to   : Component
}{
    to != from
    hole in from.holes
    peg in to.pegs
}

sig System {
   components : set Component,
   bindings   : set Binding,
   start  : Component
}{
   one start
   start in components
   some start.holes
   some bindings => some b in bindings { start = b.from }
   all c in components { c != start => some c.pegs }
}


fact noOrphans {
   all i : Interface | some c : Component { i in c.holes }
   all cl : Class | some c : Component { cl in c.pegs }
   all b : Binding | some s : System { b in s.bindings }
   all c : Component | #Component = 1 || some b : Binding { c = b.to || c = b.from }
   all c : Component | some s : System {c in s.components}
}

fun dec( c, c’ :Component, i : NumInterface ){
   i in c.holes && i.n != Ord[Number].first
   some i’ : NumInterface {
   i’.n = OrdPrev(i.n)
   c’.pegs = c.pegs && c’.holes = c.holes - i + i’
   }
}
```

```
fun add( s, s’ : System, c : Component ) {
  some c’ in s.components | some  i in c’.holes | some cl in c.pegs {
    ( c != c’ ) && no cl.abstract && i in cl.implements
    s’.start = s.start
    some c’’ : Component {
        dec(c’,c’’,i)
        s’.components = s.components + c  + c’’
    }
    one b : Binding {b.hole = i && b.from = c’ && b.peg = cl && b.to = c &&
        s’.bindings = s.bindings + b
    }
   }
}
```