University of London
Imperial College London
Department of Computing

Predictable Dynamic Plugin Architectures

Robert Chatley

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, February 2005

Abstract

Modern software systems are often assembled from collections of components. Ideally it should be possible to construct correctly functioning systems by simply deploying sets of independent components. It should also be straightforward to effect upgrades or reconfigurations after the application has been deployed. The notion of a self-organising system aims to remove as much of the configuration management effort as possible from the user or developer when working with such systems, passing the responsibility to the system itself. Unfortunately systems without explicitly defined architectures, and those subject to evolution, are prone to behaving in a surprising manner, as components from different sources are combined in different configurations. It is desirable to be confident that the system realised as the result of an upgrade or reconfiguration will behave correctly before a change is made.

We present an approach to dynamically extending applications based on *plugin* components. Plugins are optional components which can be used to enable the dynamic construction of flexible and complex systems. We present a model of plugin systems and a prototype implementation of a framework for managing them. We show how our model integrates closely with an object-oriented programming language, and requires minimal effort on behalf of the developer to create components that will work with the plugin framework.

In order to ensure the correctness of dynamic systems, some techniques for modelling and analysis are required. We generate models combining the structural and behavioural aspects of prospective system configurations and use model checking to discard those configurations that violate desired behavioural properties. In this way behavioural concerns can be used in choosing between potential configurations. By integrating our modelling and analysis techniques into the reconfiguration process, we can use the analysis that our technique provides to guide self-organisation and produce systems that behave in a predictable way.

Preface

This thesis represents the culmination of three years of work undertaken by the author at Imperial College London. During this time, several different aspects of the work have appeared in various publications. The model of plugin systems was presented in at the SAVCBS 2003 workshop [12] and an extended version of that paper, including the mapping of the model onto programming language constructs, was invited to be submitted for inclusion in a special edition of the journal Formal Aspects of Computing. The techniques for generating and checking models of plugin systems were first described in a paper published at FASE 2004 [14]. Details of the software platform developed in order to support these techniques were presented in a paper at Component Deployment 2004 [13]. The contributions of this thesis will form a chapter in a book on Plugin-Based Software Development, to be published by Springer Verlag. In all cases, this thesis should be taken to be the definitive reference for this work.

Work on this thesis has been undertaken concurrently and conjointly with work on a number of other research projects, which have provided an application for the work presented here, and also produced results in different areas of Software Engineering research. The case study presented in this thesis involves applying our techniques to provide tool support, by developing extensions for the LTSA [51] tool, for a number of different Software Engineering techniques and approaches. Our dynamic component architecture was applied to the tool and a number of different plugins were developed. The results of the different projects were published in papers accepted at the workshop Bridging the Gaps Between Software Engineering and Human-Computer Interaction [15], Visual Methods for Software Engineering 2003 [16], FSE 2004 [76], ICSE 2005 [17] and RE 2004 [75] (with an extended version of the RE 2004 paper being invited as a submission to the Requirements Engineering journal).

Acknowledgements

I would like to thank my supervisors Susan Eisenbach and Jeff Magee for their help and guidance during the course of this work, and for finding me a position in the department when my attempt at riding the dotcom wave took a turn for the worse, and then convincing me to work towards a Ph.D.

I am also very grateful to Jeff Kramer and Sebastian Uchitel for their help and advice. Discussions with them led to a clearer understanding of problems discussed in this work - and sometimes solutions. Being part of their research group has provided the opportunity to work on some interesting problems - and to visit some nice places!

Thanks to the SLURP group for their time and entertaining discussions. Sophia Drossopoulou was the inspiration for the goldfish example. Matthew Smith helped greatly in the production of some of the diagrams in this thesis during his time as a drawing tool. Thanks to Johnny Knottenbelt and William Lee for their sound technical advice.

I have had the opportunity to meet and work with some great people from other parts of the world. Thanks to my colleagues in the STATUS project, especially Natalia Juristo, Ana Moreno, Xavier Ferre and Dimitris Tsirikos. I am also indebted to Jamieson Cobleigh and Dimitra Giannakopoulou at NASA, my foremost testers and bug reporters.

Thanks to my friends from the department for all of the discussions that we have had and for making me want to come in to work every day (well, most days): Chris A, Matthew, Johnny, Will, Ashok, Nick, Gulden, Xiang, Alex A, Alex B and everyone else.

For helping take my mind off work when not in the office, to Paul, Marcus, Richard and all at ICSO; Sarah, Lev and all at RO (and lots of other people) "thankyou for the music". Lastly, a big thankyou must go to Jane for all her support.

For financial support, I would like to acknowledge the European Union for funding under grant STATUS (IST-2001-32298), without which this work would not have been possible.

Contents

1. Introduction 1

1.1. Plugin Architectures . . . 2

1.2. Predicting Behaviour . . . 5

1.3. Using Analysis to Drive Self-Organisation . . . 6

1.4. Thesis Structure . . . 8

2. Background 9

2.1. Component Technologies . . . 9

2.1.1. Java and JavaBeans . . . 9

2.1.2. ActiveX . . . 10

2.1.3. .NET and the CLI . . . 11

2.2. Service-oriented component models . . . 12

2.2.1. OSGi . . . 12

2.2.2. Avalon . . . 13

2.3. Web Browser plugins . . . 14

2.3.1. Internet Explorer . . . 15

2.3.2. Netscape Communicator . . . 16

2.3.3. Firefox and XUL . . . 17

2.4. Dynamic Extension Mechanisms . . . 17

2.4.1. Java Applets . . . 17

2.4.2. Lightweight Application Development . . . 18

2.4.3. PluggableComponent . . . 18

2.4.4. Gravity 19
2.4.5. Eclipse 20
2.5. Unanticipated Software Evolution 21
2.5.1. Dynamic Linking 21
2.5.2. DejaVU, DejaVU.NET and DJVCS 22
2.5.3. HotSwapping classes in the JVM 23
2.5.4. DRASTIC and GRUMPS 24
2.6. Self-Organising Systems 25
2.6.1. Constraint-based Self-Organising systems 26
2.6.2. Analysing Executing Systems 26
2.6.3. Reflection 27
2.7. Modelling structure and behaviour 28
2.8. Deploying Specification with Components 30
2.8.1. AsmL 30
2.8.2. Proof Carrying Code 30
2.9. Summary 31

3. Modelling 35

3.1. An Analogy 35
3.2. A Formal Model 40
3.2.1. A basic model 40
3.2.2. Binding Policies 43
3.2.3. Plugin Addition 44
3.2.4. Extending the Model with Cardinality Constraints 46
3.2.5. Binding under Cardinality Constraints 50
3.2.6. Plugin Removal 51
3.2.7. Plugin Replacement 52
3.2.8. Developing the Model 53
3.3. Summary 55

4. Programming with Plugins 57

4.1. Components 57

4.2. Provisions . . . 58
4.3. Requirements . . . 59
4.3.1. Accepting multiple plugins . . . 62
4.3.2. More complex configurations . . . 63
4.3.3. Null Objects . . . 66
4.4. Plugin Removal . . . 67
4.5. Replacement . . . 69
4.6. Summary . . . 69

5. Constraining and Analysing Plugin Systems 72
5.1. Deployment . . . 72
5.2. Structure . . . 74
5.2.1. Representing static architectures . . . 75
5.2.2. Introduction to Darwin . . . 76
5.2.3. Specifying structural constraints . . . 78
5.2.4. Darwin^{i} . . . 79
5.3. Behaviour . . . 80
5.3.1. Introduction to FSP . . . 81
5.3.2. Specifying behavioural constraints . . . 81
5.4. Matching plugin concepts with Darwin concepts . . . 82
5.5. Analysing Behaviour . . . 86
5.5.1. Composing the system . . . 88
5.5.2. Changes of configuration . . . 91
5.5.3. Verifying models . . . 92
5.6. Predicting behaviour . . . 92
5.6.1. The Source component . . . 93
5.6.2. Filter components . . . 93
5.6.3. The GZip component . . . 94
5.6.4. The Adapter component . . . 95
5.6.5. The complete system . . . 96
5.6.6. A different Adapter . . . 97

5.7. Summary 98

6. Implementation **100**

6.1. Requirements 100
6.2. Implementing Plugin Addition 102
6.3. Plugin Removal 104
6.4. Plugin Replacement 106
6.5. Constraint checking 108
6.5.1. Building the Structural Model 108
6.5.2. Building the Behavioural Model 109
6.5.3. Specifying Properties 110
6.5.4. Checking the Model 111
6.6. Technical Innovations 112
6.6.1. BackDatedObserver 112
6.6.2. Distinguishing components 113
6.6.3. Multi-methods 114
6.6.4. Proxying 115
6.7. Summary 116

7. Case study: Extensible LTSA **118**

7.1. MSC Editor 120
7.2. Darwin compiler 122
7.3. Web Animator 123
7.4. NASA Assume-Guarantee Reasoning plugin 124
7.5. Other plugins 125
7.6. Constraints for LTSA plugins 126
7.7. Discussion 129
7.8. Summary 131

8. Conclusions **132**

8.1. Contributions 132
8.2. Evaluation 134

8.2.1. Encapsulation 134
8.2.2. Supporting Framework 135
8.2.3. Simplicity 136
8.2.4. Re-use 137
8.2.5. Dynamism 138
8.2.6. Predictability 140
8.3. Future Work 142
8.3.1. Automatic Discovery of Behavioural Descriptions 142
8.3.2. Use of Assume-Guarantee Reasoning 142
8.3.3. Distribution 143
8.3.4. Hierarchical Composition 143
8.3.5. Translation to Other Platforms 144
8.4. Closing Remarks 144

Bibliography **145**

A. The Model in Alloy **156**

B. Implementing a NullObject factory **158**

C. Full Compressing Proxy Model **160**

D. Full LTSA Model **163**

E. HotSwap Experiments **165**
E.1. A framework for testing 165
E.2. Tests 166
E.2.1. Changing a method body 166
E.2.2. Adding a method 166
E.2.3. Adding a field 167
E.2.4. Changing the order of methods 168
E.2.5. Changing the order of fields 169
E.2.6. Changing the order of methods in a superclass 170

E.2.7. Calling a method in a new class 171
E.2.8. Calling a method in a new class that is missing. 172

List of Figures

3.1. Plugins extending the main application 36
3.2. Plugins extending plugins to form a chain 37
3.3. Plugins connecting to multiple components 37
3.4. Forming a Pipeline 38
3.5. One component added 45
3.6. Chaining with cardinality constraints 46
3.7. Several components forming a system 47
3.8. Constructing a pipeline under 1-1 binding 50
3.9. Constructing a pipeline under 1-n binding 52

4.1. The Virtual Fish Tank application 63
4.2. Alloy diagram representing fish tank application 65
4.3. Adding a Predator to the fish tank 66

5.1. Constructing a pipeline under 1-n binding 74
5.2. Chain of two BasicFilters 85
5.3. Client provides FSPDefinition to the plugin framework 87
5.4. LTS for Client-Server system 89
5.5. Arrangement of components in pipeline with gzip 95
5.6. Screenshot from LTSA showing trace to deadlock 99

6.1. Platform architecture managing a two component application 101
6.2. Proxy objects are used to mediate between components. 106

7.1. The default LTSA with no plugins added. 119
7.2. The LTSA running with the MSC plugin added. 121
7.3. A UML style diagram showing the structure of plugin LTSA. 123
7.4. The LTSA running with the Web Animation plugin. 125

1. Introduction

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 as they are discovered. This evolution frequently involves extension of the software, as additional functionality is required.

Traditionally, performing upgrades, fixes or reconfigurations of a software system has required either recompilation of the source code or at least stopping and restarting the system. As systems are constructed from off-the-shelf components, upgrading becomes the remit of the deployment engineer rather than the developers of the constituent components. In many cases it is inconvenient or costly to stop and restart an application in order to perform a change in configuration. High availability and safety critical systems have high costs and risks associated with shutting them down for any period of time [62]. In other situations, where 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.

It is therefore important to cater for the evolution of systems in response to changes in requirements that were not known at the initial design time (*i.e. unanticipated software evolution*). There have been a number of attempts at solving these problems at the levels of evolving methods and classes [24, 8], components [46, 27] and services [63, 64, 2]. Updating the implementation of specific methods or classes may allow for small bugs to be fixed, but larger scale changes cannot be made this way. Evolution of components and services offer a coarser granularity of change, but often require a great deal of effort in order to support and manage change. Also, the results of performing an upgrade are often unpredictable.

In many cases, in addition to the need for dynamic evolution, there is also a need

for confidence that any changes made to the system will not adversely affect its behaviour [61, 23]. Before installing a new component to upgrade the software on, for example, a space probe, engineers will want to be confident that the resulting combination of components will not cause the spacecraft to malfunction. In this thesis we consider an approach to software evolution at the architectural level, in terms of *plugin* components. Dashofy *et al* agree that the architectural level offers the most flexibility for reconfiguration in systems, as component boundaries are the most loosely coupled connection points in a software system [23]. By using a plugin architecture we can construct systems from combinations of components, with the architecture changing dynamically over time.

There are a number of types of system where it may be more appropriate to fix an architecture at the outset and to only allow evolution within this architectural style. However, there is a large class of systems where it is useful to allow evolution on a more ad hoc basis. These systems form the topic of this thesis in which we address two main questions. Firstly, how can we produce a programming model that allows developers to easily create components that can be combined at deployment time to form a software system? Secondly, how can we predict behaviour so as to be confident that a system assembled in such a way will work as expected?

1.1. Plugin Architectures

Szyperski describes components as units of composition that may be subject to composition by third parties [71]. Plugin architectures fit this description well. Plugins are components that can optionally be added to an existing system at runtime to extend its functionality. In order to support runtime assembly, some form of infrastructure is required to manage the dynamic installation of these components.

An important difference between plugin based architectures and other component based architectures is that plugins are optional rather than mandatory components. The system should run equally well regardless of whether or not plugin components have been added. Although fewer features will be available if no plugins have been added, the core of the application should still execute without problems. Another key

idea behind plugin components is to minimise the effort involved in configuring and administering such a system. It should be easy to tailor the configuration of a system as required at each deployment site, by simply adding the relevant components, and to update the configuration over time.

Oreizy *et al* [62] identify three types of architectural change that are desirable at runtime: component addition, component removal and component replacement. It is possible to engineer a generalised and flexible plugin architecture that will allow all of these changes to be made at runtime.

Each plugin may expose certain interfaces that it provides and requires [50]. By matching provisions to requirements, we can identify components that can be connected. By dynamically creating bindings between these components, calls can be made from a component requiring a service to another component that provides that service.

Plugins provide the possibility of easily adding components to a working system, adding extra functionality as it is required. 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,
- the incorporation of 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 specifically targeted and therefore limited. Either there are constraints on what changes can be made to the system over time, or creating components to work with the plugin system requires a lot of work on the behalf of the developer, writing architecture definitions that describe how components can be combined [60]. We have developed a generalised framework that permits flexible applications to be developed easily, and deals with all of the above issues.

Extending functionality It is not possible to know all of the requirements for a system when it is initially developed [84, 59], therefore it is often necessary to be able to extend the functionality at a later date. For instance, consider the development of a web browser. Over time new media types will be developed and people will want to use them on the web. In order to view these new media types (for instance new video formats, or document types like Scalable Vector Graphics [79]), extra code will have to be added to the browser. It is not possible to know all of the future media types when the browser is initially developed, but it is undesirable to have to release a new version of the entire browser every time that support for a new media type is added. By providing a mechanism for plugging in extra functionality, the browser can be incrementally upgraded as new features are developed. Macromedia's set of plugins [49], which allow their Shockwave Flash animations to be displayed in popular web browsers, are an example of this.

Decomposition With large systems, different users may require different subsets of the total available functionality. If everyone has to have all of the functionality, this may lead to unnecessary use of memory and other hardware. Alternatively, if an application must run within limited resources, perhaps on a portable or embedded device, this may constrain the total functionality that can be provided. If the program can be modularised and the modules combined in configurations tailored to each individual user, then resource wastage can be minimised. Also, users will be exposed to interfaces tailored to their needs, and the software vendor can sell different elements of functionality separately. Plugins can allow for this.

An example of such modularisation is the Eclipse Integrated Development Environment [60]. Eclipse supports development in numerous different programming languages by means of a plugin for each language. Different developers can choose to install only the tools for the languages that they require, rather than having to install support for the full set. Creating a customised version of the application becomes a deployment time, rather than development time, activity.

Upgrades Upgrading long running applications is often a problem [62]. Using traditional software, or even component based software, it is not normally possible to change the configuration of a system (especially in terms of adding new functionality) without halting execution and restarting the application. Plugins can allow for the possibility of adding code modules to reconfigure a system without having to restart.

Third party products Extensions to applications are often developed by specialist third party companies. For instance, companies specialising in computer vision technology may write extensions to major video and film processing software. The developers of the main applications are unlikely to release their proprietary source code to third party developers, yet they may want to allow their applications to be extended. Providing a plugin extension mechanism caters for this, as extensions can be purchased and added to the system separately. If a third party component is leased or licensed rather than being purchased outright, it will be desirable to remove that component from the system at a time when it is no longer needed in order to avoid unnecessary costs.

1.2. Predicting Behaviour

Confidence in the correctness of running systems is hard to attain. Harder still is to have confidence in applications that are evolved through incremental addition and removal of components. A group of components may be interacting correctly, but introducing a new component to the system may cause problems. We would like to ensure that undesirable behaviour will not occur, and that configurations that might violate certain properties are not realised. Examples of such properties might be freedom from deadlock, or ensuring that components adhere to an expected protocol when interacting with other components.

The approach to achieving this certainty presented in this thesis is to build and check a model of a software system that contains both structural and behavioural information. The structural information consists of interfaces and bindings, which define sets of shared actions through which components can interact. Structural models can be

generated automatically at runtime. However, they do not provide any information about the order in which available actions will be performed. Although we may be able to reason about the structure of systems of components based on this information, we are unable to reason in any way about their behaviour.

The behavioural information comes from the developer of a component, who can supply a specification of the way that component behaves (it is impossible to ascertain the programmer's intentions automatically). However, as components from different vendors can be combined in any number of different possible configurations, there is no way of writing a definitive model of how all different combinations will behave. To produce a model of the behaviour of the complete system requires composing the behavioural models for all of the components in a particular configuration in parallel, and ensuring that components are correctly synchronised where their interfaces are bound together. In this thesis we show how such a model can be constructed, in order to facilitate the analysis of the system's behaviour.

As systems of plugin components can have components dynamically added (and removed) over time, and especially as the overall architecture may be evolved in an ad hoc rather than a predetermined manner, it is desirable that system models be generated and tested automatically. Responsibility for management of the configuration should be passed as much as possible to the system itself. We show how our structural and behavioural specification techniques can be used for this, and how our tools can generate and analyse models automatically.

1.3. Using Analysis to Drive Self-Organisation

The systems that we consider comprise sets of components, each of which may provide or require services through interfaces of different types. These interfaces may be bound together by the component framework in order for some components to provide services to others that require them. In a self-organising system [36], this binding process should be carried out automatically, without human intervention. At a basic level, the framework must match required and provided interfaces, and bind the components together appropriately. However, there may be multiple possible

configurations that meet these interface compatibility constraints. There may, for example, be a number of possible providers that could fulfil a particular requirement, in which case there would be some ambiguity as to which should be chosen. A strategy is required for selecting which configuration to realise.

Depending on the expertise of the deployer, different degrees of human intervention may be desirable in the deployment process. A home user installing an extension to a web browser or another desktop application will likely want as much automation as possible in the installation and coordination of components. Performing an upgrade when evolution is required should be a trivial task. To as great an extent as possible the system should be self-organising. The user should not need to know what is going on behind the scenes. When dealing with a more complex system, perhaps a large system being configured by a skilled deployment engineer, it may be appropriate for the deployer to have more explicit control. They may have particular constraints that they require the system to meet. If evolving the system will cause any of these constraints to be violated then the deployer will want to be informed and given the option to abort the reconfiguration before it takes effect.

There has been some previous work on using structural constraints to ensure that a particular architectural style, for example a pipeline, is followed in self-organising or self-healing systems [37, 35]. Structures that do not adhere to the selected style will not be created. We present an alternative approach, which could be used together with an approach based on structural constraints, which assesses candidate configurations of components for adherence to behavioural properties. By using the techniques introduced above, to generate models combining the structural and behavioural aspects of prospective configurations, model checking can be used to discard those configurations that violate desired behavioural properties. In this way behavioural concerns can be used in choosing between potential configurations. By integrating our modelling and analysis techniques into the reconfiguration process, we can use the analysis that our technique provides to guide self-organisation and produce systems that behave in a predictable way.

1.4. Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 presents a survey of existing work in the area of dynamic component-based Software Engineering and sets out the aims of this thesis. In Chapter 3 we develop a specification of a plugin architecture and present a formal specification of it. Then, in Chapter 4 we discuss how the concepts presented in the formal model can be expressed and used in a popular object-oriented programming language, in this case Java. Chapter 5 describes how models can be generated from running applications and analysed to predict the effects of a particular reconfiguration before that reconfiguration is carried out, and how this can be used to preserve properties of systems as they evolve over time. Chapter 6 gives details of how the ideas and techniques discussed in the previous chapters were implemented in a framework for plugin components called MagicBeans. Chapter 7 presents a case study, using the MagicBeans framework to underpin the development of an extensible tool suite for software design and analysis. Finally, Chapter 8 summarises the contributions of this work and gives some suggestions for possible future extensions.

2. Background

This chapter presents a survey of existing work in the area of extensible and dynamic component-based systems, both in commercial and research projects.

2.1. Component Technologies

This section discusses existing component technologies that aid the development of extensible software systems.

2.1.1. Java and JavaBeans

Java [39] is a programming language from Sun that has been adopted widely in industry and academia since the late 1990s. Java is an object-oriented language. There is some support for using components and dynamically adding to applications in the existing Java platform. Java classes can be compiled and deployed separately, either as individual class files, or bundled together in Jar archives, which are essentially zip files. The Java virtual machine loads classes dynamically as they are required (when their names are first encountered in the execution of the program).

This means that it is perfectly possible to start an application running without a certain class being present, and that classes can be downloaded (or even written) later on and loaded when they are needed. However, there is no mechanism within the Java virtual machine for detecting whether or not a particular class is available, loading it if it is, but continuing or doing something else if it is not. The nearest that can be achieved to this using the standard Java language is to place an instantiation of a class inside a try/catch block, and to catch a `ClassNotFoundException`. The

exception will be thrown if the class cannot be found in the virtual machine's classpath and the programmer can then opt to do something else. This is a messy solution, and leaves the management of classes to the application programmer.

JavaBeans [44] is Sun's original component technology for Java. Beans are formed by packaging classes and other resources into Jar files, but the focus is on combining and customising Beans to create an application using a graphical builder tool. They are commonly used for combining user interface components to create a complete GUI. However, JavaBeans do not provide an automated mechanism for coordinating components. Supporting assembly of applications by the deployment engineer, rather than the programmer, requires the mechanics of how applications are constructed to be as transparent as possible, and to be performed in a way that is reactive to the other components that have already been deployed in the system. The addition of Beans to an application tends to be a development time rather than a runtime activity.

2.1.2. ActiveX

ActiveX is a technology developed by Microsoft in the 1990s. ActiveX controls are reusable software components that can add specialised functionality to web sites, desktop applications, and development tools [55]. They are primarily designed for creating user-interface elements that can be added to container applications. Every ActiveX component, or *control*, must be hosted by a container.

ActiveX is built on top of the Component Object Model (COM [54]). COM is a Microsoft component technology allowing components to be deployed as binary units and used in combination to form larger software systems. COM components, and hence ActiveX controls, are required to implement the `IUnknown` interface. Through this interface, the container can call the method `queryInterface()` on the control, to find out whether it implements a particular interface, and if so to obtain a reference to that interface.

The use of this method of querying components for interfaces follows the principles of first-party binding [66]. The container is responsible for querying the control to find out whether it provides an implementation of the interface that it requires, and

if so, requests a reference. There is no real runtime support provided by COM. The developer must take care of every detail of the runtime operation, and take care to adhere to particular protocols in order for components to interoperate correctly.

There is no standard mechanism for establishing peer-to-peer connections between ActiveX components, only between the container and the control. This limits flexibility in terms of the different configurations that can be created. In order to use a service provided by another control, the requiring control must be a container for the providing control.

2.1.3. .NET and the CLI

The .NET Framework is a new software platform from Microsoft. It is designed to allow components written in many different source languages to interoperate in one software system. This is supported by the Common Language Infrastructure (CLI). According to Abrams [81], the .NET Framework is “a component model for the Internet”.

.NET components take the form of units called *assemblies*, which contain one or more modules of code, and an *assembly manifest* which is a set of metadata describing the assembly. The manifest describes all of the types provided by the assembly. Assemblies are dynamically loaded by the .NET runtime, in much the same way that classes are dynamically loaded by the Java virtual machine.

The .NET component model is intended to supersede COM [54], although it is possible to use existing COM components from within .NET programs. There is no particular runtime support for component management (in terms of binding, configuration or dynamic replacement) under the default .NET runtime.

The assembly as a unit of deployment is largely equivalent to the Jar file in the Java platform. It groups classes and other resources into a deployable unit, but is not something that can be referenced programmatically. Where with Java classes can be deployed separately, in .NET all classes must be deployed as part of an assembly. The metadata present in each assembly gives information about the types present and the interfaces implemented by a particular component.

2.2. Service-oriented component models

Some component technologies of particular interest are those that have a service-oriented model, where service providers can be interchanged, enabling the construction of dynamic systems.

2.2.1. OSGi

The Open Services Gateway initiative (OSGi) [64] Service Platform is a specification for a framework that supports the dynamic composition of services. An implementation of this specification can be integrated into applications to provide a plugin or extension mechanism. OSGi compliant applications work by managing "bundles" that are registered with a platform.

OSGi follows a service-oriented component model, where components may be dependent on services provided by other components, but not on any named component in particular. Dependencies are on the services, or interfaces, provided by other components, which allows for run-time assembly of systems. In a purely service-oriented system, the medium through which services are implemented and provided is unspecified. OSGi combines service-oriented ideas with a component model, so all services are provided by a component (or a number of different components).

Service-oriented architectures follow a pattern of having three different types of participants: service providers, service requesters, and a service registry. Providers register their service with the registry when they become available. Clients can query the OSGi registry for components that provide a certain service, and obtain a reference through which to access the service.

Bundles are OSGi's components, and they comprise Jar files containing Java code, libraries, other resources and an XML manifest describing, amongst other things, services that the bundle requires and provides.

The Service Binder [10] is a project to provide support for the dynamic availability of services, and to manage dependencies between services. It is was developed on top of Oscar [41], an implementation of the OSGi framework. OSGi does not itself provide support for service dependency management. Bundles to be used with the

Service Binder and OSGi must contain an XML *instance descriptor* containing various meta-data about component instances to be created, cardinalities and the service's dependencies.

Dependencies can be defined as being static or dynamic. With a static dependency, the Service Binder guarantees that a required service is present for the whole time that the service requiring it is valid (*i.e.* available for use through the Service Binder). This condition is not guaranteed for dynamic dependencies. Dynamic dependency is more akin to the idea of plugins that may be added or removed from the system at runtime.

Programming extensible applications on top of the OSGi framework alone requires a lot of code to be included in the application components to register with the OSGi registry, to request services and deal with notification events from the framework. Using the Service Binder, a lot of this management is dealt with, which simplifies the programmer's task. In the XML instance descriptor for a bundle that requires a service, method names can be given to specify which methods should be called when services become available or unavailable.

2.2.2. Avalon

Avalon [2] is a service oriented component framework from the Apache foundation. It allows classes (which it calls components) to be packaged together into composite components (which it calls *blocks*) for deployment. Various metadata are required at both the class and the block level. A variation on the Javadoc style of comments is used to give names, version numbers *etc.* to classes in the source code. Special build techniques translate these into an XML type descriptor which is generated and included in the block. Dependencies on services to be provided by other components can also be specified using this Javadoc style syntax.

Avalon is supported by the Merlin runtime system. This has several features, including managing context and configuration information, and providing the target application with mechanisms for accessing this. This makes it easy for different deployments of the same components to easily be configured with, for instance, different database locations or authentication details.

Merlin also manages the availability of services for components that require them. Components can implement the `Serviceable` interface, which allows the Merlin framework to call a method `service( ServiceManager sm )`, passing a reference to the service manager. Components can then request from the service manager references to objects that provide the services that they require. An example of this is as follows (taken from the Avalon tutorial):

```
* Servicing of the component by the container during
* which service dependencies declared under the component
* can be resolved using the supplied service manager.
*
* @param manager the service manager
* @avalon.dependency type="tutorial.RandomGenerator:1.0" key="random"
* @avalon.dependency type="tutorial.Identifiable"
*/
public void service(ServiceManager manager) throws ServiceException {
   m_random = (RandomGenerator)manager.lookup("random");
   m_identifiable =
     (Identifiable) manager.lookup(Identifiable.class.getName());
}
```

This code excerpt shows how the developer must use a string name to look up the service that they require, which cannot be checked for correctness by the compiler, and must cast the result of their request to the type that they require, another point where the program could fail at runtime.

Merlin will not instantiate a component until its dependencies can be fulfilled, but does not deal with the dynamic availability/unavailability of services, or dynamic reconfiguration of systems.

2.3. Web Browser plugins

Modern web browsers are intended to allow the user to view many different media types, as many different types of content are now published on the web, for example: bitmap images, vector graphics, audio, video footage, or PDF documents. Code to render many of the more popular standards is included in the core of browsers such as Microsoft's Internet Explorer, Netscape Communicator and Mozilla's Firefox. However, support for proprietary standards such as MacroMedia's Flash animations

or Adobe's PDF documents requires external support available from the particular vendor.

Also, as the web browser has become one of the most prevalent tools used on desktop PCs, the ability to extend them to perform extra tasks has become sought after. In this section we discuss mechanisms for providing extensions to a selection of popular web browsers. We note that none of these make use of a generalised mechanism for extending applications that could be factored out and used to build other extensible applications. None of them allow plugins to be extended with further plugins; only one level of extension is possible, with each plugin connecting only to the central browser component.

2.3.1. Internet Explorer

Internet Explorer is a Microsoft application, which is built from a number of COM components, one of which is the web browser, one of which is the HTML renderer *etc.*

In addition, it is possible to add new COM or ActiveX components to deal with new MIME types. ActiveX components are identified using a GUID (Global Unique ID), which is a large hexadecimal number. They are registered in the Windows registry so that applications that want to use the components can find them. Internet Explorer also uses the registry to find each of the components that support different MIME types. Internet Explorer searches the Windows registry on startup to find registered extensions, and the MIME types they support.

An example of this is, if an Internet Explorer user has Adobe's Acrobat Reader installed, they can install an ActiveX plugin that allows Internet Explorer to open PDF files within the browser window. All ActiveX components must be hosted by an ActiveX container. Therefore, in order to allow chaining of plugins, each accepting plugin must itself be an ActiveX container and search for registered extensions in the registry. The common code is not factored out into a reusable framework.

2.3.2. Netscape Communicator

A Netscape Communicator user can use Netscape's extension mechanism to install extensions, in the form of plugins, that allow their browser to deal with new MIME types.

> "A plug-in is a separate code module that behaves as though it is part of the Netscape Communicator browser. You can use the Plug-in API to create plug-ins that extend Communicator with a wide range of interactive and multimedia capabilities, and that handle one or more data (MIME) types."
> - Netscape Plugin Development Documentation [58].

When the Communicator application starts up, it examines the contents of a particular directory to find the installed plugins. The directory differs depending on the platform. Plugins for Communicator are modules written in C or C++, compiled to native code, that are dynamically loaded. They take the form of Dynamic Link Libraries (DLLs) on Windows and Shared Object files (.SO) on UNIX.

All Communicator plugins are written starting from the template given in the plugin software development kit. This defines a set of *plugin methods*, which are methods that the core of Communicator will call on the plugin. These include methods to perform life-cycle management, like `NPP_Initialize` and `NPP_Shutdown`, as well as methods to interrogate the plugin for information (for example the MIME types that it supports), and notify it of events.

Another set of *netscape methods* defines the methods that the plugin can call of the main Communicator application. These include methods to send data to the application to display, *e.g.* `NPN_Status`, or to get or send to a particular URL, *e.g.* `NPN_GetURL` or `NPN_PostURL`.

As there is no runtime type checking in C++, failing to implement the interface properly will cause Communicator to crash when it tries to call a method that has not been implemented correctly. As with Internet Explorer, this is not a generalised extension mechanism. It is not possible to create plugins that extend other plugins without including a separate discovery and dynamic loading mechanism inside each plugin to be extended.

2.3.3. Firefox and XUL

Firefox [73] is the latest web browser from Mozilla. One of its design goals was to be highly extensible. This is achieved by providing an extension mechanism based on the language XUL (XML User-interface Language).

Extensions are formed as compressed packages of XUL, script files, CSS and images. Whether these conform to Szyperski's definition of components [71] is debatable, as the packages are binary due to compression, but not due to compilation. However, they can be integrated into any application that allows for extension via the XUL language.

Extension packages can be downloaded and hooked into the browser using a feature called overlays. The user interface of the main Firefox browser is written in XUL, and the use of an overlay allows the XUL for the main interface and each of the extensions to be combined. Overlays can be added to overlays as all of the XUL documents loaded at any one time are merged and aligned where XML element names match. However, XUL is not a generalised extension mechanism as it is not a full programming language and only deals with the description of user interfaces.

2.4. Dynamic Extension Mechanisms

Some mechanisms have been developed to allow particular applications or types of application to be extended, or assembled from sets of components that are decided on at runtime. Some of these are described in this section.

2.4.1. Java Applets

Java applets [70] allow modules of code to be dynamically downloaded and run inside a web browser. The dynamic linking and loading of classes that is possible with Java allows extra code that extends the functionality available to the user to be loaded at any time.

A Java program can be made into an applet by making the main class extend a class from the standard Java API, called `java.applet.Applet`, and following a few

conventions. The name of this main class and the location from where the code is to be loaded are included in the HTML of a web page. A Java enabled browser can then load and instantiate this class.

The applet concept has proved useful in the relatively constrained environment of a web browser, but it does not provide a generalised mechanism for creating extensible applications. As all applets must extend the provided Applet class, it is not possible to have an applet which has any other class as its parent (due to Java's single inheritance model).

2.4.2. Lightweight Application Development

In [52] Mayer *et al* present the plugin concept as a design pattern (in the style of Gamma *et al* [31]) and give an example implementation in Java. The architecture includes a plugin manager that loads classes and identifies those that implement an interface known to the main application using reflection [40].

Their work allows one application to be extended with multiple plugins, possibly with differing interfaces, but makes no mention of adding plugins to other plugins. The plugin mechanism is described in terms of finding individual classes to add to the system, where we are more concerned with adding larger elements of functionality encapsulated in components. Although components may contain sets of classes (along with other resources such as graphics), managing deployment at the level of components rather than classes would provide more flexibility, as the component boundary is typically the mostly loosely coupled connection in a software system.

2.4.3. PluggableComponent

PluggableComponent is a pattern that provides an infrastructure or architecture for exchanging components at runtime [78]. The 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. This relies on the human performing the configuration having total knowledge of the system.

All PluggableComponents are derived from the PluggableComponent base class. As with applets, this reduces the flexibility of systems that can be built as any class that is derived from a class other than PluggableComponent cannot be used as a plugin. This also means that the requirement for using plugins must be incorporated in the design from early on in order to prevent the need for expensive refactoring of the inheritance hierarchy. The ability to use PluggableComponent with a system cannot easily be retrofitted.

PluggableComponent has been used in an application where administrators can configure their own "shops" in an e-commerce system by choosing a particular configuration of components to specialise various aspects of the shop. The system comprises two applications, the shop system (implemented as a Java servlet), which runs all of the different shops, and a separate configuration tool which administrators use to create and configure shops in the system. The configuration tool is a GUI application, and administrators are assumed to be non-programmers. The management of the configuration is performed by a human.

An interesting feature of PluggableComponent is the provision of a mechanism for storing and transferring configured PluggableComponents based on Java serialization. PluggableComponents can be loaded from and saved to disk (or other persistent storage) by serializing the components. This allows the components to be stored already configured, so that when they are reloaded, the previous configuration will remain. However, the way that this works means that different instances of the same plugin with different configurations will be stored as separate objects in the store, rather than as a class and different configurations.

2.4.4. Gravity

Gravity [11] is an application that uses Oscar [41], an implementation of OSGi (see Section 2.2.1), to allow applications to be built dynamically from components that may vary in their availability.

Gravity is an execution environment, targeted towards user-oriented, interactive, applications, that uses a service-oriented component model to allow applications to be

constructed as an abstract composition of components. This composition may evolve and adapt over time as services become available and unavailable, and also as the user decides to integrate new components/services into the application.

Cervantes and Hall [11] identify two main challenges in creating such a service-oriented component model; these are dealing with ambiguity and dynamic availability. The Gravity environment, together with the Service Binder that runs on top of OSGi, deals well with dynamic availability, allowing applications to be reconfigured by selecting different components from a list. However, Gravity does not deal with ambiguity. There can often be multiple candidates for providing a service, or multiple possible configurations for combining sets of components. Gravity does not provide the user with any guidance in resolving such ambiguities.

2.4.5. Eclipse

The Eclipse Platform [60] is designed for building integrated development environments. It is built on a mechanism for discovering, integrating and running modules which it calls plugins.

Any plugin is free to define new *extension points* and to provide new APIs for other plugins to use. Plugins can extend the functionality of other plugins as well as extending the kernel. This provides flexibility to create more complex configurations. It is not possible to place restrictions on the number of any type of plugin added in this architecture. This may be important where resources are limited.

Each plugin has to include a manifest file (XML) providing a detailed description of its interconnections with other plugins. The developer needs to know the names of the extension points present in other plugins in order to create a connection with them. With the Lightweight Application Development technology described above, the actual Java interfaces implemented by classes in plugins are interrogated using reflection, and this information is used to organise and connect components. This reduces the effort required from the developer. If such techniques could be used in a more flexible plugin framework then perhaps a truly self organising system could be developed.

On start-up, the Eclipse Platform Runtime discovers the set of available plugins, reads their manifests and builds an in-memory plugin registry. Plugins cannot be added after start-up. This is a limitation as it is often desirable to add functionality to a running program without having to stop and restart it.

A plugin declares any number of named extension points and any number of extensions to one or more extension points in other plugins. The manifest provides quite a detailed description of the plugin and how it fits with other plugins. In this respect it is a form of Architecture Description Language. Is it mandatory for the plugin programmer to include this information.

Once activated, a plugin remains active until the platform shuts down. This means that plugins cannot be individually shut down, even if their functionality is no longer required.

In version 3.0 of Eclipse, the plugin mechanism has been made OSGi compliant (see Section 2.2.1). This affords more flexibility in the dynamic reconfiguration of the application. For example, plugins can be added without restarting the whole application.

2.5. Unanticipated Software Evolution

This section presents an overview of work on Unanticipated Software Evolution, where changes can be made to software after it has been deployed and often while it is running.

2.5.1. Dynamic Linking

With software written in languages such as C and C++, all the code for a piece of software is linked together at compile time. This means that if the code for one class or module is changed, then everything must be recompiled and relinked in order for the system to work. Languages with dynamic linking (such as Java and C#) leave the linking activity until runtime. This allows separate compilation of classes and modules, so that a new version of a certain class can be used in conjunction with old versions of

other classes without recompiling everything (as long as the change made is a binary-compatible change [25]).

The advent of dynamic linking systems means that it is now much easier to make small incremental changes to software systems, rather than having to replace a complete deployed system every time that an upgrade is required. Combined with Java's reflection system [40], dynamic linking allows us to write programs that use classes the name of which is not known at compile time. It is clear that having such a capability will be important in the creation of systems that are dynamically extensible through plugin components.

2.5.2. DejaVU, DejaVU.NET and DJVCS

Current trends in software development are such that third party components are often used to provide library functions to support common features in applications. Ideally, such a library would only have to be installed once on each machine, and several applications could share it, re-using code and saving on hardware resources.

However, different applications are often built to work with particular versions of libraries. When two applications are installed on the same machine that require different versions of the same library, which are not necessarily compatible, problems can arise. On the Windows platform this situation is commonly referred to as "DLL Hell".

If a new version of a library is backwards compatible with an old version, *i.e.* all programs that work with the old version will work with the new one, then it is desirable to replace the old with the new. However, if at any stage the old application is reinstalled, it may write over the new version with the old version, causing the second application to break. If two versions of a library are incompatible, then it may be required to install both versions side by side, and have each application use the version that is right for it.

To manage this situation in the general case is a difficult problem. Several tools and techniques have been developed for assessing the compatibility of different versions of components, both for Java and .NET. DejaVu [27, 26] and DJVCS [6] are examples

of these. They help to find the most recent version of a component that will work in conjunction with another. By this means they attempt to maintain the smallest possible working set of components in any situation.

2.5.3. HotSwapping classes in the JVM

Reloading classes into the Java Virtual Machine is something that is generally difficult to do. The default behaviour of the virtual machine is that it loads a class the first time that an instance of that class is required, or a static member of that class is accessed. This is done by means of a classloader, which reads the class file from the filesystem (or network) and loads it into the virtual machine's memory. The next time that a new object of this type is required, the class is not reloaded (even if it has changed on disk) as it is already in memory, and this cached class is used to create the new instance. Hence, it is difficult to change the definition of a class once it has been loaded into the JVM.

It is possible to write custom classloaders which override the behaviour of the default classloader. Using this technique one can write a classloader that does not keep a cached copy of the class in memory once it has created an object, and so every time that a new object is required (or a static member accessed) the class is reloaded from disk. This works, but is not particularly efficient and leads to less readable programs as classloaders have to be instantiated explicitly and used to create objects. Doing this for all object creations in a sizable program will lead to a fair amount of extra code, and likely a substantial decrease in performance. Another problem with this approach is that objects that are created with an old version of the class may persist and exist alongside objects created from a newer version of the class. As it is only the class definition that is replaced, existing objects are not transformed. This may lead to unpredictable behaviour, and may mean that it is difficult to make upgrades to classes which are initialised early, for instance the one that contains the application's main method.

In his web pages [24], Mikhail Dmitriev describes a technique for overcoming some of these problems. By using features provided by Sun's HotSpot virtual machine, it is

possible to redefine a class and have objects of that class which exist within the virtual machine transformed in alignment with the new class definition. This is subject to some restrictions, which are discussed in detail in Dmitriev's techical report [24], but certainly provides a big step towards being able to update class definitions in a running virtual machine.

The HotSwap mechanism can be accessed through the JPDA debugging API [69]. In essence this allows us to run a Java program in one virtual machine, and to run a debugger or other managing application in another virtual machine, which can attach to the first. The debugger/manager can then perform operations such as suspending/resuming execution of the target application, getting a list of the currently loaded classes, and most interestingly, submitting a set of class definitions to redefine classes currently loaded. The HotSwap mechanism built in to the target VM can then transform any objects to be of the new class (there are certain restrictions on what sorts of changes can be made to classes, see Appendix E for the results of our experiments investigating this).

Experimentation has shown that it is in fact possible to make the target and managing VMs the same VM, so that it is possible to have an application managing and upgrading itself, rather than having to run two separate applications. This is beneficial as an application can have much more information about itself than what it can glean about an application running on a different VM through the debugging interface.

2.5.4. DRASTIC and GRUMPS

DRASTIC and GRUMPS are two different architectures supporting evolution of software systems at runtime [29]. DRASTIC supports large but infrequent changes. In contrast, GRUMPS is targeted towards smaller changes, but assumes that these will occur frequently.

Both of the approaches assume that the initial developers of an application will be involved in performing the evolution. They rely on the fact that the configuration of the system is known by a human who can guide or adapt the evolution depending on the system's current configuration, and any previous evolutions that have occurred,

possibly by developing special filter components to adapt data formats, or to mediate between different components or services.

In a system composed of components that may be developed by various different parties and assembled into different systems by different system administrators, it will not be possible to characterise all possible configurations of components that may exist prior to evolution, and so a more generalised mechanism is necessary.

2.6. Self-Organising Systems

There is growing interest in the area of self-organising software systems. According to Georgiadis *et al* [37], "a self-organising architecture is one in which components automatically configure their interaction in a way that is compatible with an overall architecture specification." The self-organising nature of a system means that it aims to minimise the amount of effort required by a human administrator or manager during configuration and subsequent reconfiguration of the system.

A closely related category is those systems that are self-healing. Self-healing (or self-adaptive) systems monitor their runtime behaviour to detect deficiencies in areas such as performance, and instigate a reorganisation of the system architecture to try and overcome the problem.

In their IEEE Intelligent Systems article [61], Oreizy *et al* discuss a number of issues relating to self-adaptive software systems. They identify the importance of an approach that "maintains system consistency and integrity by examining each change and vetoing any changes that render the system inconsistent or unsafe."

Bringing these ideas together, we identify the requirement for an infrastructure that allows the explicit management of a system to be minimised while ensuring that the consistency and safety of the system is not breached by any proposed reconfiguration.

Dashofy *et al* [23] describe an approach to self-healing systems involving reconfiguration at the architectural level. They also note that "many systems will want to delay a repair until there is confidence that the result of a repair will not violate [certain] constraints." Their approach is based on the use of *design critics* [68] for architectural analysis that can be used as part of their tool suite ArchStudio 3. Each

critic monitors a certain constraint or property each time that a change is proposed to the configuration. Changes are described as architectural differences between two specifications described in the xADL 2.0 language. It is not clear what the scope of the analysis that can be performed using design critics is, or whether they can be used effectively in a continuous deployment situation, or only in the context of a design tool.

2.6.1. Constraint-based Self-Organising systems

Georgiadis [36] describes a system where applications configure themselves automatically, in line with a set of architectural (structural) constraints. In that work, components are taken to perform the configuration activity themselves, without recourse to an overall system manager. The components behave autonomously. This can place an overhead on each of the components, which could be factored out into a more traditional component management framework.

Georgiadis presents a system where a set of architectural constraints are imposed in order to ensure that the architectural specification of the system is preserved before and after reconfiguration. These constraints take the form of formulae in the modelling language Alloy[42]. Reconfiguration is carried out through the evaluation of *selector functions* which are derived from the Alloy specifications. This is a manual step, rather than an automatic process.

This work only deals with structural constraints, as a way of enforcing particular architectural styles. It is possible to reconfigure the system within the bounds of a particular style, but not to change the style of the architecture dynamically. No account is taken of the possibility of preserving behavioural properties during reconfigurations.

2.6.2. Analysing Executing Systems

Oreizy *et al* [61] recognise the need for an analysis phase in each adaptation cycle. They state that: “Ongoing adaptation continuously threatens system safety, reliability, and correctness. Therefore, facilities for guiding and checking modifications are an integral part of [an] adaptation infrastructure”.

A number of approaches to analysing the behaviour of an executing system rely on the insertion of monitors or *probes* into various parts of the systems. These may return information on performance. In order to analyse a system, a model needs to be constructed. Garlan and Schmerl [35] describe an approach to system adaptation in which execution of the target application is monitored and this information is abstracted to yield a model of the system. The model can then be analysed for conformance to various properties in terms of performance *etc.*

This approach is in a sense *post hoc*, as probes and monitors detect the behaviour of the executing system, and are used to decide whether further adaptation is desirable or necessary. In order to analyse the results of a system reconfiguration, that reconfiguration must be effected, and the resulting system measured. It is preferable to be able to predict the behaviour of the resulting system before effecting the reconfiguration, so that if the results are unsatisfactory, that particular configuration is never realised.

Dashofy *et al* [23] describe a method where before effecting a change to the architecture of the system, that change is made to a model of the architecture, in their case described in the architecture description language xADL 2.0. A set of *design critics* [68] is used to monitor changes in the model to ensure that they are valid according to a set of constraints relevant to the desired architectural style. This work currently supports only checking of structural constraints.

2.6.3. Reflection

Reflection is a technique that allows a program to talk about parts of a program. The Java reflection [40] mechanism can be used to "look inside" classes to see what interfaces and methods they provide and (to an extent) require. The use of such reflection can means that components do not have to provide explicit descriptions of themselves in the form of meta-data. The code of the component itself can be examined at runtime. Blair *et al* discuss two styles of reflection [9], *structural reflection* and *behavioural reflection*.

Structural reflection involves observing the system at the interface level, *i.e.* what interfaces are exposed by each component. It may also involve *architectural reflection*

where the reflection technique permits the programmer to observe and manipulate the system at the level of components and connectors. Component interfaces may be inspected to determine the services that they provide and require. Configuration information regarding which components are connected, and the current structure of the system may also be available.

Behavioural reflection is more concerned with observing events that occur in a system. These events may be the invocation of services through interfaces, perhaps through the calling of methods or the sending of messages, or at a higher level, events corresponding to changes in the configuration of the system. This can be implemented by means of *interceptors*, effectively proxies, that can be used to monitor when a particular method or service is accessed. In the reflective middleware described by Blair *et al* [9], operation is monitored using such interceptors, and this information is used in order to decide when to effect a reconfiguration of the system. In contrast to this, in the previous subsection we identified the need to predict behaviour rather than observing it. This would enable us to prevent undesirable behaviour rather than observing it and trying to cure it after the event.

2.7. Modelling structure and behaviour

There is a general movement towards the idea that the specification of a component should include information about its behaviour as well as its interface [5]. Several ADLs have been extended or complemented with languages for describing behaviour.

C2SADEL is a language developed by Medvidovic, Rosenblum and Taylor to support architectural evolution. They use the language to describe structural and behavioural aspects of architectures in the C2 style [53]. Structure is described in terms of components and connectors. In the C2 style systems are constructed as layers of components joined by connectors. Behaviour is described in terms of invariants, which must always hold for a particular component, and logical pre and post-conditions for sets of operations provided and required by the component. However, the use of logical pre and post-conditions for behaviour specification means that interaction protocols cannot be specified or analysed.

Wright [32], and PADL [7] are examples of architecture description languages that allow behavioural aspects of components (or connectors) to be specified. They use process calculus style notation to specify the behaviour of particular architectural entities, which allows greater scope for the specification of interaction protocols. However, while these languages allow for both structural and behavioural aspects of an architecture to be described, they do not support the separation of these concerns. They are single languages that allow an overall description to be written.

Structure and behaviour are fundamentally different aspects of a software architecture. It is often necessary to alter one independently from the other. The same set of components, each individually exhibiting a particular behaviour, may be combined to form a number of different structures. Alternatively, within a fixed structure, individual components may be replaced with upgraded versions which behave differently. In the situations considered in this thesis, the structure of the system is determined by the deployer or automatically by the runtime system. The behaviour of each component is due to the developer of that particular component, who may not know the situations in which their component will end up being used. To have a system that is flexible enough to allow these two concerns to be combined to create a full model, but to be specified and altered independently, we require a language that supports these separately.

Darwin [50], is a sufficiently abstract representation of software architecture that it supports different views, one of which is the behavioural view. Behavioural descriptions are given in the FSP process calculus. This is not a part of the Darwin language; the Darwin syntax concentrates on structural aspects. There is however provision in the tools that we have developed for processing Darwin to combine the Darwin and FSP specifications into one model. The two concerns can be kept separate and combined at the compilation and analysis stage. This is particularly useful in the work described in this thesis, as the two parts of the model come from different sources, to be combined each time that analysis is performed.

2.8. Deploying Specification with Components

The notion of delivering a component packaged together with a specification of the way it behaves has been considered by various researchers. Such specifications may take various forms, and may be used for varying purposes, but most notably involve checking the correctness of systems.

2.8.1. AsmL

The idea of incorporating a specification within a component is supported by Microsoft's AsmL [4], developed by Barnett, Schulte *et al* . AsmL is the Abstract State Machine Language. It is an executable specification language. Abstract specifications written in AsmL can be compiled into .NET intermediate language and executed in the same way as any other .NET program.

AsmL specifications can be used alongside full implementations of the component in question to allow for runtime conformance checking. Barnett has also said [3] that a good use of the specification would be to allow a client to inspect it (either manually, or using some automated tests) to decide whether or not to use the corresponding implementation. However, no framework for doing this currently exists.

2.8.2. Proof Carrying Code

Another angle on including within a component a way to check that it meets some property is the use of proof-carrying code [57]. Components can be provided along with a proof that they fulfil some property. The system on which they are intended to run can verify these proofs using a proof checker. Checking the proof is a much faster operation than generating it in the first place, and so can be carried out by the consumer before using the component.

Rather than providing a specification of what the component can do so that the consumer can check that it does what it wants, the producer is deployed with a proof that it does a particular thing. This means that if a component is to be used in different environments that require different properties, then it will have to provide different

proofs, and in order to have full flexibility, all properties will have to be predicted before the component is deployed.

In order to provide a proof of a safety property, the producer of the component needs to know what the safety property is that the consumer requires. It is not possible for the consumer to check arbitrary properties of the component, only those that the producer has predicted and generated a proof for.

2.9. Summary

We have examined a number of different technologies aimed at enabling the extension of software after its initial release. From this survey we have seen that despite numerous attempts at addressing different aspects of the problem, no one system currently provides mechanisms for evolving software dynamically while ensuring the safety of a particular reconfiguration. We now consider the work discussed and produce a set of requirements for a predictable dynamic plugin architecture.

Encapsulation Most of the technologies discussed in the survey involve the encapsulation of features to be added into modules or components, with only a few, such as PluggableComponent, operating at the individual class level. This supports the idea that the component level is the most appropriate level of granularity at which to consider system reconfiguration. Technologies such as JavaBeans and .NET assemblies allow program code to be modularised, and for functionality from different modules to be accessed programmatically, but do not directly support configuration of applications at deployment time. *We require a system that will allow the creation of applications to be managed by the deployment engineer. This requires the coordination of components by a managing framework, without the deployer needing to write "glue code" to mediate between components.*

Supporting framework Components require a container or framework that manages their execution and allows applications to be constructed. In some cases, notably with plugins for different varieties of web browser, the managing framework is the core

of the application. This allows components to be added to that particular application to extend its functionality, but does not provide a generalised mechanism for creating extensible applications, or allowing plugins to be chained together.

Other systems, such as Eclipse, strive to provide a general mechanism for assembling more complex configurations of components. This is done by having each component detail its provided and required interfaces in an XML metadata file. Version 2 of Eclipse does not allow dynamic modification of the configuration, new plugins are only detected and loaded at startup. The latest version of Eclipse has moved to using the OSGi framework, which has greater support for dynamic reconfiguration, but still requires clients to programmatically query a central registry to obtain access to services. *We require a framework for plugin components that will support the construction of extensible applications in a generalised and reusable way.*

Dynamism We discussed approaches to reconfiguring or upgrading systems at different levels of abstraction. The HotSwap mechanism allows individual methods within a class to be replaced in a running Java virtual machine. This is a much lower level of change than that concerning the deployment engineer. While it may be useful for debugging, it is unlikely to be useful in performing larger scale system upgrades.

Most of the technologies examined deal with the addition of functionality, but most do not address the problems of removing or replacing components identified by Oreizy *et al* [62]. The approaches that do most to accommodate these types of reconfigurations are the service oriented architectures exemplified by OSGi, which can deal with the dynamic availability of services. *We require a generalised dynamic architecture that will permit these three types of change: dynamic addition, removal and replacement of components.*

Simplicity With a number of the systems examined, considerable effort on the behalf of the developer is required in order to create a plugin component, or an application that can accept plugins. This often involves creating meta-data documents that describe interfaces and extension points (*e.g.* Eclipse), or using a special type of comment structure in the source code to specify dependencies (*e.g.* Avalon). The

overhead of learning these different approaches may prohibit a developer from using a particular extension system. To make the use of an extension system attractive to developers, a simple mechanism for programming with plugins is needed, which conforms to the concepts that they are familiar with, reducing the learning curve. *We require integration with a familiar programming language, following its idioms, to make the use of a plugin system intuitive for the developer. This will also allow more use to be made of the compiler for static checking of component code.*

Re-use If, as with ActiveX, every component that can accept another component as an extension has to be a container for that component, and provide discovery and registration services, this will lead to a lot of replicated code. If this management code can be factored out into a common framework to be used by all components, this will lead to the code of individual components being shorter and clearer. Rather than each application, or even each component, incorporating its own extension management mechanism, a generalised system could be created that could support many different applications. *We require that component management code should be factored into a reusable framework that is not tied to any particular application. It should not be necessary to duplicate component management code in multiple components.*

Predictability Correctness of software systems is a desirable property. We examined some systems that monitor the behaviour of running systems, *e.g.* the work of Garlan and Schmerl, that will instigate a reconfiguration on detecting a drop in some aspect of the system's performance. Such systems are reactive rather than predictive. On detecting a problem with the system they aim to cure it. It is often said that prevention is better than cure, and so a better technique would involve predicting the behaviour of a particular configuration before realising it. In this way, systems that do not perform as expected should never be created.

While some of the techniques examined will perform some sort of analysis before installing a new component, for example managing dependencies and not loading a new component until its requirements are fulfilled, none of them provide any support for checking that a reconfiguration will result in a correctly behaving system. We

discussed some techniques for specifying structural and behavioural characteristics of systems and for analysing them.

In the event that a reconfiguration is proposed, and analysis reveals that it will result in a violation of one of the desired system properties, it should be possible to abort the change. In this way, if a new component is introduced that will cause a malfunction, the deployer should be informed and given the option to back out of the change.

We require that a proposed configuration can be analysed for desired properties before it is realised, so that the deployer can gain confidence that the proposed reconfiguration will result in a correctly functioning system. It must be possible to abort the change if the result will violate system properties.

The approach presented in this thesis combines a generalised dynamic component system with automated modelling and analysis to produce a system that gives the deployer freedom to reconfigure a system in the confidence that the results will behave in the way that they expect.

3. Modelling

This chapter discusses the design of a generalised system of plugin components, addressing the issues identified in the previous chapters as being important in creating dynamically evolvable software configurable at the component level. We present our design decisions in the form of a model for how plugin components may be combined to create flexible software systems[1]. The model is first introduced through a familiar analogy, and then developed in terms of a formal specification.

3.1. An Analogy

> "A component is a unit of ... program structure that encapsulates its implementation behind a strict interface comprised of services provided by the component to other components in the system and services required by the component and implemented elsewhere." - S. Crane [66]

The notion of *plugin* components is one of separate units of software that are designed to be easily connected together. 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.

This analogy sits well with the definition above. The pegs represent the implementation of a particular service by a component, which it can provide to other components. The holes represent an interface known to one component through which

[1] Some of the material presented in this chapter was presented at the Specification and Verification of Component Based Systems (SAVCBS) workshop at ESEC/FSE 2003 [12]. It has also been submitted for inclusion in a special edition of Formal Aspects of Computing.

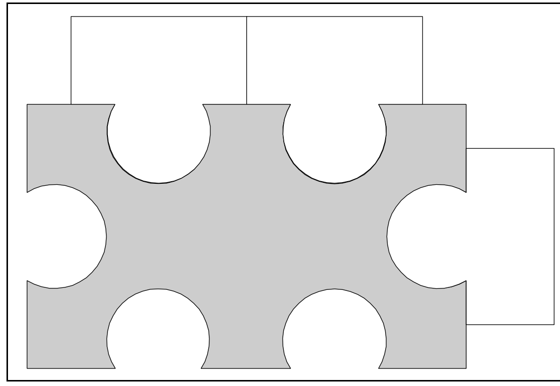

Figure 3.1.: Plugins extending the main application

it could interact with others. We will use the term *requirement* to refer to this use of services provided by other components to be consistent with previous work on software components [50], but it should be noted that plugins are optional components, so it is not mandatory that a requirement is fulfilled in order to have a working system. In this context requirements are extension points through which extra components can be *accepted* into a system when and if they are needed and available.

The core component of an application extensible through plugins may feature a number of "holes", into which components providing extra functionality can plug. If an application has an interface that allows other components to extend it, and a plugin contains an implementation of this interface, a connection can be made between them. The peg will fit into the hole. This situation, the addition of components to a central application, is shown in Figure 3.1.

Thinking about plugins in this way, it becomes clear that some other, more sophisticated, configurations would be possible if plugin components are allowed to have holes as well as pegs, *i.e.* if plugins are allowed to extend other plugins rather than only allowing them to extend the main application. Chains of plugins can then be formed, as shown in Figure 3.2. For an example of this situation, consider the main application being a word processor, extended by plugging in a graphics editor, and this graphics editor 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

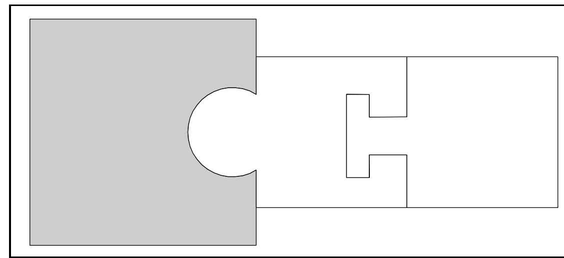

Figure 3.2.: Plugins extending plugins to form a chain

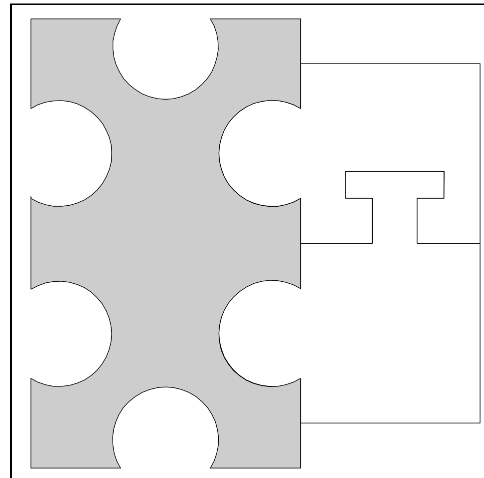

Figure 3.3.: Plugins connecting to multiple components

more complicated configurations of components, such as those shown in Figure 3.3. Such a configuration might be useful in a situation where the main application is an integrated development environment, the first plugin is 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.

As well as being able to add new components to a system, it is desirable to be able to remove components that are no longer needed. It may be the case that a component consumes a resource that has limited availability. To release this resource, either because it is no longer needed, or because it is needed by a different component, may require a component to be removed. It should be the case that a plugin component can simply be unplugged, in much the same way that a jigsaw piece may be removed.

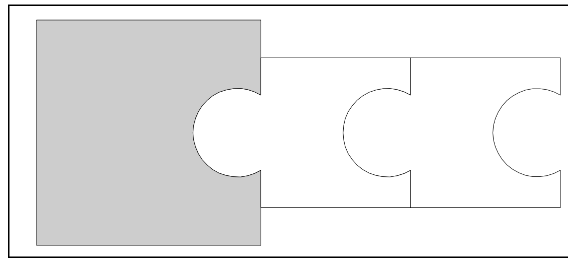

Figure 3.4.: Forming a Pipeline

In Figure 3.2, removing one of the plugins would free up a hole for another to be connected.

However, removing the middle plugin in Figure 3.2 would also cause the plugin at the end of the chain to become disconnected. In this case, considering the example given before, where the main component was a word processor extended with graphics facilities, removing the graphics editor would disconnect the tool that was plugged in to it, as no components would be using it. The question that arises is whether the component that is now disconnected should be removed from the system.

The application of a garbage collection policy would mean that such orphaned components would be removed from the system when their chain of connection to the core of the system became broken. Otherwise, orphaned components would remain in the system, but would not be able to interact with any other components. However, if later a new component was added to which they could connect, then they could function usefully again. For example, if the middle component in Figure 3.2 were removed, and the rightmost, now orphaned, component held in abeyance, then if later the central component (or a different component providing and requiring the same services) was reintroduced, then the binding to the rightmost component could be recreated.

This is the design decision that we have taken. This means that in the case shown in Figure 3.4, if the central component is removed from the chain, instead of removing the last component in the chain when it becomes disconnected, it can be reconnected where the removed component has freed up a hole, reconnecting the chain. Another reason for not garbage collecting orphaned components is that it makes the assembly

of systems of components more flexible, as, for example, chains of components as in Figure 3.2 can be built without necessarily loading the components in the correct order. If the user knows that an orphaned component will not be needed in future, it can be explicitly removed using the same removal procedure.

The third type of reconfiguration that is desirable in plugin systems is the replacement of specific components. Such an operation might be performed in order to effect an upgrade from an older version of a component to a newer one. A replacement is more than simply a removal followed by an addition; there is an extra stage. Before removing a component, it is important to ensure that replacing it with a particular component will not cause linkage errors. In order for the upgrade to be safe, the new component must be able to provide all of the services that were being provided by the old one. If this condition is met then the old component can be removed and replaced with the new one.

It is our aim to provide the described plugin architectures in self-organising systems [34]. 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. The removal and replacement of components should be managed automatically, so that the system is automatically configured with the correct set of connections between components at any point. 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. Control over what structures are formed can be exercised by applying different binding policies, and later chapters will describe how further constraints can be applied to manage what systems are constructed if more explicit control is required.

As the notion of a component is largely concerned with encapsulating implementation details behind well specified interfaces, we tested our ideas by providing an implementation in the strongly typed object-oriented programming language, Java. The components that will be considered are bundles of classes and interfaces (and also graphics, text or other data files). In practice, the component would be an archive (Jar)

file containing the binary files for the classes and interfaces as produced by the Java compiler.

3.2. A Formal Model

This section presents our model of plugin systems, developed in first-order logic. We represent formally the way in which systems can be composed from plugin components, the conditions under which a new component may be added to the system, when an existing component can be removed, and when one may be replaced. There are various constraints as to what constitutes a component, and how components can be connected together.

3.2.1. A basic model

The artifacts modelled here could be created by a compiler for an object-oriented language with name equivalence, *i.e.* decisions regarding whether two objects have compatible types are resolved by comparing their names. These sorts of artifacts could be created by a Java or C# compiler.

Classes are defined in terms of the interfaces they implement and carry information about whether or not they are abstract. The type interface $\mathcal{I}$ is atomic.[2] A class may or may not be abstract. $\mathcal{P}(\mathcal{I})$ denotes the power set of $\mathcal{I}$. As they have already been successfully compiled, we know that classes must implement the interfaces they are declared as implementing.

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

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

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.[3]

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

[3] In the implementation of this model, components also include sets of resources, but these would add nothing to the model so they have been omitted.

There may also be classes and interfaces that are used only internally to the component, rather than being exposed to other components. However, these internal entities are not relevant to this model, as it is concerned only with the interaction between different components.

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 components by linking interfaces with corresponding implementing classes (we sustain the pegs and holes metaphor). The components that form the two ends of the binding (the to and the from) must be different, so that components cannot plug in to themselves.

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

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

such that:

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

This definition places no restrictions on the types of the class and interface that feature in a particular binding. The function bind given in Definition 7 makes use of the function canBind as given in Definition 5 to ensure that in assembling a system of plugin components, only bindings where the class and interface have compatible types are constructed.

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, where execution begins. Throughout this section we discuss systems being made up from collections of components. In fact, when a component is added to a system, the classes that implement its provided interfaces are instantiated (in turn instantiating any other classes as used by these classes) creating a component instance. For brevity we will continue to refer to these constituent parts of a system as components. The start component must have at least one hole (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 named class in a different component is taken to be a different class), *i.e.* duplicate bindings are not allowed.

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:

$$\begin{aligned}
&\mathsf{start} \in \mathsf{comps} \\
&\neg\exists i : \mathcal{I}.(i \in \mathsf{start.holes}) \implies \mathcal{B} = \emptyset \\
&\forall c \in \mathsf{comps}.(c.\mathsf{pegs} \neq \emptyset \vee c = \mathsf{start}) \\
&\forall b_1, b_2 : \mathcal{B}.(((b_1.\mathsf{from} = b_2.\mathsf{from}) \wedge (b_1.\mathsf{hole} = b_2.\mathsf{hole}) \wedge \\
&\quad (b_1.\mathsf{to} = b_2.\mathsf{to}) \wedge (b_1.\mathsf{peg} = b_2.\mathsf{peg})) \Rightarrow (b_1 = b_2))
\end{aligned}$$

Classes and interfaces cannot exist in isolation. Every class and every interface are considered to be deployed as part of a component. Similarly, bindings are always associated with systems, and cannot exist outside. These constraints were not thought about explicitly before we started modelling our proposed systems. Without them the model describes systems that we do not wish to consider, for example those where classes are deployed individually without an enclosing component. These constraints are encoded in the NO ORPHANS property. This property has to be built into any framework that implements our model to ensure that it accurately predicts the behaviour of systems.

Property 1 (No orphans in any $s : \mathcal{S}$)

$$\begin{aligned}
&\forall i : \mathcal{I}.\exists c : \mathcal{C}.(i \in c.\mathsf{holes}) \\
&\forall cl : \mathcal{CL}.\exists c : \mathcal{C}.(cl \in c.\mathsf{pegs}) \\
&\forall b : \mathcal{B}.\exists s : \mathcal{S}.(b \in s.\mathsf{bindings})
\end{aligned}$$

3.2.2. Binding Policies

There are a number of possible binding policies that could be adopted in a plugin system. Which policy is chosen affects the bindings that are made between sets of components, and the structures that result. Different policies may be applicable in different situations. It should therefore be possible to select from a set of different policies. Here we discuss two different policies, and provide definitions in the form of properties. It is down to the engineer performing the deployment at a particular site to choose which property is desirable for that particular system. The plugin management framework should be able to enforce whichever policy is selected.

1-1 binding

With 1-1 binding, each provision is bound to at most one requirement. This follows the jigsaw analogy closely, as each peg on a jigsaw piece can fill only one hole at any one time. It is not possible for one peg to fill two holes simultaneously. The following property specifies that there cannot exist two bindings in a system which connect to the same peg on the same component.

Property 2 (1-1 binding in a system $s : \mathcal{S}$)

$$\forall b_1, b_2.(b_1 \in s.\mathsf{bindings} \wedge b_2 \in s.\mathsf{bindings}$$
$$\wedge (b_1.\mathsf{to} = b_2.\mathsf{to}) \wedge (b_1.\mathsf{peg} = b_2.\mathsf{peg}) \Rightarrow (b_1 = b_2))$$

However, it is not possible to create some common structures under this policy. For example, it is not possible for one server component to provide the same service to multiple client components, a situation that is often desirable. There are many occasions when it would not be efficient to have to create a new instance of a server for each client that requires a service.

Also, there may often be circumstances where there are multiple candidate binding targets available; two (or more) different components may accept the same type of plugin. In such a situation, with a 1-1 binding policy, a decision needs to be made regarding to which of the available requirements a provision should be bound.

1-n binding

With 1-n binding on the other hand, each provision is bound to all matching requirements at the time of binding (but at most one per component). Plugins can provide the same service to more than one other component, so if there are multiple possible requirements that a provision could fill, it will be bound to all of them [4]. In the case where a server may be shared by a number of client components, this configuration will automatically be generated by using this policy. This means that it is never necessary to pick the "best" target from a set of possible requirements, as connections will be made to all of them.

Property 3 (1-n binding in a system $s : \mathcal{S}$)

$$\forall b_1, b_2.(b_1 \in s.\mathsf{bindings} \wedge b_2 \in s.\mathsf{bindings}$$
$$\wedge (b_1 \neq b_2) \wedge (b_1.\mathsf{to} = b_2.\mathsf{to}) \wedge (b_1.\mathsf{peg} = b_2.\mathsf{peg}) \Rightarrow (b_1.\mathsf{from} \neq b_2.\mathsf{from}))$$

In the model specified so far, there is nothing to limit the number of pegs of a particular type that may be connected to a particular hole. A component that can accept plugins of a particular type can accept as many of these as are available. Referring to the jigsaw analogy, binding multiple times to one hole effectively creates copies of that hole, so that there is always a space for another plugin to be connected. Placing limits on the number of plugins that can be accepted is discussed in Section 3.2.4.

3.2.3. Plugin Addition

The addition of a plugin component to an existing system needs to be modelled. A component can only be bound if it contains a class that is not abstract, and that implements an interface accepted by a component in the existing system. But before looking at a function to add a new component to a system, it is necessary to test whether two components with an associated interface and class can be bound at all.

[4] Unfortunately this is not something that it is possible to describe using the jigsaw analogy, as it is akin to putting one peg in more than one hole at the same time.

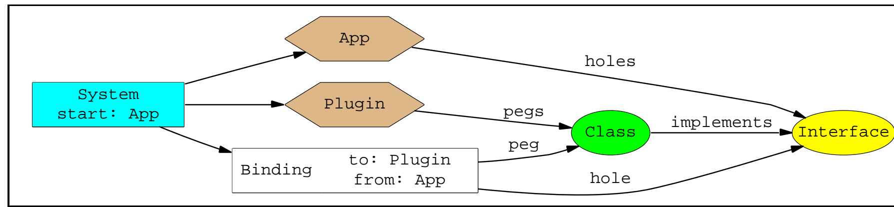

Figure 3.5.: One component added

Definition 5 (canBind)

$$\begin{aligned}\mathsf{canBind} &\subseteq \mathcal{CL} \times \mathcal{C} \times \mathcal{I} \times \mathcal{C} \\ \mathsf{canBind} &\; (cl, c', i, c) \iff (i \in c.\mathsf{holes}) \wedge (cl \in c'.\mathsf{pegs.}) \wedge \\ &\; (c' \neq c) \wedge \neg cl.\mathsf{abstract} \wedge i \in cl.\mathsf{implements}\end{aligned}$$

It is assumed that the set cl.implements contains all of the interfaces that the class is declared as implementing, plus all of the superinterfaces of each of these interfaces (if a class implements interface i, it automatically implements all of the superinterfaces of i).

If a component can be bound to another component in a system, then it can be added to that system and interact with the other components. If it cannot be bound (because none of the components currently in the system have compatible interfaces), then it can be added in a dormant state. None of its code will execute until the system configuration changes so that it can be bound.

The following three functions define how a component is added to the system. It is first loaded, *i.e.* added to the set of components that comprise the system, and then bindings are created between compatible pegs and holes. One of the binding policies should be applied so that either 1-1 binding or 1-n binding is followed as desired.

Definition 6 (load)

$$\begin{aligned}&\mathsf{load} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S} \\ &\mathsf{load}(s, c) = s' \Rightarrow s' = \{s.\mathsf{comps} \cup \{c\}, s.\mathsf{bindings}, s.\mathsf{start}\}\end{aligned}$$

Definition 7 (bind)

$$\begin{aligned}&\mathsf{bind} : \mathcal{S} \longrightarrow \mathcal{S} \\ &\mathsf{bind}(s) = s' \Rightarrow s' = \{s.\mathsf{comps}, b, s.\mathsf{start}\} \quad \Rightarrow b = s.\mathsf{bindings} \; \cup \; B(s)\end{aligned}$$

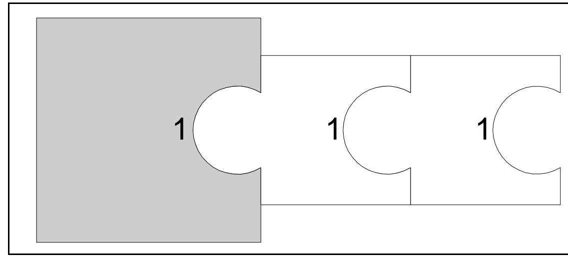

Figure 3.6.: Chaining with cardinality constraints

such that:

$$B(s) = \{(cl, c, i, c')|(c, cl) \in P(s) \wedge (c', i) \in H(s) \\ \wedge \mathsf{canBind}(c, cl, c', i) \\ \wedge \neg\exists i' : \mathcal{I}.(i' \in \mathsf{comps.holes} \wedge \mathsf{canBind}(cl, c, i, c') \wedge i' \subseteq i)\}$$

$$P(s) = \{(c, cl)|c \in s.\mathsf{comps} \wedge cl \in c.\mathsf{pegs}\}$$

$$H(s) = \{(c', i)|c' \in s.\mathsf{comps} \wedge i \in c'.\mathsf{holes}\}$$

where $i' \subseteq i$ indicates i' being a subtype of i

Definition 8 (addition function)

$$\mathsf{add} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S}$$
$$\mathsf{add}(s, c) = \mathsf{bind}(\mathsf{load}(s, c))$$

Figure 3.5 shows a system with an application and a single plugin[5]. In this system the starting component is App, which has a single hole whose type is defined by the Interface. Plugin is added and a binding is formed from App to Plugin because Plugin contains Class, which implements Interface.

3.2.4. Extending the Model with Cardinality Constraints

In the model described so far, the number of plugins that can be bound to a particular interface (of a particular component) simultaneously is not prescribed. A given interface in a component's holes set may have any number of classes bound to it. This is

[5]This fi gure was generated using the Alloy visualiser tool, see Section 3.2.8 for more details.

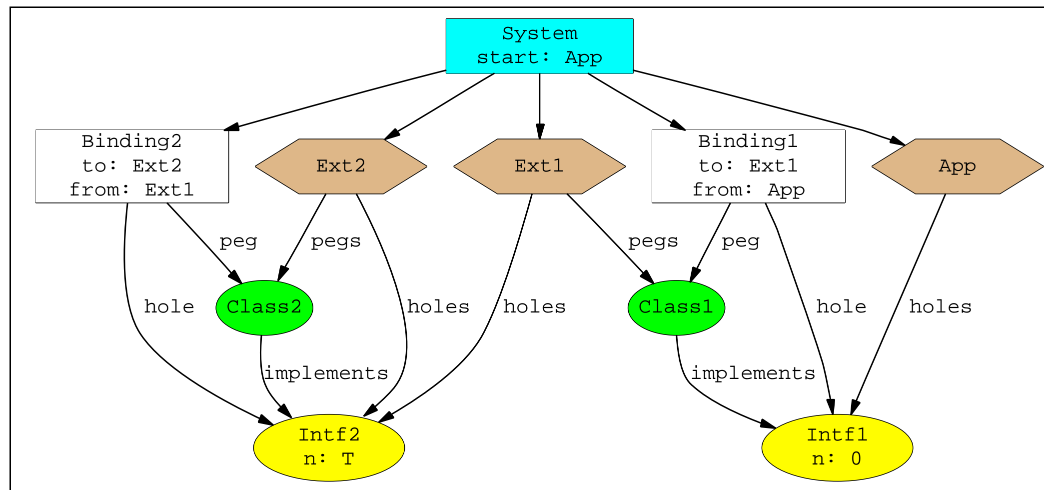

Figure 3.7.: Several components forming a system

not always what is required. Sometimes the number of classes that can be bound to an interface is limited. Returning to the first case discussed, (Figure 3.1), it should be possible to add any number of plugins to the application as long as they implement the correct interface. However, there might be cases where it is desirable to put limits on the numbers 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. Alternatively there might be multiple possible provisions available, only one of which is required. In this case one of the available provisions should be selected, either randomly or in accordance with further constraints (as detailed in Chapter 5).

Revisiting the chaining patterns discussed earlier (see the example in Figure 3.2), but restricting cardinalities, a number of different components of the same type can be chained together, by having each provide and accept one peg of the same shape (see Figure 3.6). This is similar to a Decorator pattern [31] 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 when chaining 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 (*e.g.* 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.

The effect of restricting the cardinality of a hole is to allow it to become full after a certain number of plugins are bound to it. After this, no more plugins can be bound to this interface until such time as one or more of the existing bindings are removed. In theory, the cardinality of a hole could be set as any of the natural numbers, or left unlimited. However, it is more practical to restrict the possible values for cardinalities to 0, 1 and infinity (where infinity denotes no limitation). These values are sufficient for the cases where no more components may be bound (*i.e.* the hole is full), where one more may be bound, and where there is no limit on how many more may be bound. This relates nicely to the notions of scalar values and lists. In the case that it is necessary to be able to accept precisely two plugins of a particular type, this can be achieved using two scalar holes (each with cardinality 1).

To incorporate this into the model, interfaces need to be extended so that they also model the number of bindings in which they can participate at any one time. A set of numbers to be used for possible cardinality values is defined as discussed above.

Definition 9 *The numbers $\mathcal{N}$ are defined as:*

$$\mathcal{N} = \{0, 1, \top\}$$

such that:

$$\top - 1 = \top + 1 = \top$$

$$0 + 1 = 1$$

$$1 - 1 = 0$$

Definition 10 *A numInterface $ni : \mathcal{NI}$ is defined as:*

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

NumInterfaces need to replace interfaces throughout the definitions and properties. More importantly, the definitions of canBind and bind need to be changed to take the numbering into account. In addition to the checks regarding subtyping that were

present in the previous version of canBind, a peg can only be bound to a hole if the number associated with that hole is not zero (so the hole is not full). Secondly, when a new component is bound, the number associated with the relevant interface should be decremented, so that it will become zero when the hole becomes full.

Definition 11 (dec)

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

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

Definition 12 (canBind **(revised)**)

$$\begin{array}{ll} \mathsf{canBind} & \subseteq \mathcal{CL} \times \mathcal{C} \times \mathcal{NI} \times \mathcal{C} \\ \mathsf{canBind} & (cl, c', (i, n), c) \iff (i \in c.\mathsf{holes}) \wedge (cl \in c'.\mathsf{pegs}.) \wedge n \neq 0 \\ \wedge & (c' \neq c) \wedge \neg cl.\mathsf{abstract} \wedge i \in cl.\mathsf{implements} \end{array}$$

Definition 13 (bind **(revised)**)

$$\mathsf{bind} : \mathcal{S} \longrightarrow \mathcal{S}$$

$$\mathsf{bind}(s) = s' \Rightarrow s' = \{n, b, s.\mathsf{start}\} \Rightarrow n = D(s.\mathsf{comps}, B(s)) \wedge b = s.\mathsf{bindings} \cup B(s)$$

such that:

$$B(s) = \{(c, cl, c', i) | (c, cl) \in P(s) \wedge (c', i) \in H(s) \\ \wedge \mathsf{canBind}(c, cl, c', i) \wedge \neg \exists i'.\mathsf{canBind}(cl, c, i, c') \wedge i' \subseteq i\}$$

$$P(s) = \{(c, cl) | c \in s.\mathsf{comps} \wedge cl \in c.\mathsf{pegs}\}$$

$$H(s) = \{(c', i) | c' \in s.\mathsf{comps} \wedge i \in c'.\mathsf{holes}\}$$

$$D(cs, bs) = \{k | k \in cs \wedge \neg \exists j.j \in bs \wedge j.\mathsf{from} = k\} \\ \cup \{\mathsf{dec}(k, i) | k \in cs \wedge \exists j.j \in bs \wedge j.\mathsf{from} = k \wedge i = j.\mathsf{hole}\}$$

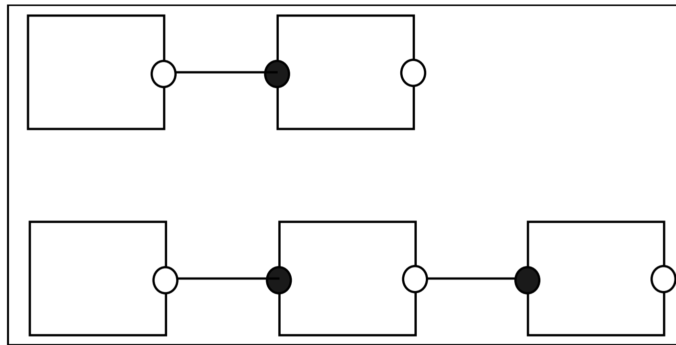

Figure 3.8.: Constructing a pipeline under 1-1 binding

The bind function creates a new system with all of the original bindings, plus new bindings as created by the function $B(s)$. The components of the new system are the same as those of the old, but with the numbers of holes available for newly bound NumInterfaces decremented. $B(s)$ constructs a set of new bindings between compatible pegs and holes in the system. Where there are multiple possible candidate bindings, they are generated in a random order and the first is selected.

3.2.5. Binding under Cardinality Constraints

A pipeline architecture can be formed by plugging together a set of components each of which provides and accepts the same service. To have a linear configuration, rather than a pipeline that may fork, requires restricting cardinalities so that each component can accept only one plugin (that will form part of the pipeline). With the application of a 1-1 binding policy, the pipeline is constructed as expected. Each component in the pipeline is connected to one upstream and one downstream component forming a linear chain. Figure 3.8 shows the pipeline (with the source on the left) growing as a third component is added. In this figure a Darwin [50] style notation has been used. White circles represent required (acceptable) services, and black circles provided services. In the Darwin language these are referred to as required and provided *ports*. Lines joining provided ports to required ports signify bindings.

However, somewhat unexpectedly, the intended structure does not result if the 1-n binding policy is applied instead. In this case binding may proceed as pictured in

Figure 3.9. The first addition has the same effect as with 1-1 binding, but with the next addition, the second the hole at the new end of the chain may be filled by the provision of the second component. The structure that results is: a) not the linear pipeline that was intended, b) unable to be extended with further plugins as all the requirements are bound - all the holes are full.

In order to guarantee that the desired structure is produced, another global constraint on the structure can be introduced. In this case it would state that there should be no cyclical paths between components[6]. Again, whether this constraint is imposed is a choice to be made by the deployer of a particular system.

Property 4 (No cycles in any $s : \mathcal{S}$)

$$p \subseteq \mathcal{C} \times \mathcal{C}$$
$$p = \{(c_1, c_2) | \exists b : \mathcal{B}.b = (c_1, _, c_2, _) \wedge b \in s.\mathsf{bindings}\}$$
$$\forall c : \mathcal{C}.(c, c) \notin p^*$$

3.2.6. Plugin Removal

The capability to model the removal of components is also required. In contrast to the addition function, it is not necessary to test whether a component has any particular ports before removing it. Any component that has previously been added can be removed. However, as well as removing the component in question, any bindings in which it was involved must also be removed. Once these bindings have been removed, the binding algorithm is re-applied so that bindings are formed between any pairs of components that may be now bound. Any components left isolated remain in the system, and may be reconnected the next time that the system is reconfigured.

Definition 14 (remove)

$$\mathsf{remove} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S}$$
$$\mathsf{remove}(s, c) = \mathsf{bind}(s') \Rightarrow c \in s.\mathsf{comps} \Rightarrow s' = \{ s.c \setminus \{c\}, s.\mathsf{bindings} \setminus \{x\}, s.\mathsf{start}\}$$
$$where\ x = \{(c', cl, c'', i) | (c = c' \vee c = c'') \wedge (c', cl, c'', i) \in s.\mathsf{bindings}\}$$

[6]The superscript $*$ in the defi nition of the property represents transitive closure

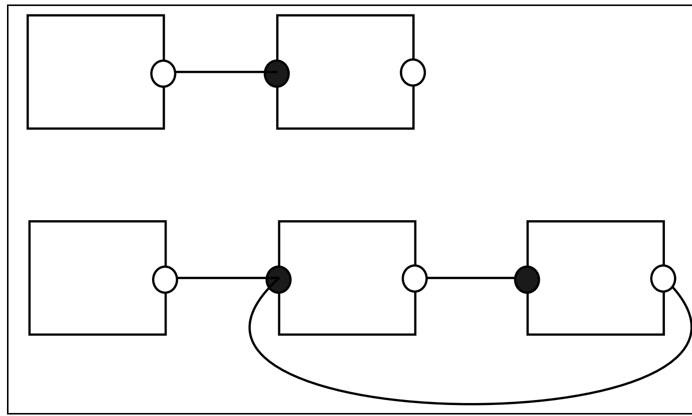

Figure 3.9.: Constructing a pipeline under 1-n binding

3.2.7. Plugin Replacement

It is desirable to be able to replace a component in a running application with a new one, possibly to perform maintenance by changing to an improved version of that component. Before replacing one component with another, it is important to check that the proposed replacement contains the functionality of the existing component.

A safe upgrade is one in which no functionality that was available before the upgrade is unavailable afterwards. In terms of provided and required services, the necessary condition is that the proposed replacement component provides at least the services that the one it is replacing does. However, it may be the case that not all of a plugin's provisions are being used in a particular system configuration. In this case, the check can be restricted to the provided services that are bound in the current configuration.

In plugin systems it is only the provisions that need to be considered, as requirements are always optional. The following predicate can be used to check whether a proposed replacement component has a provided service for every provided service of an existing component involved in a binding in the current configuration. If this is the case then the new component can safely replace the old one.

Definition 15 (canReplace)

$$\mathsf{canReplace} \subseteq \mathcal{C} \times \mathcal{C} \times \mathcal{S}$$

$$\begin{aligned}\mathsf{canReplace}(c, c', s) \iff \quad & (c \in s.\mathsf{comps}) \wedge (c' \notin s.\mathsf{comps}) \wedge \\ & \forall b.(b \in s.\mathsf{bindings} \wedge b.\mathsf{to} = c) \Rightarrow \\ & \quad \exists p : \mathcal{CL}.(p \in c'.\mathsf{comps} \wedge c \subseteq b.\mathsf{from})\end{aligned}$$

If the canReplace predicate holds, the replacement can proceed by removing the old component from the system, adding the new component into the set of components, and substituting the new one for the old one in any bindings where the old component was providing a service. As the new component may provide or accept more services than the old one, the binding function is applied again to create any new bindings that are now possible.

Definition 16 (doReplace)

$$\mathsf{doReplace} : \mathcal{C}, \mathcal{C}, \mathcal{S} \longrightarrow \mathcal{S}$$

$$\mathsf{doReplace}(c, c', s) = \{s.\mathsf{comps} \setminus \{c\} \cup \{c'\}, b', s.\mathsf{start}\}$$

$$such\ that$$

$$\begin{aligned}b' = & \{b_1 | b_1 \in s.\mathsf{bindings} \wedge b_1.\mathsf{to} \neq c\} \cup \\ & \{\mathsf{swapTo}(b_1, c') | b_1 \in s.\mathsf{bindings} \wedge b_1.\mathsf{to} = c\}\end{aligned}$$

$$\mathsf{swapTo}((t, p, f, h), c') = (c', p, f, h)$$

Definition 17 (replace)

$$\mathsf{replace} : (\mathcal{C}, \mathcal{C}, \mathcal{S}) \longrightarrow \mathcal{S}$$

$$\mathsf{replace}(c, c', s) = \begin{cases} \mathsf{bind}(\mathsf{doReplace}(c, c', s)) & \textbf{if } \mathsf{canReplace}(c, c', s) \\ s & otherwise \end{cases}$$

3.2.8. Developing the Model

The development of this model was aided by the use of the Alloy modelling and visualisation tools [42]. Part of the Alloy model appears in Appendix A. The Alloy analyser was used to verify that the specification does not contain inconsistent constraints.

Alloy is a lightweight notation that supports the description of systems that have relational structures. The systems that are described by this model are concerned with sets of linked components, so Alloy is a particularly appropriate language. The notation allows any first-order logical expression plus transitive closure to be written. In addition to providing language constructs that fit the domain, Alloy has the advantage that specifications can be analysed automatically. Analysis is supported by the Alloy Constraint Analyser (ACA) [43].

Given a specification, a system of constraints, the Alloy tool uses a SAT solver to try and find an example system that satisfies the constraints. In the case of inconsistent constraints, the analyser will report that it could not generate an example that satisfies the constraints specified. This indicates that it would not be possible to construct a system corresponding to this model. For example, initially we stated that there should be no components in the system that were not bound to other components, so that no "orphan" components could exist, but as a result of analysis realised that this meant that systems with only one component were not valid. This made it impossible to construct systems by plugging new components into an initial core component, as the system with just the core component would never be valid. The constraints therefore had to be relaxed to allow this case.

Where the constraints can be satisfied, Alloy generates a witness, which can be examined textually or graphically. This can lead to example situations (configurations of components *etc.*) that may not previously have been considered, being generated. For example, with early models we quickly found that witnesses often featured components bound to themselves, fulfilling their own requirements. This was not a desirable situation and so constraints had to be added in order to prohibit it.

The ACA tool provides a visualiser that 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 chapter 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 parts of the model to be omitted if desired, and showing labels either within an object or with an arrow from the object. In Figure 3.5 both techniques have been used, to add clarity.

Figure 3.7 was produced from the Alloy version of the model including numbers[7]. The figure shows an application extended by a chain of components, as in the second example in Figure 3.1. Where n : 0 appears in the diagram it means that no more classes can be bound to this interface and n : T indicates an infinite (unlimited) cardinality, so any number of classes can still be bound to this interface.

The model presented is a general representation of all possible configurations of components that we wish to consider as plugin systems. It includes the concepts of interfaces and classes implementing those interfaces, but does not mention any particular classes. If it were possible to create a more specific model, for a particular configuration of components, it might be possible to use Alloy's SAT solver engine to generate possible new configurations when a new component is added. However, with the current technology available, it is very difficult to provide Alloy with an exact situation from which the constraint solving algorithm should start. As we have seen, the constraints tend to talk about the world as a whole. It is difficult to specify that there are, for instance, four components that have particular interfaces and are bound in a particular way. An interesting future avenue of work might be trying to generate suitable sets of constraints that represent a particular configuration, while still allowing the constraint solver the flexibility to find possible alternatives for new bindings. However, this has not been pursued in this thesis.

3.3. Summary

By creating a formal model of plugin systems, we have defined precisely what we consider plugin systems to be and how they can be configured and assembled. The model was presented in the form of a set of logical definitions and constraints. Various design decisions were discussed, including what we consider must be true for all plugin systems, for example that all classes must be deployed as part of an enclosing component, and what may be altered by the deployment engineer dependent on the particular situation in which plugins are being deployed, for example the particular

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

binding policy that is selected. The model forms the basis for the work presented in the next chapters on programming using systems of plugins (Chapter 4), and implementing a platform that supports the use of plugin components (Chapter 6).

4. Programming with Plugins

In Chapter 3 we defined an abstract model of plugin systems, describing components and how they can be combined to form systems. In this chapter we explain a developer's view of our plugin model, as realised in a practical programming language. Examples are given showing how the concepts of plugin components, provisions and requirements presented in the previous chapter may be expressed in an object-oriented programming language. We show how the features of an object-oriented language may be used to organise software into sets of components that may be assembled in accordance with the model. The examples presented here are in Java. We are not concerned in this chapter with the implementation of the runtime framework that will support the management of plugins, this is discussed in Chapter 6, only with what concerns a programmer using plugins to develop software.

4.1. Components

In the previous chapter, a component was presented as being a unit of encapsulation and composition, a bundle of resources (code and data) that together may provide certain functionality. It is the code inside the component that provides its functionality, the component itself is just a container, a mechanism for packaging together sets of resources. The standard mechanism for packaging Java classes and other resources together into a deployable unit is to form a Jar archive. This is a compressed file that can aggregate a number of resources, and may be read (and decompressed) by the standard Java runtime system.

In some component models, components are provided as completely black boxes,

but come together with a set of meta-data that describe the interfaces through which the component may interact with other components.

In order to try and minimise the programmer effort required, the approach taken here is not to provide a meta-data description, but to allow the contents of the component to be inspected by the component framework. The provisions and requirements will be deduced from the code within the component. The manner in which provided and required interfaces manifest themselves in code is explained in the following sections.

Also aiming to minimise the extra effort required by the developer using the plugin architecture, as much of the management and infrastructure code as possible has been factored into the plugin framework. To reduce the components' dependence on their environment (and hence attempting to increase their possible re-use), the third-party binding model is used [66]. Components do not try to discover and bind to other components. All connections between components are established by a third party, the plugin framework.

4.2. Provisions

A plugin is considered to provide a particular service if it contains a class that provides an implementation of that service. In Java a class signifies that it provides an implementation of a service by declaring that it implements a particular Java interface. The example in Listing 4.1 shows a class and interface declaration. Compiling the interface and the class and packaging them into a Jar archive would form a component that might be added as an extension to any application where it is desirable that certain events are logged. From here on this component will be called Logger.

Making such functionality a plugin means that it is easy to switch logging on and off, or to switch between different types of logging dynamically at runtime without interrupting execution of the application. The logging example is often used by the Aspect-Oriented Programming [21] community. Using aspects, code can be woven into a program to execute before or after a particular method; for example, after every database query, code is run to log that query. This means that the need for logging need not necessarily be anticipated at design time. However, in order to add or remove

an aspect such as logging, the source code of the system needs to be recompiled, and execution stopped and restarted. Even using plugins logging cannot be introduced dynamically if the need for it was never anticipated. A call to the logger must be present in the code of the requiring component so that it may be executed when an appropriate provider is present. However, whether or not logging is performed, or which component performs it can be changed dynamically during execution.

The name of a component is represented as the name of the Jar file, and is therefore not mentioned in the source code of any of the classes. The `FileLogger` class implements the `Log` interface. In the `write()` method it timestamps each log message and records it by writing it out to a file. The modifier `public` is used for both the interface and class declarations, as these entities are intended to be used by other components. The Logger component can be plugged in to any application that can accept a `Log` as a plugin.

```
public interface Log {
   public void write( String msg );
   public void close();
}

public class FileLogger implements Log {
   FileOutputStream logfile;
   ...
   public void write( String msg ) {
      logfile.println( currentTime() + msg );
   }
   ...
}
```

Listing 4.1: Definition and Implementation of Log interface

4.3. Requirements

A component that can accept a plugin has a slightly more complex design than one that only provides services. Here we show how part of a typical e-commerce application might be written so that it can accept a `Log` and use it to record the occurrence of events.

```
class BookShopServer {
   Log systemLog;
   BookShopServer() {
      PluginManager.getInstance().addObserver( this );
   }
   void pluginAdded( Log plugin ) {
      systemLog = plugin;
   }
   ...
   void placeOrder( Book b ) {
      ...
      database.execute( ''INSERT INTO orders '' + getDetails( b ) );
      if ( systemLog != null ) {
         systemLog.write( ''Order placed for '' + b );
      }
      ...
   }
}
```

Listing 4.2: Simple server that can accept a Log plugin

For a component to use the services provided by a plugin, it must obtain a reference to an object from the plugin. As a third-party binding mechanism is being used, *i.e.* all bindings are created by the plugin framework, this is achieved through a notification mechanism, based on the Observer design pattern [31]. Any object can register with the platform to be notified when a new binding is made that is relevant to it.

An object calls the following method in the `PluginManager` class, which is part of the framework, to register as an observer. This is shown in context in Listing 4.2.

```
PluginManager.getInstance().addObserver( this );
```

This only registers that an object is interested in new plugins, but does not specify the plugin type. An object signifies that it can accept a plugin of a certain type by declaring a method `pluginAdded(...)` that takes a parameter of that type, in this case `Log`. Classes can define multiple `pluginAdded()` methods with different parameter types, so that they can accept several different plugins of different types.

When the Logger plugin is added, observing objects with a `pluginAdded()` method that can take a `Log` as their parameter are called by the framework, passing a reference to the newly available `Log` object, through which methods can be called.

```
public interface GraphicsTool {
   public Icon getIcon();
   public void draw( int x, int y, Canvas c );
}

public class AirBrush implements GraphicsTool {
   ...
   public void draw( int x, int y, Canvas c ) {
      //implement drawing code here
   }
   public Icon getIcon() {
      return icon;
   }
}
```

Listing 4.3: Definition and Implementation of GraphicsTool interface

The parameter will no longer be in scope after the end of the `pluginAdded()` method, so in order to maintain a reference to this object, as is normal object-oriented programming practice, this reference is assigned to a field of the object. Calls to the plugin object are enclosed in a conditional block with the condition `if ( systemLog != null )`, as the field will be null until the plugin is added, and calling a method on a field that is null will cause an exception to be thrown.

Here the class `BookShopServer` has a field `systemLog` of type `Log`. It also has a `pluginAdded()` method that takes a `Log` as a parameter. Inside this method the parameter is assigned to the field. In the `placeOrder()` method, if a value has been assigned to the `systemLog` field then the plugin is called, recording the order.

It is the presence of a field in a class, together with a `pluginAdded()` method that assigns to that field, that is used to signify a requirement. If the presence of a suitable method alone were used, it would not be possible to determine how many times that method could be called before all of the requirements were fulfilled. The assignment to a scalar field indicates that this is a hole with a cardinality of one. Once the method has been called and the assignment made, the hole is full.

It will be noted that it is much more difficult to extract information about the required services from a component than the provided services. It is necessary to look for names of fields, or examine the body of the `pluginAdded()` method, rather

than simply finding the type of the class. A trait of object-oriented programming is that objects typically declare the methods that they provide (and may exhibit these through a type), but not those that they use from other objects.

4.3.1. Accepting multiple plugins

An application is likely to use only one logging source, but there are times when multiple plugins would be useful. Consider a component that can accept many plugins of a particular type, as specified in our model in the previous chapter, for example a simple paint program that can be extended by plugging in a number of different drawing tools. A Brush component might comprise the following class and interface. The `GraphicsTool` interface states that all `GraphicsTools` must be able to provide an icon, and be able to draw on a given canvas (see Listing 4.3).

The paint program that uses this plugin keeps a list of all the currently available tools. Whenever a new tool is added, it is added to the list. Whenever the window is redrawn, the program iterates through all of the available tools to draw their toolbar buttons (see Listing 4.4).

```
class PaintProgram {
   List tools;
   GraphicsTool current;
   PaintProgram() {
      tools = new ArrayList();
      PluginManager.getInstance().addObserver( this );
   }
   void pluginAdded( GraphicsTool plugin ) {
      tools.add( plugin );
   }
   void redrawWindow() {
      ...
      for ( Iterator i = tools.iterator() ; i.hasNext(); ) {
         GraphicsTool gt = (GraphicsTool)i.next();
         drawButton( gt.getIcon() );
      }
   }
   ...
}
```

Listing 4.4: Simple paint program that can accept multiple GraphicsTools

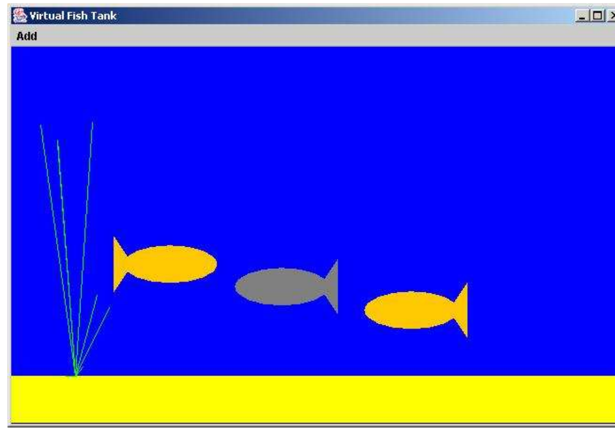

Figure 4.1.: The Virtual Fish Tank application

In this case, the reference to the plugin that is added is not assigned to a scalar field, but is added to a List (one of the Java Collection classes). Calling the `pluginAdded()` method repeatedly with different objects will therefore not overwrite the previous plugin object that was added, but record the addition in the list. In this way a hole has been created that has no cardinality constraint upon it. New plugins of the `GraphicsTool` type can always be added.

4.3.2. More complex configurations

The Virtual Fish Tank is an example application demonstrating the use of plugins. The basic application displays an uninhabited fish tank on the user's screen. Over time, different inhabitants can be added to the tank (see Figure 4.1). These inhabitants are supplied in the form of plugin components.

Initially, the system starts off with only the Tank component. In order to be added to the tank, a prospective inhabitant must have a class that implements the following interface (in terms of the jigsaw analogy, Tank has a hole with a shape defined by this interface):

```
public interface Inhabitant {
  public void move();
  public void draw();
}
```

All `Inhabitants` can therefore be asked to move themselves, and be asked to draw themselves on the screen. Figure 4.2 shows a picture of an example configuration of components in the Virtual Fish Tank, produced using the Alloy Constraint Analyser's visualiser (with some editing of labels for clarity).

It is possible to add a Weed to the tank. A Weed component comprises only one class (but it is still enclosed in a Jar file), which knows how to draw a weed, and when asked to move will do nothing. This class provides the peg that allows the Weed component to connect to the Tank. It also provides an implementation of the Food interface, meaning that possible consumers can gain sustenance by eating the weed.

```
public interface Food {
   public void eat();
}

public class Weed implements Inhabitant, Food {
   public void move() { /* do nothing */ }
   public void draw() { /* draw weed */ }
   public void eat() { /* get smaller */ }
}
```

Listing 4.5: Code for the Weed component

Another possible inhabitant for the tank is the Goldfish. Goldfish is a component comprising two classes:

```
class Fish {
  void draw() { ... }
}

public class GoldFish extends Fish
                 implements Inhabitant, Food {
  Color getColor() { ... }
  public void move() { ... }
  public void eat() { ... }
}
```

Listing 4.6: Code for the Goldfish component

This component is implemented according to the template method pattern [31]. Behaviour common to all types of fish is defined in the superclass, with the subclass

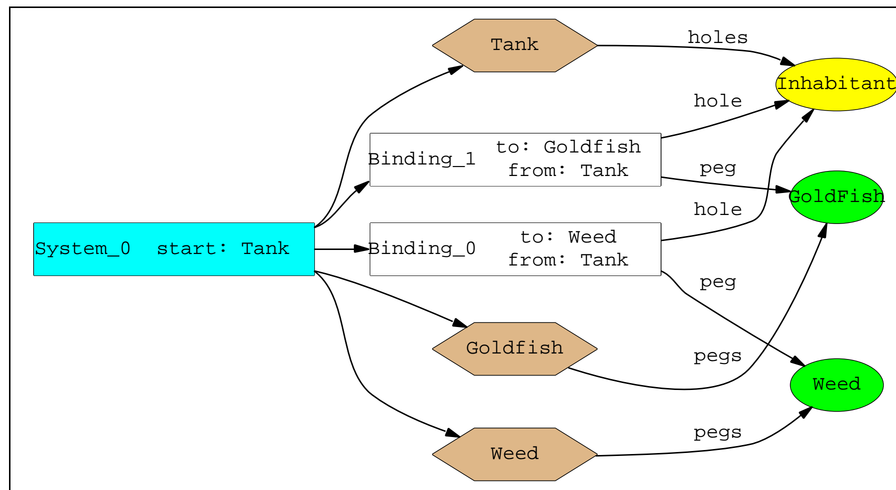

Figure 4.2.: Alloy diagram representing fish tank application

providing the detail specific to Goldfish about its colour and how it moves. The `GoldFish` class provides the peg to fit in an `Inhabitant` hole, and also provides an implementation of Food, as it is possible that the fish may also be eaten.

The last type of component that will be considered is a predator. The Hungryfish is implemented in a similar way to the Goldfish (it has a different colour, and moves slightly faster than the Goldfish). The Hungryfish can also maintain a reference (in a scalar field) to the `Food` that is its prey at a particular time. In fact, the `Fish` class in both the Hungryfish and Goldfish components is identical (the differences occur in the subclasses), however it is still necessary to include `Fish` in both components. It is not known which of the two will be added first (if they are added at all) and so each component must independently provide all of the resources it needs in order to function.

Figure 4.3 shows a situation in which a Hungryfish, which is both an `Inhabitant` and can accept `Food` is about to be added to the Tank. The Hungryfish will be bound to the Tank by its `Inhabitant` interface. It will also be bound to one of the other inhabitants, which it sees as food, by its `Food` interface. In the situation shown, which inhabitant the Hungryfish will be bound to is not clear. There are two possibilities, the Hungryfish can eat either the Weed or the Goldfish.

Which binding is made could be decided by making a random selection. However,

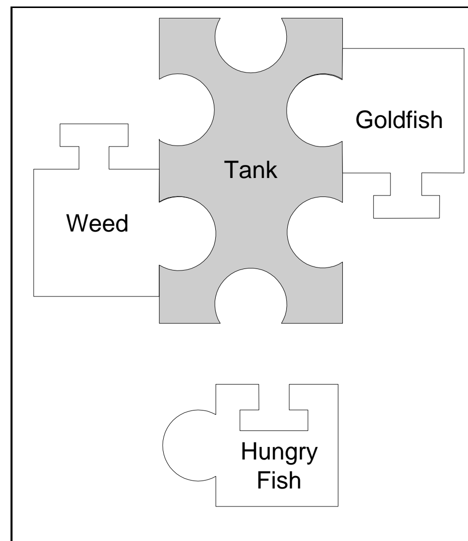

Figure 4.3.: Adding a Predator to the fish tank

it may be desirable to use a more sophisticated mechanism to make the choice. The next chapter will examine how constraints may be defined that inform or govern the binding behaviour.

4.3.3. Null Objects

Returning to the logging example, at the part of the program where the service provided by the plugin is actually called, the following line is present:

```
if ( systemLog != null ) {
   systemLog.write( ``Order placed for '' + b );
}
```

The test for `null` is there to ensure that a method is not called on a null reference (which in Java would result in a NullPointerException being thrown at runtime). Having to include this code is a side-effect of using the third-party binding mechanism. A technique for avoiding this is to use a NullObject to act as a placeholder. A NullObject provides an empty implementation of an interface. The methods can be called but will do nothing. Hence the BookShopServer's code could be updated to that shown in Listing 4.3.3:

Here the empty implementation of the `GraphicsTool` interface is included in the component. Rather than leaving this as a somewhat tiresome task for the developer,

```
class BookShopServer {
   Log systemLog;
   BookShopServer() {
      systemLog = new NullLog();
      PluginManager.getInstance().addObserver( this );
   }
   void pluginAdded( Log systemLog ) {
      systemLog = plugin;
   }
   ...
   void placeOrder( Book b ) {
      ...
      database.execute( ``INSERT INTO orders '' + getDetails( b ) );
      systemLog.write( ``Order placed for '' + b );
      ...
   }
}

class NullLog implements Log {
   public void write( String msg ) {}
}
```

Listing 4.7: BookShopServer with NullLog

NullObjects can be generated dynamically for a given interface. Details of how this may be achieved are given in Appendix B.

4.4. Plugin Removal

It is not possible for a component to free up a hole just by discarding its reference to a plugin. As it is the plugin framework that creates bindings, it must also be the framework that removes bindings, in order that the framework always maintains an accurate set of the connections in the system. If a component nullifies a field, releasing a reference to a plugin, then although that component can no longer access that plugin, from the point of view of the framework, the configuration has not changed. In order to effect a change in the configuration, it must be the framework that initiates the breaking of the connection between two components.

Components are notified of the imminent removal of a plugin using the same

Observer mechanism as is used during plugin addition. The framework will call a `pluginRemoved()` method that has the appropriate parameter type in an observing object. It will pass a reference to the object that is about to be removed as a parameter. On notification, the component should clean up any references that it holds to this object. For instance, in the Logging example:

```
class BookShopServer {
   Log systemLog;
   ...
   void pluginAdded( Log plugin ) {
      systemLog = plugin;
   }
   void pluginRemoved( Log plugin ) {
      systemLog = new NullLog();
   }
}
```

Here the `systemLog` field is nullified in the `pluginRemoved()` method. After this method completes, the framework can remove the plugin. The programmer can assume that the plugin will still be available during the execution of the body of the `pluginRemoved()` method, and so can, and should, call any methods on the plugin that are required to clean up before it is removed. For example it might be desirable to close the log, or even log the fact that the log is being removed. In the paint program example the `pluginRemoved()` methods would simply call `tools.remove( plugin )` to remove the tool from the list of currently available tools. After the relevant `pluginRemoved()` method has been called, a scalar hole is considered to be empty again.

```
class BookShopServer {
   Log systemLog;
   ...
   void pluginRemoved( Log plugin ) {
      systemLog.write( ``Log plugin removed.'')
      systemLog.close();
      systemLog = null;
   }
   ...
}
```

4.5. Replacement

The replacement of plugins is something that the programmer should not have to consider. If the BookShopServer is using a Logger component, and that component is upgraded to a newer version, or switched for a logging component that will send log messages by email rather than writing to a file, then this should not affect the server. As all logging plugins implement the same `Log` interface, which particular implementation is being used at any time is not important to the server, and it should be possible to interchange the different logging plugins transparently.

Because of this, the fact that it is desirable to be able to replace plugins with different versions dynamically should not manifest itself in the source code written by the developer using the plugin framework. Details of how replacement can be performed are discussed in Chapter 6, but as these should not affect anyone developing with plugins, they will not be discussed further here.

4.6. Summary

This chapter has given several concrete examples of programs that have been designed to use plugin components. The examples given are in the Java programming language and follow the model presented in the previous chapter. These examples show how a system of plugin components can be used by a developer to create a dynamically extensible or reconfigurable system.

The main contribution of the approach to programming with plugins presented here is that the notions of provided and required (or *accepted*) services have been mapped to idioms familiar to object-oriented programmers, namely interface implementation and callback method parameters. This is in contrast to approaches such as those taken by Avalon [2] or Eclipse [60] where the provided and required interfaces are defined by various meta-data. We do not need to introduce these different data formats as all the information is expressed in the source code itself. The programmer does not need to swap between different data representations.

A trait of object-oriented programming is that it easy for classes to express the

services that they provide, through their public methods and interfaces. It is however much more difficult to determine the services that a component requires, as these are not typically exhibited through types. We have used the types of parameters in callback methods to specify required interfaces. This is in contrast to approaches such as those used by OSGi [64] or Avalon where a string is used to name the services that a client requires.

In OSGi, and similarly with ActiveX [55], each plugin component must implement a particular interface in order to be called back by the plugin framework. While this does provide some static assurance of compatibility between user components and the supporting framework, in order for the implemented interfaces to work with any component, they must be very general. This commonly means that any reference received from the supporting framework must have a very general type (`Object` in Java) and be cast to a more specific subtype before it is used.

We use a reflective mechanism to allow method declarations to be typed more specifically. We do not require the implementation of a specific interface, but rather the adoption of a convention. The runtime framework uses reflection to dynamically detect which methods it should call by matching parameter types. This allows the programmer to write callback methods whose parameter types match the types that they wish to use in their program. This removes the need for ugly casting. The parameter types of callback methods then also indicate the services required from other components. By using Java types rather than strings to identify interface names and types the code is cleaner and remains within the standard Java programming style. The Java compiler can also use the type information to give greater assurance of the correctness of programs statically.

In the case that plugins must extend a particular class, *e.g.* in the PluggableComponent system [78], the use of a plugin mechanism cannot be introduced to a class that is already part of an inheritance hierarchy (in a single inheritance model at least). Also, any class that provides a service as a plugin cannot be used in an environment that does not involve the plugin framework, as in order to load the class, all of its superclasses and interfaces must also be loaded. We avoid these restrictions by allowing any class to be used as a plugin that provides a service. We do not require that it forms part of

the plugin framework's type hierarchy.

We showed the programming idioms to which developers need to adhere to create plugin systems according to our model. Mechanisms were described for the addition and removal of components. The additional code that a developer has to write in order to use our plugin system is minimal: one line to register an interest with the plugin platform, and one short callback method for each type of plugin that can be accepted by the current component. Replacement of components should not have an effect on the source code of the application, as it should occur transparently. Considering the replacement of components should be the responsibility of the deployment engineer, not the developer of individual components.

The style of programming shown in the examples in this chapter uses a third-party binding paradigm. This means that the client does not explicitly ask for a component that will provide it with a service, rather, when one becomes available, the client will be notified of this by a third-party, the runtime framework. One consequence of this is that the client programmer does not have explicit control over when components are connected or disconnected. It is not possible for user code to trigger a reconfiguration of the system. Although this may seem to be a limitation in some cases, for example if the application is monitoring its own performance in some way, we believe that encapsulating coordination and reconfiguration functionality in a lower layer of the system makes user level code cleaner and more maintainable. Organising the system this way reduces duplication of common code, for example that used to query a service registry, among components.

In order to support the development of plugin systems in the style presented, some sort of runtime plugin framework is required. We require that the framework can inspect the code of components to detect which interfaces they implement, *i.e.* which services they provide, and to detect the occurrence of `pluginAdded()` methods, extracting their parameter types to identify required services. The framework must be able to match provided and required services and create and manage bindings between components when appropriate pairs are found. The implementation of such a plugin framework is discussed fully in Chapter 6.

5. Constraining and Analysing Plugin Systems

The model that we have presented for writing and assembling software systems based around plugin components allows the formation of many different systems, either by combining different sets of components, or, for a given set of components, binding them together in different ways to form different configurations. Having such a large set of possibilities can lead to systems being formed that do not behave in a desirable way. In order to prevent this, constraints need to be placed upon the systems assembled. In this chapter we discuss the sorts of constraints that may be desirable to place on systems, and how they might be specified and enforced[1].

5.1. Deployment

Systems created from plugin components may be deployed in stages, adding functionality as and when it becomes needed and available, by adding or replacing components over time. Components may be acquired from many different developers or vendors. The plugin model makes it possible for system administrators to add components to a system without their developers having to write additional code, and so the issue of configuration is out of the hands of the software developer. This also means that a developer, in writing a new component or a new version of an existing one, cannot know the set of components that it will be interacting with. He also cannot know the

[1] Some of the material presented in this chapter formed the core of a paper published at the Fundamental Approaches to Software Engineering (FASE) 2004 conference [14].

configuration in which his component will be deployed, as different sites may provide very different environments.

This could easily lead to incorrect operation of a component or set of components. A component could be bound or called in a way not intended by the developer, and therefore cause malfunction in the system. If a system is started with a configuration that causes problems, then it may not be viable to put it into service. A more significant problem would be if a plugin were added dynamically to a running system, perhaps performing a critical service, and caused it to malfunction.

When deploying a software system, engineers desire a degree of assurance that it will function correctly. In some critical cases, having confidence that the system will work correctly is of the utmost importance. One way of obtaining this confidence would be to perform a trial deployment. Experiments can then be performed to test whether the system behaves as expected. This approach has a number of problems. In order to be sure that the software will operate correctly in the environment in which it will finally be deployed, the trial deployment must be done in an environment that simulates the final deployment environment exactly. With large systems and distributed deployment environments this is difficult to achieve. In cases where there is an existing system that is to be upgraded without halting execution, the trial must be run alongside the original, meaning that significant hardware and software resources need to be duplicated. This may be costly.

An alternative is to build a model of the proposed system and test that instead. As it does not incur the cost of constructing the full system, modelling should provide a more pragmatic solution. Building and testing models before performing the final deployment is common in other engineering disciplines. There is a large body of research work dealing with formal modelling and analysis of systems. However, formal methods are generally considered to be “hard”, and so modelling of software systems is a specialist activity. We propose techniques for automatically constructing and analysing models of plugin systems.

While discovering that a system does not behave as expected is informative, what is really needed is a way of using desired properties as a way of guiding the assembly of correctly functioning systems. In this sense, properties that are required to hold

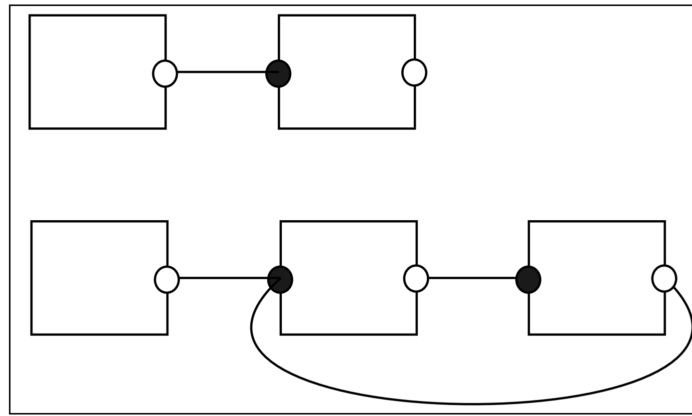

Figure 5.1.: Constructing a pipeline under 1-n binding

are used as constraints on a system, and only systems that meet the constraints may be assembled. There are two different types of constraints that may be considered. Structural constraints deal with the configuration of the system, how components are bound together and the shapes that result. The imposition of a particular *architectural style* can be seen as a structural constraint. Behavioural constraints deal with the actions that can occur in a system. They can enforce a particular protocol, or ensure that a particular sequence of actions is never executed. Enforcing either of these types of constraints requires a formal description of both the constraint and system to which it is applied. In this chapter we will discuss techniques for applying both structural and behavioural constraints to plugin systems.

5.2. Structure

In the plugin model presented in Chapter 3, the binding process matches only the types of interfaces that are discovered to be provided or required by each of the components added to the system. If a provided port is present with the same interface type as a required port (and they do not both belong to the same component) then a binding can be created between them.

This leaves considerable scope for different configurations to be formed. We may want to enforce compliance with a particular architectural style (for example a pipeline,

or a client-server configuration). To do this we need to be able to constrain the structure in some way.

In Chapter 3 we discussed two different binding policies, 1-1 and 1-n, the application of which resulted in different systems being formed from the same sets of components. Under one of these policies, the system formed was not what was desired. The intention was to construct an extensible pipeline, but employing the 1-n binding policy, a non-linear, non-extensible system resulted, as shown in Figure 5.1. To cure this, a further property was defined, specifying that there should be no cycles in the bindings formed.

These binding policies and properties can be generalised into sets of constraints on the structure of systems. When a reconfiguration is to be applied to a system, through the addition, removal or replacement of a component and any subsequent rebinding, candidate configurations can be checked against the set of constraints currently imposed, to see if the configuration is allowable. If not, then an alternative configuration can be generated and checked. Only when a configuration has been found that meets the specified constraints should changes in binding actually be carried out.

5.2.1. Representing static architectures

At any point during the evolution of a dynamic component based system, the system will have a particular configuration. In order to check whether particular constraints are satisfied, we need a way of representing the current configuration, or the configuration that would be present after a proposed reconfiguration is carried out. The representation needs to be capable of capturing all of the concepts presented so far in the plugin model: components, provided and required services and bindings between them. There are many languages that could be used to specify particular configurations of plugin components expressed in terms of these concepts, for example C2 [72], Darwin [50], Rapide [48] and Wright [1]. We have chosen to use the Darwin architecture description language, developed at Imperial, as it is the language with which we are most familiar, and it has relevant tool support which can be adapted as necessary. Darwin will be

used from here on to describe the structural aspects of component configurations. We will describe how our plugin concepts map to Darwin and discuss possibilities for specifying structural constraints.

5.2.2. Introduction to Darwin

The Darwin architecture description language [50] can be used for specifying the structure of component based and distributed systems. Darwin describes a system in terms of components that manage the implementation of services. Components provide services to and require services from other components through ports. The structure of composite components and systems is specified through bindings between the services required and provided by different component instances. Darwin has both a textual and a complementary graphical form, with appropriate tool support.

Darwin structural descriptions can be used as a framework for behavioural analysis. Darwin has been designed to be sufficiently abstract as to support multiple views, two of which are the behavioural view (for behavioural analysis) and the service view (for construction). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behavioural specification or service implementation.

Components

The basic building block for systems described in Darwin is the component. Components may provide and require particular services through their exposed interfaces, and may also compose instances of other components to create larger composite components. A component description is similar to the notion of a class in object-oriented programming, in that it can be instantiated to create several instances of the same component.

Ports

Components can expose particular interfaces through which they can provide and require services. Ports are declared as having a particular interface type (defining the services that can be accessed through that port) and as being either provided ports or

required ports. Ports can be named so that they may be referenced in other parts of the architecture description (for example in bindings). *Portals* are ports that are not specified as being either provided or required. These are typically used for forwarding ports from inner components of a composite component to expose them as ports of the composite component. As well as simple scalar ports, arrays of ports of a certain type may be defined. These are referenced using a name and an integer index.

Interfaces

Interfaces are used to define types for ports. They define sets of actions which may be accessed through a particular port. To this extent, an interface is simply a set of action names. The interface itself is given a name so that it may be referenced elsewhere in the description.

Bindings

Two instances of components may be bound together by a particular pair of ports. This is achieved by specifying a binding. All bindings must be specified inside a component definition, so it is usual to create one large component that composes instances of all of the other components in the system. Bindings may be specified either between explicitly named ports (from required to provided), or just to a particular port type if this is sufficient to distinguish the port (*i.e.* if the component in question has only one port of a particular type).

In the example shown in Listing 5.1, the component Server provides a service Shop, through which the three actions *login*, *pay* and *download* may be performed as given in the definition of the Shop interface. The component Client uses the Shop service through a required port named *serv*. The System comprises one instance of both Server and Client (s and c respectively), with a binding between the provided port of s and the required port of c.

All of the systems of plugin components discussed so far can be described using the Darwin language.

```
interface Shop { login; pay; download; }
component Server {
   provide Shop;
}

component Client {
   require serv:Shop;
}

component System {
   inst s:Server;
        c:Client;
   bind c.serv -- s.Shop;
}
```

Listing 5.1: Darwin representation of client-server system

5.2.3. Specifying structural constraints

Once we have a representation of a particular configuration of components, we would like to check whether that particular configuration meets certain constraints. Georgiadis [36] uses the Alloy [42] language[2] to specify structural constraints for self-organising component systems. For example, in specifying that the pipeline architectural style should be followed, the following constraints are given:

- `all c:Comp | (one c.prov) && (one c.req)`

  Every component c in the pipeline has one provision and one requirement.

- `all c:Comp | sole c.prov.provBind`

  Cardinality constraint: each provided port has at most one required port bound to it and vice-versa.

- `some c:Comp | c.*connected = Comp`

  The set of components reachable (transitively through bindings) from each component is the set of all components, *i.e.* all components form a single chain, not several disjoint chains.

[2]Georgiadis uses the version of Alloy that was current in 2002. The syntax has changed markedly since then, but the ideas behind the language remain the same.

- `some leftEnd:Comp | no (leftEnd.prov.bind.~port & Comp)`
  The pipeline does not form a ring, *i.e.* the provided port of the left end of the pipeline should not be connected to any component of the pipeline chain. The left end can be unbound.

This defines a "closed" architecture, where all of the components in the system must be part of the same pipeline, and must all have exactly one provided and one required port. A more useful concept is that of an "open" architecture, which Georgiadis goes on to explain, where components that are to be used in the pipeline must have at least one provided and one required port (to allow them to be connected into the pipeline) but may have other ports, allowing components to have ports not concerned with pipeline communication, and to connect to components that are not part of the pipeline. Open architectures also permit multiple different architectural styles, and associated constraints, to co-exist within one system, so a pipeline may contain one component that is also part of a ring architecture. The system of constraints reproduced above is made more sophisticated to allow for open architectures.

The Alloy specifications needed to enforce these structural constraints are quite detailed and complex. Constraints must be specified in terms of an Alloy model of Darwin, rather than directly in terms of Darwin constructs. A higher level constraint language would be more expressive and more conducive to writing structural constraints pertaining to component systems. Work on the language Darwin^i is aiming to address this.

5.2.4. Darwin^i

Darwin^i [80] is a language for specifying structural constraints for different styles of architecture in terms of the Darwin language. "The Darwin^i specification is not a specification of a specific 'snapshot' of the running system, but it is a specification of an architectural pattern or style, describing component types and the allowable interconnections of their instances."[80].

The fourth Alloy constraint reproduced above, stating that in a pipeline architecture the components may not form a ring, can be written in Darwin^i in the following way.

```
forall f in Filter { !bind f.p ~~ f.r; }
```

This constraint states that it is not permitted to be able to follow bindings transitively from a provided port of a certain component instance in the pipeline (where all components in the pipeline are taken to be of type Filter) and eventually reach a required port on the same component instance.

There are currently some limitations with the Darwin^i language. In the above example, `Filter` is defined as a particular component type, with one provided and one required port of the same type. It cannot be parameterised, or used as a supertype for other components that have additional ports to those that are required to make it a filter. This means that it is currently not possible to describe open architectures with Darwin^i. Also, tool support for the language is only now being developed. Hopefully, when this work is completed it will be possible to use the Darwin^i language to specify and impose structural constraints on our plugin systems.

5.3. Behaviour

We have discussed ways of constraining the structure of component systems so that they adhere to certain architectural styles. However, this is not necessarily sufficient to ensure that assemblies of components will behave in a desirable fashion.

A consequence of taking the view that two ports having the same interface type is sufficient for binding them together is that types define a set of actions or methods that must be available, but not what should happen when those methods are called, or in which order they are supposed to be called.

This means that two components may be bound that expose the same interface, but expect a different protocol. Therefore, components with compatible interfaces may not in fact be compatible in terms of their behaviour. To ensure correct operation, it is necessary to be able to specify behavioural constraints as well as structural constraints.

Again, in order to impose constraints on a system's behaviour, it is necessary to be able to represent that behaviour in some sort of model. Darwin's structural view can be enriched with a behavioural specification for each component. Focussing on the behavioural view, we can use a simple process algebra - Finite State Processes (FSP) [51] - to specify behaviour. A complete system specification can be written by using the same action names in the behavioural specification as in the Darwin service descriptions (the actions named in the interface definitions). These specifications can be translated into Labelled Transition Systems (LTS) for analysis purposes. Analysis is supported by the Labelled Transition System Analyser (LTSA) tool.

5.3.1. Introduction to FSP

As described in Kramer and Magee's book [51], FSP stands for Finite State Processes. It is a simple process calculus. FSP specifications generate finite Labelled Transition Systems. The syntax and semantics owe much to both Hoare's CSP and Milner's CCS. FSP behaviour specifications contain two sorts of process definitions: primitive processes and composite processes. Safety properties are specified using property automata and liveness properties using Buchi automata.

5.3.2. Specifying behavioural constraints

Behavioural constraints can be specified in terms of sequences of actions that are desirable or undesirable. Here we use the FSP process calculus to specify such properties.

In the client-server system that was introduced with the previous example, the server might have behaviour that could be modelled in the following way:

```
Server = ( { login , pay , download } -> Server ).
```

This indicates that any of the three actions (*search*, *pay* or *download*) may be performed at any time. After the completion of any of these actions the server returns to its initial state. The client's behaviour could be modelled similarly:

```
Client = ( login -> download -> pay -> Client ).
```

This expresses the sequence of actions that the client will call. Its behaviour is to log in, download an item, and then pay for it when successfully downloaded.

These two components could be plugged together to form a system, as they have matching interfaces, but it may be the case that the operator of the server does not want it to be possible to form a system that behaves this way. They may want to ensure that payment is made before a download is initiated, as although the client above will eventually settle its debts, it could quite easily be replaced with a less honest client that just logs in, downloads and never pays. Therefore, the server administrator wants to ensure that payment is made up front. We can specify this by means of the following FSP property:

```
property MoneyUpFront = ( pay -> download -> MoneyUpFront ).
```

Composing this property together with the models of the Client and the Server, a model checker can be used to exhaustively check the model for possible behaviours that violate the property. In this case, performing this check would reveal that the trace `login -> download` violates the property.

The system administrator would wish to check that the desired property holds before installing this client plugin to work with this server plugin. The client might work perfectly well with other servers that expected a cash-on-delivery protocol rather than a money-up-front protocol. Therefore, a model for this particular configuration needs to be generated and tested before the new plugin is bound to the server and begins operating. The following sections discuss in detail how such a model may be generated automatically for any configuration of components.

5.4. Matching plugin concepts with Darwin concepts

The plugin components being considered comprise collections of (Java) classes and interfaces bundled together in a Jar file (which may also contain other resources such as graphics or data files). The Java code for a basic filter plugin and the corresponding

Darwin description which is generated from it are shown in Listings 5.2 and 5.3. The Java code follows the outline introduced in Chapter 4 to provide a service and use a service provided by another component. The code for the class and interface would be compiled and packed into a Jar file, forming the plugin component.

For each Jar file loaded as a plugin, a corresponding `component` construct in Darwin is generated. The name of the component is generated from the name of the Jar file. In this case the name of the Jar file is BasicFilter.jar .

The Jar file may contain a number of class files representing interfaces. These are collections of methods that define types. These are equated with Darwin interface definitions. Darwin components communicate through ports, either provided or required. A Darwin interface definition defines a type that may be assigned to a port. Interfaces are defined in terms of sets of actions.

```
public interface Filter { public void data( String x ); }
public class FilterImpl implements Filter {
   Filter next;
   // constructor
   public FilterImpl() {
      PluginManager.getInstance().addObserver( this );
   }
   // implementation of Filter interface
   public void data( String x ) {
      if ( next != null ) { next.data( x ); }
   }
   public void pluginAdded( Filter f ) { next = f; }
}
```

Listing 5.2: Java code for Filter implementation

```
interface Filter { data; }
component BasicFilter {
    require next:Filter;
    provide Filter;
}
```

Listing 5.3: Equivalent Darwin

Some of the classes in the Jar file may be declared as implementing certain public interfaces. These are classes that provide services that can be used by other components. The inclusion of such a class in a plugin is equivalent to declaring a

Darwin component to have a provided port with the type named by the interface. Such a class may be instantiated several times, by a third party, to produce objects that provide this service. These objects, therefore, do not have explicit names, and so in the Darwin model provided ports are declared with only the type of the service provided.

Components can use services provided by other components. When a new plugin is added to a system, any component that accepts it needs to be able to call methods provided by that plugin in order to use it. In Darwin this corresponds to a required port. In Java, in order to be able to call methods on an object of a certain type, a reference to that object is required. The mechanism by which such a reference is acquired in the plugin system is as follows.

An object registers as an observer by calling the method `addObserver()` in the PluginManager class, which is part of the plugin platform. To be notified of new plugins, the object can define a number of `pluginAdded()` methods with different parameter types. When a new plugin is connected, the platform picks the relevant method and calls it, passing a reference to the object from the new component that provides the service. In the body of the `pluginAdded()` method this reference is assigned to a field of the appropriate type.

In the case of scalar requirements, where only one plugin of a certain type is to be accepted by a particular component at any one time, assignment of the reference to a scalar field is sufficient. In the case that multiple plugins of a certain type are to be accepted, some sort of data structure is required to record the references, such as a list or a set. We detect assignment to scalar fields and addition of references to data structures (anything that is a subtype of `java.util.Collection`) by reading and analysing the byte-code in the relevant component. In the Darwin description, either a scalar or an array port of the correct type is inserted accordingly.

In the example above, a name is included for the required port. It corresponds to the name of the field declared in the class, `next` in this case. This is necessary as it is possible for a component to have more than one required port of the same type, a component may accept multiple plugins of the same type. These could be assigned to different fields in the class, or added to an array. For example, a forking filter would forward data to two different downstream components, and so would accept, and keep

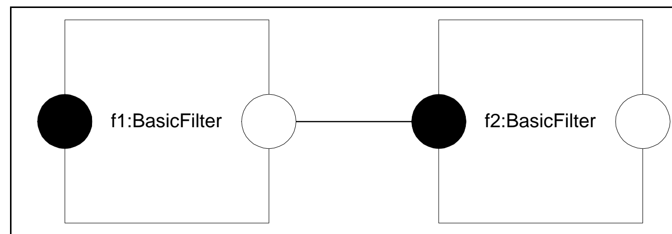

Figure 5.2.: Chain of two BasicFilters

references to, two plugins with the same interface. To find the names of fields from the compiled code of components, again the byte-code is analysed.

When constructing a system, configuration is performed at the level of components. A new Jar file is loaded to add a component. Any provided ports in the new component that match required ports in other components, or vice versa, are identified. The class that provides the service is instantiated by the plugin platform and any observers in the component requiring the service are notified, passing a reference to this new object. This process creates a binding between the two components. In Darwin terms, the complete system is modelled as a composite component, inside which are included instances of all the components involved in bindings. These instances are assigned arbitrary, but unique, names. A binding between each relevant pair of ports and components is also added. The following would be generated for a chain of two BasicFilters:

```
component System {
        inst f1:BasicFilter;
             f2:BasicFilter;
        bind f1.next -- f2.Filter;
}
```

The `inst` keyword begins the declaration of the two instances of type `BasicFilter`, `f1` and `f2`. The `bind` keyword begins a line stating that `f1`'s required port `next` is bound to `f2`'s provided port of type `Filter`. The equivalent graphical view is shown in Figure 5.2.

5.5. Analysing Behaviour

A simple example (introduced earlier) showing how these concepts might be extended to include behaviour is a client that is connected to an e-commerce server allowing it to purchase and download items. The Client component contains an interface `Shop`, declaring the methods `login()`, `pay()` and `download()`, and when notified of an object of this type will call these methods. The Server component contains a class that implements the `Shop` interface. The plugin framework can create a description of the interface and the two components in Darwin. Provided and required ports are declared with the appropriate types. In the example, the system as a whole comprises one instance each of the Client and Server components, with the two ports connected by a binding.

```
interface Shop { login; pay; download; }
component Server {
    provide Shop;
}
component Client {
    require serv:Shop;
}
component System {
    inst s:Server;
        c:Client;
    bind c.serv -- s.Shop;
}
```

The information in the Darwin description is purely structural. To analyse behavioural aspects of a plugin system requires a behavioural specification for each component. These can then be combined, in accordance with the given structure, to give a full system model.

We require component developers to provide an abstract description of the behaviour of their components. This could be provided separately from the component, however one of the ideas underpinning plugin technologies is that plugins should be deployed as single entities. It is therefore desirable not to have to provide the behavioural model separately from the component, but to include it within. Plugins can then form deployable units that include everything they need in order to be used.

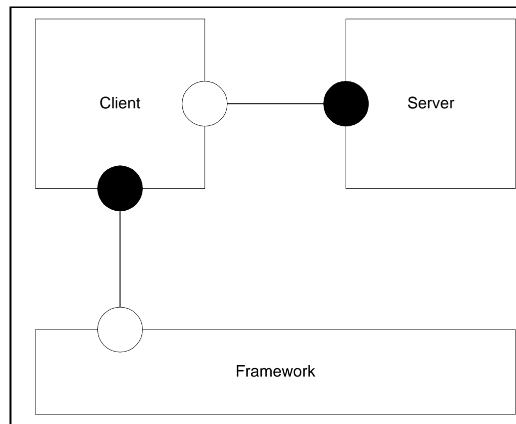

Figure 5.3.: Client provides FSPDefinition to the plugin framework

The approach taken here is to allow each component to provide a description of its own behaviour. The behaviour is described textually in the FSP process calculus [51]. When the framework generates a Darwin description of the current state of the system, it requests the FSP from any components that provide it, and includes this in the model.

This is achieved by allowing the plugin to provide another service which binds directly to the plugin framework (which is itself a component that can be connected in the same way that any other in the system can), see Figure 5.3. The technicalities of this are discussed in Section 6.5.2.

The following behavioural description for the Client could be written in FSP and included in the Client component:

```
Client = ( serv.login -> serv.pay ->
                         serv.download -> Client ).
```

This behavioural description shows an ordering of actions called through the `serv` port. The process is called `Client` matching the component name. The actions that it performs are logging in, then downloading an item, then completing payment. After completing this sequence the process repeats. This ordering of events cannot be derived from the interface descriptions alone.

When the framework generates the system description, it requests the FSP description from the Client and includes it inside the definition of the Client component (inside a special type of comment `/% ... %/`) in the Darwin specification, as below.

In the case that a component does not provide an FSP description of its behaviour, as with the Server component in this example, a process can be generated that allows any of the actions from the component's provided interfaces to be performed in any order. If no further detail is given, it is assumed that there is no interdependence between the methods exposed in the component's public interface, and that they may be called in any order. This may lead to over optimistic analysis, but without further behavioural specification for the component, no more accurate checks can be performed.

```
interface Shop { login; pay; download; }
component Client {
    require serv:Shop;
    /% Client = ( serv.login -> serv.pay
                -> serv.download -> Client ). %/
}
component Server {
    provide Shop;
    /% Server = ( { login, pay, download } -> Server ). %/
}
component System {
    inst s:Server;
        c:Client;
    bind c.serv -- s.Shop;
}
```

5.5.1. Composing the system

The Darwin compiler constructs a parallel composition of the behaviours of each of the separate components, employing an appropriate relabelling such that components that are bound together are synchronised. This relabelling is determined by the binding statements that are given in the Darwin description. For every pair of ports that are bound, `providesport.action` is relabelled to `requiresport.action`. In this way, any server action called by a client is relabelled so that it has the same name as the associated client action. The behaviour of the two components is therefore synchronised through the occurrence of shared actions. In order for one component to perform a shared action, all other components sharing it must perform it at the same time.

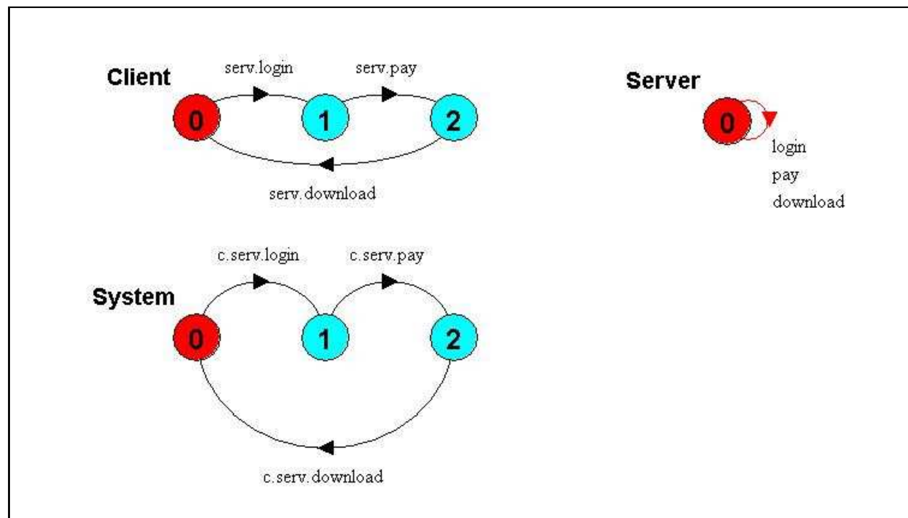

Figure 5.4.: LTS for Client-Server system

The above Darwin description for the simple client-server system is compiled to the following parallel composition in FSP.

```
set Shop = { login, pay, download }
Client = ( serv.login -> serv.pay
                      -> serv.download -> Client ).
Server = ( { login, pay, download } -> Server ).
||System =
    (s:Server || c:Client)
    /{c.serv/s
    }.
```

The set Shop is generated from the interface definition present in the Darwin description. For each of the components, Client and Server, the behavioural descriptions are extracted from the Darwin. For the composite component System a parallel composition expression is generated. System is defined as the parallel composition of the two instances that it composes, s:Server || c:Client. The next line applies a relabelling to this composition. The binding in the Darwin description states that the required port c.serv is bound to the provided port s.Shop. In the FSP, wherever s appears as an action prefix in the process description, it is to be replaced by c.serv.

Compiling this FSP description produces the labelled transition systems shown in Figure 5.4. The effect of the relabelling is observed, as the only actions that appear in the composite LTS are those in the alphabet of the client. As all of the client and server actions are synchronised, the behaviour of the system as a whole must proceed according to the behaviour of the client.

A problem with the synchronisation of the two components is observed if their behaviours are changed slightly. The same structure is now considered with the behaviours for the client and server changed to be following:

```
Server = ( login -> pay -> download -> Server ).
Client = ( serv.login -> serv.download -> Client ).
```

In this scenario, the server expects a protocol where a login and a payment action must occur before the download is allowed. The client however is looking to download without paying. In its process it does not perform the action `serv.pay`. It might be expected that when these two components are combined, the interaction would not be able to proceed as they do not agree on the protocol. However, due to the relabelling and the parallel composition, when the compilation process is applied, the resultant composite LTS for the system is still that of Figure 5.4. The reason for this is that as the `pay` action does not feature in the `Client` process, it is not shared, and so the only restrictions on its occurrence are those imposed by the `Server` process. This merely states that `pay` must happen after `login` and before `download`. Therefore, in the composite process, after `login` has been performed, `pay` (which is relabelled to `serv.pay` in the composite) is free to occur in the `Server` process, and so will occur, even though it is not performed by the client.

This is not the desired behaviour. The server should not spontaneously perform an action if it is not called by the client. To prevent this, `serv.pay` must be made an action on which the two processes synchronise (so that each can only perform it if the other does so at the same time) even though `serv.pay` is not performed by the client. This is done by using alphabet extension to introduce the action into the alphabet of `Client`.

The specification of `Client` is now:

```
Client = ( serv.login -> serv.download -> Client )
                                              + { serv.pay }.
```

This is generalised, so that for all components with required ports, their alphabets are extended with all of the actions available to be called through those required ports. This is done by extending the alphabet with the set of actions included in the interface definition, prefixed by the name of each required port. So `Client` becomes:

```
Client = ( serv.login -> serv.download -> Client )
                                              + { serv.Shop }.
```

where

```
set Shop = { login, pay, download }
```

When this definition of `Client` is composed with `Server`, a deadlock can be detected in the composite LTS, as the server is waiting for the client to perform an action that it never will.

5.5.2. Changes of configuration

As the configuration of a piece of software constructed from plugin components changes over time, the way that particular components behave may also change. Components may behave differently depending on whether they have other components connected to their required ports.

When a plugin is connected to the system, other components need to change their behaviour to take advantage of the new services provided. Existing components need to be notified that a new component has been connected. To achieve this, components register with the plugin framework as observers, to be notified when a change in configuration occurs that is relevant to them.

The framework calls the observer back through the `pluginAdded()` method. In the FSP we can use the corresponding action `pluginAdded` as a signal to change from one mode of operation to another. If a component implemented a basic matrix

analysis algorithm, but allowed a plugin to be connected that provided a more efficient implementation of this algorithm, the component might perform the calculation itself while its requires port is unbound. If and when it is notified that a plugin has been added (the port has been bound), the component will change its behaviour so that from then on the call is delegated to the plugin. This could be described in Darwin/FSP as follows:

```
component MatrixSolver {
   require fast:Algorithm;
   /%
   MatrixSolver = ( input -> calculate -> output -> MatrixSolver
                  | pluginAdded -> FastSolve ),
   FastSolve = ( input -> fast.solve -> output -> FastSolve ).
   %/
}
```

5.5.3. Verifying models

A risk in taking the approach described above is that the behavioural specification written by the developer does not match the actual behaviour of the implemented component. Possible approaches to overcome this include attempting to extract the behavioural model directly from the program code (see Section 8.3.1 for further details on this approach), or using a tracing technique to step through the execution of the program in parallel with stepping through corresponding actions in the model. These are both interesting approaches, but neither has been developed further in this thesis.

5.6. Predicting behaviour

To give a larger example of how the described techniques can be used to predict the behaviour of configurations of plugins, we consider an example based loosely on the Compressing Proxy Problem [33]. This is a problem originally from Garlan and Wing that provides a classic example of architectural mismatch, where components that appear to be compatible from their interfaces do not work together correctly in practice. This is precisely the sort of situation that the modelling techniques described here aim to detect and hence avoid.

A set of components are chained together to form a pipeline through which data can flow. Further components can be plugged in to the end of the pipeline increasing the length of the chain over time. The basic premise of the problem is that in order to increase the efficiency of data transfer along the pipeline, a compression module is introduced at either end, compressing the datastream at the source and decompressing it again at the sink.

In the original Compressing Proxy Problem, the pipeline comprises a set of filters which all run in a single UNIX process. Integrating a compression module that uses the gzip utility with this system requires some thought, as gzip uses a one pass algorithm, and runs in a separate process. An adapter is therefore used to coordinate the components.

5.6.1. The Source component

Each component is implemented as a set of Java classes and interfaces packaged into a Jar file. The plugin framework is started, with a Source as the component that forms the core of the system. The Source can accept a plugin forming the next element of the pipeline through the port `next` which has type `Filter`. The source generates data and sends it down stream if further components are connected, see Figure 5.5. The component includes an FSP description of its behaviour. The combined Darwin/FSP description of the Source component is as follows:

```
interface Filter { data; }
component Source {
   require next:Filter;
   /%
   Source = ( next.data -> Source ).
   %/
}
```

5.6.2. Filter components

Each of the other components contains a similar behavioural description. A BasicFilter can be added to the pipeline. The `BasicFilter` simply reads data from upstream and passes it on downstream. As seen previously, the component contains a class

`FilterImpl` which implements the `Filter` interface, and has a field of type `Filter` for a reference to the next component in the pipeline. `FilterImpl`'s `data()` method just calls the next component's `data()` method. The Darwin/FSP for this component is given below:

```
component BasicFilter {
   require next:Filter;
   provide Filter;
   /%
   BasicFilter = ( data -> next.data -> BasicFilter ).
   %/
}
```

5.6.3. The GZip component

The GZip component has two ports, one for input data and one for output. It takes in packets and writes them into a buffer. Once the end of the input data is signalled, the gzip algorithm is applied to the data in the buffer, and a (hopefully smaller) number of packets is sent out of the output port until all of the compressed data has been output.

In the behavioural description given below, the `GZip2` process describes the input and output behaviour, and it is composed with a bounded buffer with capacity two. It is the composite process `||GZip` that represents the behaviour of the component as a whole.

```
interface Proc { packet; end; }
component GZip {
   require out:Proc;
   provide Proc;
   /%
   GZip2  = ( packet -> put -> In ),
   In     = ( packet -> put -> In | end -> Zip ),
   Zip    = ( zip -> Out ),
   Out    = ( get -> out.packet -> Out | out.end -> GZip2 ).
   Buffer(N=2) = Count[0],
   Count[i:0..N] = ( when (i<N) put -> Count[i+1]
                  | when (i>0) get -> Count[i-1]
                  | when (i==0) out.end -> Count[i] ).
   ||GZip = ( GZip2 || Buffer ).
   %/
}
```

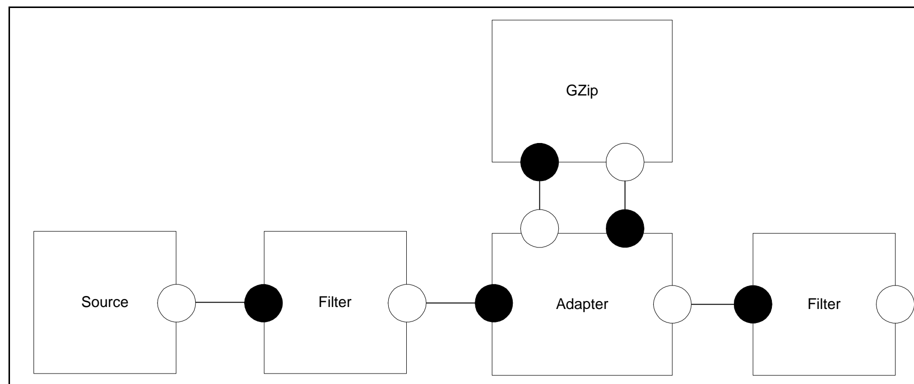

Figure 5.5.: Arrangement of components in pipeline with gzip

5.6.4. The Adapter component

The gzip compressor cannot be placed directly into the pipeline, and so needs an adapter component to pass data to it. The GZip component then plugs in to the adapter. When there is no gzip processor present, the adapter should behave like a plain filter. When in adapting mode, the adapter sends packets out to the processor and reads the processor's output back in before sending the processed packets on downstream. The `pluginAdded.proc` action corresponds to the calling of the `pluginAdded()` method with a parameter of type `Proc`. This triggers the transition from plain filter to adapting behaviour.

```
interface Proc { packet; end; }

component Adapter {
   require next:Filter;
         out:Proc;
   provide Filter;
         Proc;
   /%
   Adapter  = BasicFilter,
   BasicFilter = ( data -> next.data -> BasicFilter
              | pluginAdded.proc -> Adapt
              ),
   Adapt    = ( data -> out.packet -> ToProc ),
   ToProc   = ( out.packet -> ToProc | out.end -> FromProc ),
   FromProc = ( packet -> FromProc | end -> next.data -> Adapt ).
   %/
}
```

5.6.5. The complete system

The basic system may be constructed by assembling the chain of components from left to right (see Figure 5.5) without the GZip component. Before each new component is bound in to the system, a Darwin/FSP model is generated.

The Darwin compiler can be used to translate this specification to FSP, which can then be compiled into an LTS model. A model checker can then be used to determine whether particular properties hold for that configuration. In this case the property of interest is freedom from deadlock.

Without the GZip component, the system comprises four component instances, and three bindings.

```
component System {
   inst s:Source
       b:BasicFilter
       a:Adapter;
       b2:BasicFilter;
   bind b.Filter -- s.next;
       a.Filter -- b.next;
       b2.Filter -- a.next;
}
```

As the GZip component is not included, no bindings are present to the adapter's required port that has type `Proc`. This means that the adapter should not enter its adapting mode. Currently the model is free to perform the action `pluginAdded.proc` as it is not synchronised with anything. In order to prevent this, the system model is composed in parallel with a process modelling the framework that synchronises on `a.pluginAdded.proc` but never performs this action. Such a process can be defined as `STOP` with an alphabet extension to include the `a.pluginAdded.proc` action. Such a process can be generated to include an alphabet extension for any required port of any component instance that is not bound, *i.e.* the framework process describes all of the actions that it will not perform.

```
Framework = STOP + {a.pluginAdded.proc}.
||CompleteSystem = (System || Framework).
```

If the CompleteSystem process is built and checked, the model checker reports that it is deadlock free. If the GZip component is added, it will be bound to the

two `Proc` ports of the Adapter. Now the Adapter's required ports are all bound, and so `pluginAdded.proc` can occur. Rebuilding the model and running the model checker again, the following trace is detected as leading to deadlock (the output from the LTSA model checker is shown in Figure 5.6):

```
Trace to DEADLOCK:
    b.data, a.pluginAdded.proc, a.data, b.data, g.packet,
       g.put, g.packet, g.put, g.packet
```

This indicates that adding the gzip processor can lead to a deadlock if too many packets are put into GZip's buffer (causing it to become full) before the compression algorithm is applied and data is output. By detecting the possibility of this problem occurring in the model, before the system is assembled, a decision can be made not to bind this component with this adapter. The complete Darwin and FSP specifications are given in Appendix C.

5.6.6. A different Adapter

To solve the problem, and include the gzip processor successfully, a different adapter component can be used. The PacketAdapter described below passes data to the compressor one packet at a time and waits to read the compressors output before sending the next input packet. Because of this, the compressor's buffer never becomes full, and the system will not deadlock.

If the PacketAdapter is proposed as a replacement for the Adapter previously in the system, then a new model can be constructed. All bindings in which the old Adapter participated are adjusted so that they bind to the PacketAdapter instead. The GZip component is also bound to the PacketAdapter. Compiling the model for this configuration and invoking the model checker reveals an absence of deadlock. As this is the desired property, the old adapter can safely be replaced with the PacketAdapter, and the GZip component bound in.

```
component PacketAdapter {
   require next:Filter;
         out:Proc;
   provide Filter;
         Proc;
   /%
   PacketAdapter = BasicFilter,
   BasicFilter = ( data -> next.data -> BasicFilter
              | pluginAdded -> Adapt
              ),
   Adapt    = ( data -> out.packet -> out.end ->
               packet -> end -> next.data -> Adapt ).
   %/
}
```

5.7. Summary

This chapter has presented a technique for automatically generating a description of the structure and behaviour of an application that has been composed dynamically from plugin components. Using tools this description can be compiled into an LTS model, which can be tested, using a model checker, to determine whether various desirable system properties hold.

The structural description can be generated automatically by the plugin middleware, based on the interfaces exported by each of the plugins and the bindings made between them. Behavioural information for each plugin must be provided by the programmer in the form of a description in the FSP process calculus which is included in the component. By combining the behavioural information about each component with the description of the system structure, a model of the behaviour of the system as a whole can be generated.

The model can be compiled to the form of an LTS which can be analysed automatically, using a model checker, for adherence to desired system properties. Performing such analysis before a new plugin is added to the system allows us to predict whether the addition of this new component would cause the system to behave in an undesirable way.

The rationale that we advocate is one in which a reconfiguration to a running system is proposed, but before any change is actually effected, a model is generated that can

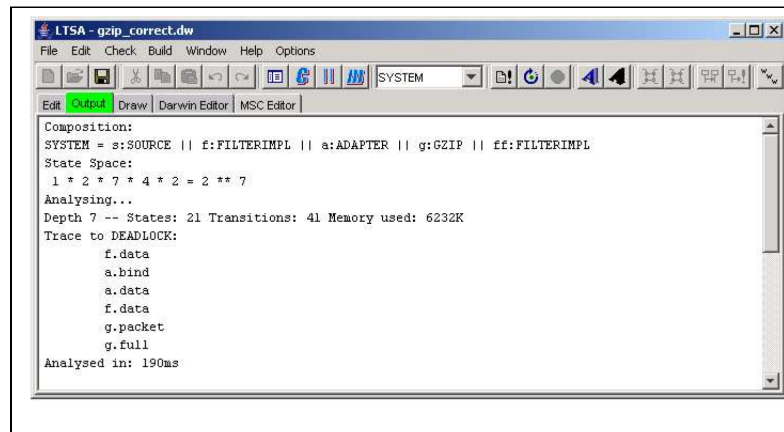

Figure 5.6.: Screenshot from LTSA showing trace to deadlock

be analysed. The deployer can then ensure that the configuration that will result from performing the upgrade preserves any properties that they require from the system. The model gives them the opportunity to experiment before making what may be a critical change. If they are happy with the results of the analysis, the middleware system can proceed in effecting the change, so that the system is updated in line with the model.

The degree of interactivity that is required varies in different situations. With a home user who does not have a great deal of understanding about the architecture of the system they are using, a fully automated cycle of modelling, analysis and reconfiguration is desirable. The user is unlikely to have enough knowledge of the software system to make an informed decision about a reconfiguration, and would prefer upgrades to proceed as automatically as possible, without being exposed to the details of what is going on behind the scenes. If the reconfiguration is being overseen by a deployment engineer with a higher level of skill and knowledge, we have found it useful for them to be allowed to manually approve or rehect the reconfiguration after seeing the results of the analysis phase. This allows them to strengthen or weaken properties in response to the output from the model checker in order to ensure that the system's behaviour will meet their expectations. In our implementation we have allowed either of these modes of operation to be selected.

6. Implementation

In this chapter, we describe the implementation of a platform for managing plugin-based applications, which we call MagicBeans. The requirements and details of the implementation are discussed and some technical issues of particular interest are highlighted[1].

6.1. Requirements

To enable the evolution of software systems through the dynamic addition and coordination of plugin components at runtime, we require some kind of runtime framework to be built. Examining the different cases we considered in terms of the models in the previous chapters, we have a number of functional requirements for the system.

The framework should form a platform on top of which an application can run. The platform should launch the application, and from then on manage the configuration of plugin components.

Plugins should be installed and configured as automatically as possible. The right interfaces and classes from each component should be detected, loaded and bound by the framework without the developer having to do any extra work. The matching of components should be taken care of by the framework.

It should be possible to plug components together in chains and other configurations as seen in Figures 3.1, 3.2 and 3.3. The configuration should be managed entirely by the platform.

Other plugin frameworks, *e.g.* that used by Eclipse [60], require a large amount

[1] A version of the material presented in this chapter was published in the proceedings of the Component Deployment 2004 conference [13].

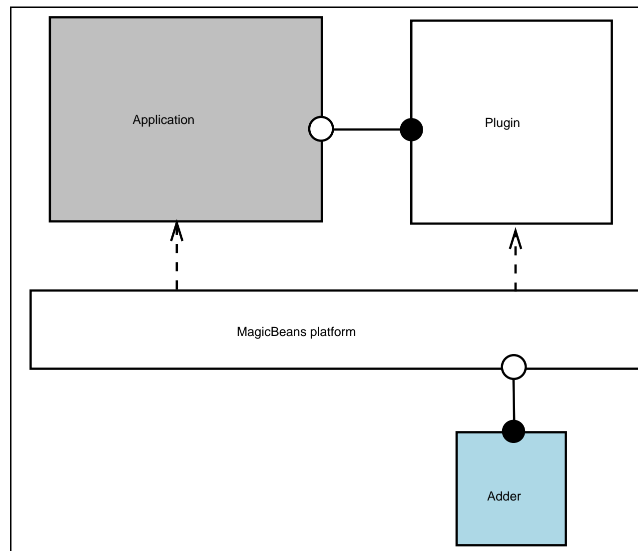

Figure 6.1.: Platform architecture managing a two component application

of metadata for each component in order to coordinate components and assemble systems. We propose that a large amount of this metadata is unnecessary, as the relevant information, especially regarding provided and required interfaces, may be recovered from the code of the component itself.

Using the plugin platform should have minimal impact on the developer or the user (or system administrator). The developer should not be forced to design their software in a particular way, to make extensive calls to an API, or to write complex descriptions of their components in any form of architecture description language. There should be no particular installation procedure that needs to be gone through in order to add a component, simply allowing the platform to become aware of the new component's location should be enough.

The mechanism by which new components are introduced to the system should not be prescribed by the platform. It should be possible to easily adapt the framework to allow components to be added in new ways, for instance: located by a user, or discovered in the filesystem or network etc.

As discussed in Chapter 5, there are various types of constraints that we may want to apply to the assembly of plugin systems, both structural and behavioural.

In order to successfully deal with resource management, it should be possible to

specify whether a certain component can accept one or many of a particular type of plugin, to be connected to a certain interface. The managing framework should ensure that such cardinality constraints are enforced. More general structural constraints might be specified in a language like Darwin^i [80]. When work on Darwin^i is completed, it will hopefully be possible to integrate it with the MagicBeans platform.

In order to ensure that the behaviour of assembled systems conforms to certain desired properties, it should be possible to construct and analyse a model of the system, in accordance with the techniques described in Chapter 5. Runtime support for this should allow the effects of potential upgrades to be assessed, to ensure that properties will be preserved, before a system of components is dynamically evolved.

6.2. Implementing Plugin Addition

MagicBeans is implemented in Java, and allows a system to be composed from a set of components, each of which is comprised of a set of Java classes and other resources (such as graphics files) stored in a Jar archive.

When a new plugin is added to the system, the platform searches through the classes and interfaces present in the new component's Jar file to determine what services are provided and required by the new component, and how it can be connected to the components currently in the system.

A class signifies that it provides a service by declaring that it implements an interface. In the example in Section 4.3.1, we showed an AirBrush component that might be added as an extension to a paint program. The `AirBrush` class implements the `GraphicsTool` interface. It can be added to any application that can accept a `GraphicsTool` as a plugin.

The paint program that can accept and use our `GraphicsTool` plugin has a slightly more complex design. For a component to use the services provided by a plugin, it must obtain a reference to an object from the plugin. This is achieved through a notification mechanism. The mechanism is based on the Observer pattern [31]. Any object can register with the platform to be notified when a new binding is made which is relevant to it.

This registers that an object is interested in new plugins, but does not specify the plugin type. An object signifies that it can accept a plugin of a certain type by declaring a method `pluginAdded(...)` that takes a parameter of that type, in this case `GraphicsTool`.

When the AirBrush plugin is added, observing objects with a `pluginAdded()` method that can take a `GraphicsTool` as their parameter are notified. This is done by the plugin manager calling the `pluginAdded()` method, passing a reference to the new `GraphicsTool` object, through which methods can be called. It is normal to assign this reference to a field of the object or add it to a collection within `pluginAdded()` so that the reference is maintained. In the paint program example, the new tool is added to a list of all the available tools. Classes can define multiple `pluginAdded()` methods with different parameter types, so that they can accept several different plugins of different types.

For each component, the plugin manager iterates through all of the classes contained inside the Jar file, checking each for implemented interfaces (provisions) and `pluginAdded()` methods, and finding all the pairs that are compatible. For a class to be compatible with an interface, it must be a non-abstract subtype of that interface. The matching process is performed using Java's reflection [40], custom loading [47] and dynamic linking features, which allow classes to be inspected at runtime. If a match is found, a binding between the two components is added to the system. The class in question is instantiated (if it has not been already), and the notification process is triggered.

There are various mechanisms through which plugins could be introduced to the system, and which is chosen depends on the developer and the application. Possibilities include that the user initiates the loading of plugins by selecting them from a menu, or locating them in the filesystem, that the application monitors a certain filesystem location for new plugins, or that there is some sort of network discovery mechanism that triggers events, in the manner of Sun's Jini [45]. MagicBeans does not prescribe the use of any of these. It uses a known filesystem location as a bootstrap, but components that discover new plugins can be added to the platform in the form of plugin components themselves (the platform manages its own configuration as well

as that of the target application) which implement the Adder interface. Figure 6.1 shows an example of the platform running, managing an application extended with one plugin, with an Adder component plugged in to the platform itself. Each Adder is run in its own thread, so different types can operate concurrently. Whenever an Adder becomes aware of a new plugin, it informs the platform and the platform carries out the binding process. We have written example applications that load plugins from a known filesystem location, and that allow the user to load plugins manually using a standard “open file” dialog box.

6.3. Plugin Removal

As well as adding new plugins to add to the functionality of a system over time, it may be desirable to remove components (to reclaim resources when certain functionality is no longer needed) or to upgrade components by replacing them with a newer version. Together these form the three types of evolution identified by Oreizy *et al* [62].

Removal is not as straightforward as addition. In order to allow for the removal of components, we need to address the issue of how to revoke the bindings that were made when a component was added. The platform could remove any bindings involving the component concerned from its representation of what is bound to what, but when the component was added and bound, the classes implementing the relevant interfaces were instantiated, and references to them passed to the components to which they were connected. These components will retain these references and may continue to call methods on the objects after the platform has removed the bindings. It is not possible to force all of the objects in the system to release their references. If the motivation for

```
class PaintProgram {
   ...
   void pluginRemoved( GraphicsTool gt ) {
      tools.remove( gt );
   }
}
```

Listing 6.1: The pluginRemoved() method

removing the plugin was to release resources then this objective will not be met.

We can have the platform inform any components concerned when a plugin is about to be removed. This is done using the same observer notification mechanism as we use when a component is added, but calling a different method `pluginRemoved()`. Any references to the plugin or resources the component provides should then be released. This can be seen in Listing 6.1.

When all of the notifications have been performed, the bindings can be removed. However, this technique relies on the cooperation of the plugins. We cannot force references to be released, only request that components release them. Components could be programmed simply to ignore notifications (or may not even register to be notified) and in this case will continue to retain their references after a binding is removed.

As a solution to this problem, in addition to using the notification mechanism, when classes providing services are initially instantiated, instead of providing another component with a reference directly to that object, the reference passed is to a proxy (see Figure 6.2). All the references to the objects from the plugin are then under the control of the platform, as the platform maintains a reference to each proxy. When a component is removed, the reference from each proxy to the object that provides its implementation can be nullified, or pointed at a dummy implementation. In this way we can force that resources are released. In the event that a component does try to access a plugin that has been removed, we can throw a suitable exception.

In order to provide this level of indirection, we use Java's `Proxy` class (from the standard API) to create a proxy object for each binding created. The `Proxy` class allows us to create an object dynamically that implements a given interface (or interfaces), but which, when a method is called, delegates to a given `InvocationHandler` which actually implements the method or passes the call on to another object. Using this mechanism, the implementation of the method can be switched or removed at runtime simply by reassigning object references. This gives us exactly what we need. When a plugin is removed, the reference to the delegate can be nullified. When a method is called we check for the presence of a delegate, and if it has been removed throw a suitable exception back to the caller.

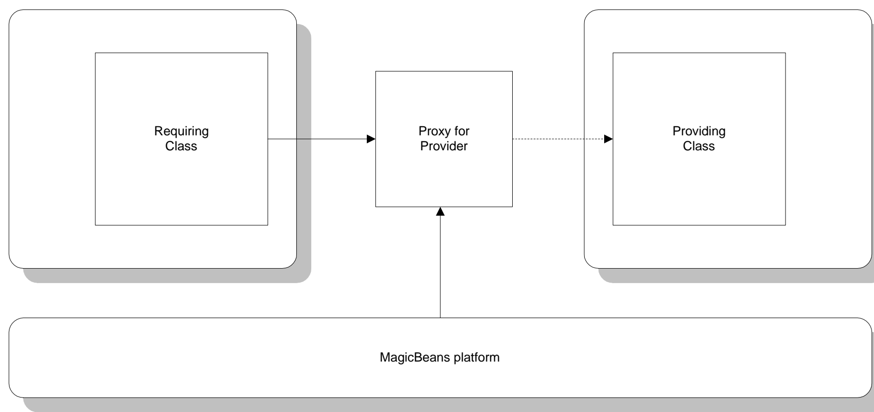

Figure 6.2.: Proxy objects are used to mediate between components.

Another major concern is deciding when it is safe to remove a component. For instance, it will be very difficult to replace a component if the system is currently executing a method from a class belonging to that component. In order to remove a component it must be in a quiescent state, *i.e.* not currently servicing any requests from other components. This problem is solved by synchronising the methods in the proxy object.

As Java's synchronisation model uses object level locks, if code from one synchronised method is executing, no other synchronised method in the same object can be executed concurrently. The methods in the `InvocationHandler` that change and remove the delegate, and also the `invoke` method, have been declared synchronized. As all calls to the delegate are passed through the `invoke` method, this mechanism ensures that a delegate cannot be removed or replaced during a call to a method that it provides. When a call is made to a provided service, the invocation is always carried out through a proxy object. As only one synchronised method of an object may be entered at any one time, a call to the proxy object to remove the implementing service will block until the service invocation is completed, and a quiescent state is reached.

6.4. Plugin Replacement

Plugin replacement can be performed in order to effect an upgrade of a system. However, before removing the old component, checks must be made to ensure that

the new version is compatible.

A safe criterion for compatibility of components might be that the new one must provide at least the services that the one it is replacing provides, and must not require more from the rest of the system [28]. In this way we can compare two components in isolation to decide whether one is a suitable substitute for the other.

However, in the case of plugin systems, there are a few more subtleties to be considered. With plugin systems, components that are used by other components are not strictly *required* but optional extensions that may be accepted. Therefore, in comparing components for compatibility we do not need to consider the case of what the components require, only what they provide. It is only this that is critical to the success of the upgrade.

Also, as we are performing upgrades at runtime, during system operation, we have more information than we would have if we just had the components in isolation. At any point the MagicBeans platform knows the current structure of the system, and so knows which of the interfaces that a plugin provides are actually being used (those for which a binding has been created during the addition process). It is therefore possible to say that new component is safe to replace another if it provides at least the same services as those that are provided by the old one, and are currently being used.

For example, we might have a component Brush which contains classes that implement `GraphicsTool` and `Help`. This plugin can be used with a graphics application as was shown previously, which will use the `GraphicsTool` interface. We could also use it with an application that allowed help to be browsed, or an application that combined both of these features. However, let us consider the case where we are using the Brush with our simple paint application. In this case, only the `GraphicsTool` interface will be used.

We may now write or purchase a new tool, say a SuperBrush. We want to upgrade the system to use this instead of the Brush. The SuperBrush does not provide the `Help` interface, but its implementation of `GraphicsTool` is far superior. If we use our first criterion for deciding compatibility, then we will not be able to upgrade from a Brush to a SuperBrush, as SuperBrush does not provide all the services that Brush does. However, if we use the second criterion, then in the context of the simple

paint application, SuperBrush provides all the services that are being used from Brush, (*i.e.* just `GraphicsTool`) and so we can perform the upgrade.

Replacement could be done by first removing the old component, and subsequently adding the new one, using the mechanisms as described above. However, due to the presence of the proxy objects which allow us to manage the references between plugins, we can swap the object that implements a service, without having to notify the client that is using it. In this way it is possible to effect a seamless upgrade.

6.5. Constraint checking

As discussed in Chapter 5, in addition to conforming to the typing information provided by interfaces, there may be other constraints that should be considered when constructing configurations of components.

In order to check particular configurations against sets of constraints, models need to be generated which have both a structural part and a behavioural part. Here we describe how the two parts of the model are generated from the running system, combined and then analysed.

6.5.1. Building the Structural Model

Each plugin that the MagicBeans platform uses is loaded as a Jar file containing Java class files, which contains Java byte-code, and any other resources needed by the plugin, but not (necessarily) the source code. To identify the services provided and accepted by a new plugin, a custom classloader is used to load each class in turn from the component. For each class, a list of the interfaces implemented by that class is determined using reflection. To find the information about accepted or required interfaces requires a more in depth analysis than is possible through the functionality provided by Java reflection alone. We need to determine the names of fields assigned to within particular methods. To do this, the bytecode of each class is analysed directly using the Byte Code Engineering library (BCEL) [22]. This library permits the byte-code for specific methods to be examined. It is possible to extract

artifacts like identifiers for variables and fields. These are used as port names, as described in Section 5.4.

The MagicBeans platform has all the information about the current configuration of the running application at runtime, as all bindings are created and revoked by the platform. It holds a data structure that records the current set of bindings between ports and components which is used to generate the Darwin description of the complete system, including the instances of components and the bindings between them. The resultant model is stored in an internal data structure that can easily be converted to a textual representation for output.

6.5.2. Building the Behavioural Model

The behaviour of a specific component needs to be specified, in terms of an FSP model, by that component's developer. In order to be able to deploy the component with the specification, the specification needs to be included in the component in some way. However this needs to be done in such a way that the specification is accessible to the plugin framework.

In the MagicBeans implementation, the way that a component provides its specification to the framework is by providing access to it through a plugin interface. Each component provides a port with the `FSPSource` interface.

```
public interface FSPSource {
   public String getFSP();
}
```

The framework can accept plugins of this type. During the generation of the model, for each plugin that is bound to the platform through this interface, the platform calls the `getFSP()` method, which should return a string representation of the FSP description of the component's behaviour. This is then included (wrapped in special comment symbols) with the Darwin specification for each component as it is output textually.

The MagicBeans platform manages its own configuration, as well as that of the application that is running on top of it. It uses exactly the same mechanisms for binding

to the `FSPSource` port provided by plugins as is used to create bindings between components in the target application.

In order to make it easier for the developer to create the class that implements the `FSPSource` interface to include in their component, we have created a tool that allows this class to be generated automatically and included in the component, so that the developer only has to provide the textual process description. This is helpful as the inclusion of long strings in Java source code is error prone, due to having to escape particular characters *etc.* , and also removes the need for developer to know about the `FSPSource` interface, which is part of the MagicBeans framework rather than the target application. The tool also analyses the interfaces provided and required by a particular component to provide an alphabet of actions for the developer to use in writing their process description, based on the information used to generate the structural model. This helps to ensure the correct correspondence between the behavioural and structural models.

6.5.3. Specifying Properties

Properties are also specified textually in FSP. The most useful types of properties in practice have been found to be those that are specified as assertions in linear temporal logic (LTL).

Properties can be included within a particular component by providing a class that implements the `FSPSource` interface. This allows a specification to be given for things that a component requires of its environment. In all cases that this component is used in conjunction with other components that particular property must hold for the interaction between them.

System wide properties are specified separately from any particular component, written by the deployment or configuration manager rather than the developer of a particular component. These properties are used to guide assembly of the overall system at a particular site. They can be added to the system as separate components, just containing a class that implements the `FSPSource` interface, that will be bound to the component framework. Alternatively, if the deployer is using the analysis tools

in a more interactive mode, they can enter properties directly into the model checker's edit window.

6.5.4. Checking the Model

The generated model can be compiled into FSP and then to a labelled transition system by the Labelled Transition System Analyser tool (LTSA) [51] as discussed previously. In order to achieve this, we extended the existing Darwin compiler and integrated it (as a plugin, see Chapter 7) into LTSA.

To allow models to be processed and analysed automatically, we built a further extension, allowing LTSA to receive Darwin/FSP models (in a textual format) over the network. The LTSA then forms a server which the MagicBeans platform can contact when it needs to perform any analysis. On receipt of a model, the LTSA converts from Darwin/FSP to pure FSP, and then to LTS. It then performs a model-check to analyse whether the model conforms to given properties.

Depending on the result of the analysis, a notification is returned to the MagicBeans platform over the network. If the notification indicates that the proposed configuration preserves the desired properties then the reconfiguration may proceed. If the analysis has revealed that a particular property will be violated, a different configuration of the same components, if one is possible, may be tried. Alternatively the reconfiguration may be aborted and operation allowed to continue in the previous configuration.

Properties can be included along with the code and specification of a particular component, or independently. Properties that are included within a component typically place a constraint on the environment within which it will operate. They may specify a behavioural requirement of the services provided to them by other components, in addition to the interface requirements that are expressed as types in the component's code. These will normally be written by the developer of the component in question.

Global system properties will be specified independently of any particular component, as they refer to the operation of the system as a whole. They will most likely be specified by the deployment engineer at a particular site. They are interested in ensuring the correct operation of the integrated system, but may have particular, site

specific, requirements. Such global properties can be packaged into components that contain no other code and deployed into the plugin system. They will be bound directly to the plugin platform, and can be included in the analysis phase.

6.6. Technical Innovations

During the design and implementation of the MagicBeans platform, some interesting technical issues arose. These are discussed in the following sections.

6.6.1. BackDatedObserver

There are some cases in which the notification system described in previous sections has limitations. If, on adding a new plugin, multiple bindings are formed, it may be the case that bindings are created before the objects that will observe the creation of these bindings have been initialised and registered as observers.

For example, consider the case where we have two components, each providing one service to and accepting one service from the other. If component A is already part of the system, and component B is added, a binding may be formed connecting B's requirement with A's provision. Currently no observers from B have registered, and so none are informed of the new binding.

A second binding is then formed between B's provision and A's requirement. At this point, a class from B is instantiated. A reference to this object is passed to any observers in A. During the creation of the object from B, the constructor is run, and the object registers as an observer with the PluginManager. As the registration is too late, although the PluginManager matched two pairs of interfaces to create bindings, the situation that results is that A holds a reference to B, but not the other way around.

To solve this problem, we introduce the notion of a BackDatedObserver. This is an observer that, on registering, has the opportunity to catch up on previous events that occurred before it registered, but which are relevant to it. In the last example, having the observers register as BackDatedObservers would mean that the observer from B

would be passed a reference to the object from A as soon as it registers, and it would be possible to call methods in both directions.

Implementing this variation on the traditional observer pattern requires that the participant that performs the notification keeps a history of past events, so that it can forward them to new observers when they register.

6.6.2. Distinguishing components

In order to be able to tell which observers need to be notified about which new bindings, it is necessary to maintain a record of which objects come from (or belong to) which components. That is to say, which component contains the class from which the object was created. This could be done by requiring the developer to create objects by calling a special factory method that would create objects and update the relevant data structures. However, such a scheme would impinge greatly on the natural style of programming. It would be necessary for the programmer to write something like:

```
A myA = (A)ObjectFactory.create( ``A'' );
```

instead of the usual

```
A myA = new A();
```

There are a number of problems with this. Firstly, it is a lot more cumbersome to write. Secondly, it removes static type safety. Thirdly, we cannot force programmers to use this mechanism, and no information will be recorded about any objects created in the normal style.

In a language that allows operator overloading (for example C++), we could implement a `new` operator that performs the appropriate record keeping, allowing object creation using the normal syntax. However, operator overloading is not available in Java.

The solution to this problem that has been adopted utilises the fact that in Java every object created holds a reference to its class, and every class in turn to its class loader. By associating a separate class loader with each plugin component, we can group

objects into components on the basis of their class loaders. In fact, we made the class `Component`, which manages all of the information relevant to a particular plugin, a subclass of `java.lang.ClassLoader`, so that for any object, calling

```
getClass().getClassLoader()
```

will return a reference to its `Component`.

6.6.3. Multi-methods

As all of the objects that come from plugin components may be of types that are unknown to the MagicBeans platform, objects are created using reflection, and the references that are used to point to them have the static type `Object`, which is the ultimate superclass of all classes in Java.

If the PluginManager were to attempt to call one of the `pluginAdded()` methods in a component, it would pass a parameter with static type `Object` and the Java runtime would require that the method being called took a parameter of type `Object`, even if the dynamic type of the parameter was something more specific.

In fact, during the compilation of the plugin component, the Java compiler will complain that there is no method `pluginAdded( Object o )`. If the developer adds this method, then this is the one that will be called at runtime, regardless of the dynamic type of the parameter passed. The reason for this is that methods in Java are only dispatched dynamically on the type of the receiver, not that of the parameters [39].

This causes a problem as we wish to use `pluginAdded()` methods with different parameter types to specify the types of plugin that a component can accept.

In the implementation of MagicBeans we have overcome this problem by using reflection to dispatch certain methods dynamically on the parameter types as well as the receiver. This is often called “double-dispatch” or “multi-methods” [18].

We created a class `MultiMethod` which has a static method `dispatch()` which takes as parameters the intended receiver, the name of the method and the parameter.

```
MultiMethod.dispatch( receiver , ''pluginAdded'' , parameter );
```

Reflection is used to search through the methods of the given receiver to find one with the given name and a formal parameter type that matches that of the parameter passed as closely as possible. This method is then invoked, again using the reflection mechanism. Calling methods in this way does remove a degree of type safety from the language, as the calls cannot be checked by the compiler. However, the semantics that we have defined is that if there is no method in the target object that matches, no method is called. Therefore it is never possible to have the wrong method called, the worst that can happen is that nothing happens.

Double dispatch is only used when calling the methods `pluginAdded()` and `pluginRemoved()`, not for any subsequent calls between components. This means that the performance penalty incurred by calling methods in this way is kept to a minimum.

6.6.4. Proxying

To give the plugin platform the ability to manage references between components, when a binding is created, what is passed to the accepting component is not a reference directly to an object from the providing component, but rather to a proxy for that object. This allows the plugin platform to maintain control over to which object the proxy delegates calls, and therefore affords reconfiguration.

One complication is shown in the following scenario: a component A calls a method on a proxy object which delegates the call to an object in component B. The method executes and returns an object (which may contain a reference directly to an object from B). The proxy passes this back to A. There can now be a reference (or chain of references) directly from A to B without going through a proxy. This means that there can be a connection from A to B that is not controlled by the plugin platform. It is now impossible for the platform to disconnect all references to component B when it wants to remove it.

A similar situation can occur if an object from component A calls a method on the proxy, delegated to B, and passes a reference to an object as a parameter. This will again allow a direct reference from one component to another to be held, which the

platform is unable to manage.

Our solution to this problem is to intercept any objects returned from methods called on a proxy, or any objects passed as parameters to methods executed via a proxy. A further proxy is then generated for each parameter to be passed, or the object to be returned, provided that the object in question is not already a proxy object. The new proxy is then used in the method call or return.

However, in general, use of this mechanism would entail generating proxies on the fly for arbitrary objects, which is not supported by the standard Java Proxy class. In order to use the Java Proxy API, the type of the object to be proxied must be an interface type and not a class type. This means that it is not possible to create a proxy for an object of a class that is not declared as implementing a particular interface. It might be possible to overcome this problem by using byte-code manipulation techniques to generate interfaces for all classes, and insert code into the classes to declare them as implementing the interface at load time, but this solution seems messy.

A more pragmatic approach, and the one that we have taken, is to restrict the types of all parameters, and the return values of any methods, that are to be passed across component boundaries, to be interface types instead of class types. This does not seem an overly problematic restriction, as the external interface to a component should be declared in terms of interfaces rather than classes in line with the general principles of encapsulation in object-oriented and component-oriented programming. This allows the standard Java Proxy API to be used to create proxies for any objects that cross component boundaries.

6.7. Summary

We have described the implementation in Java of MagicBeans, a platform to manage the runtime behaviour of plugin components, which follows the component model set out in Chapter 3. Various technical aspects and implementation details of loading plugin components and determining their required and provided services were discussed, along with details of how configurations of components are managed at runtime.

The details of how MagicBeans generates models of prospective configurations of

components and interacts with the LTSA model-checker in order to analyse properties of these models were also discussed.

7. Case study: Extensible LTSA

The Labelled Transition System Analyser (LTSA) [51] is a Java application that allows complex systems to be modelled as labelled transition systems. These models can be mechanically checked for various properties, making sure that either nothing bad happens (safety) or that eventually something good happens (liveness). The core functionality of LTSA is to take textual input in the form of the FSP process calculus [51], to compile this into state models, which can be displayed graphically and animated, and to check properties of these models.

The tool was distributed to a large number of users for use in the modelling and analysis of concurrent and distributed systems, both in academia and industry. With time, scope for a variety of extensions to the tool was realised. However, not all of the extensions would be useful to all users. A monolithic tool incorporating all of the features would quickly become large, complicated and slow. Managing the continual redeployment of the system to users as upgrades became available would likely be problematic.

To avoid these problems, the core of LTSA was altered to use our MagicBeans software platform to allow the various extensions to be implemented as plugins. The aim of using the plugin architecture was that rather than having one monolithic tool which combined all of the above functionality, the different extensions could be encapsulated in separate modules, and only the modules that the user required would be loaded. This selection of features should be able to be done in a dynamic way, so that no changes to the source code need to be made in order to add or remove features.

By providing a standard interface for LTSA plugins, the core of the application can use any extensions (plugins) that the user requires. Currently plugin components are placed in a specific directory and discovered when the application starts up, but the

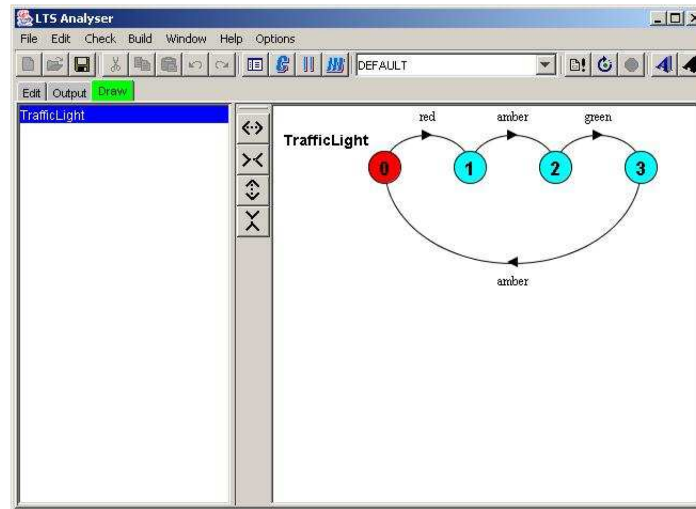

Figure 7.1.: The default LTSA with no plugins added.

mechanism could easily be altered to allow plugins to be added at any time during execution, all that would be required would be the inclusion of a “discovery” plugin for the platform that checked for the presence of new components periodically. The LTSA core accepts any plugins with a particular interface. The application interrogates each plugin in turn (by calling methods declared in the interface) to find out whether it provides certain types of GUI features (menus, tool bar buttons *etc.*) that should be added to main application’s user interface. The plugins then respond to the user clicking on the buttons or menus by executing code from handler classes inside the relevant extension component.

In order to allow the plugin to call methods in the core of LTSA, for example as would be needed in order to invoke the FSP compiler or the model-checker, another

```
interface LTSAPlugin {
    boolean addToolBarButtons();
    List getToolBarButtons();
    boolean addMenuItems();
    Map getMenuItems();
    ...
    void initialise();
}
```

Listing 7.1: Fragment of the LTSAPlugin interface

binding needs to be made in the opposite direction. In order to achieve this, the core of LTSA was made to expose a particular interface, giving access to particular core functions. Extension plugins can then accept a plugin of this type (and the appropriate matching is performed by the MagicBeans platform). To ensure that the two-way binding is constructed successfully, the BackDatedObserver pattern (see Section 6.6.1) is used.

```
interface LTSA {
    JEditorPane getEditorPane();
    void parse();
    void compile();
    void composeCurrentState();
    ...
}
```

Listing 7.2: Fragment of the LTSA interface

Various extensions have been built, notably to allow more illustrative animations of the behaviour of models, to allow FSP to be synthesised from graphical Message Sequence Charts representing scenarios [74] so that properties of these scenarios can be analysed, and to provide a facility for interacting with behaviour models by means of clicking items on web pages served over the internet to a web browser [15]. These are described in more detail in the following sections. More extensions for LTSA are currently in development and the use of the plugin framework has made it very easy to integrate new functionality with the tool.

7.1. MSC Editor

Message Sequence Charts (MSCs) are a graphical technique for describing the behaviour of systems. They provide a specification in the form of examples of acceptable behaviour, consisting of sequences of messages exchanged between the components forming a system. MSCs also outline the high-level architecture of the system.

Several algorithms exist for synthesising labelled transition systems from MSCs (notably the work of Whittle and Schumann [82]). Uchitel extends this work to analyse

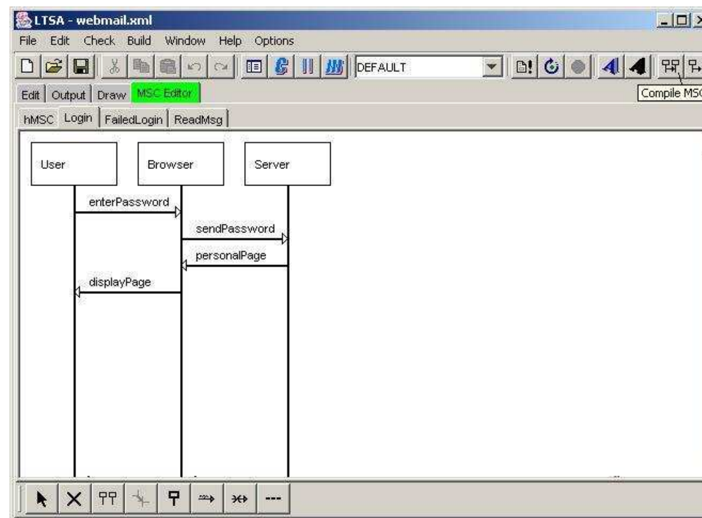

Figure 7.2.: The LTSA running with the MSC plugin added.

MSC specification for the presence of *implied scenarios* [77]. To automate this analysis we developed an extension to the LTSA tool.

It was the development of this extension that first motivated the provision of a plugin mechanism for providing extensions to LTSA. LTSA is already used by a large user base, most of whom just write and analyse textual specifications. Only a small subset of the users would (at least initially) be interested in the extra functionality provided for working with MSC specifications.

The MSC plugin provides a graphical editor for MSCs so that MSC specifications can be input and edited. It encapsulates the algorithm for converting the MSC specifications to FSP models, as well as an analysis algorithm for detecting implied scenarios in the specification [74].

The plugin generates textual FSP from the MSC model, and passes this to the core of the LTSA tool. This is done by calling the `getInputPane()` method in the `LTSA` interface (see Figure 7.2), and appending the text to this pane. The plugin can then invoke other methods through the same interface to parse, compile and safety check the FSP specification. Counter example traces are read in by the plugin and displayed graphically as MSCs.

Possible future work on this project involves providing pluggable algorithms for implied scenario detection. This was proposed by Henry Muccini [56] (University of

L'Aquila, Italy). The plugin model could be used so that, if the algorithms could be encapsulated in separate components, behind a common and well defined interface, they could be interchanged freely without having to alter any of the source code.

7.2. Darwin compiler

Darwin [50] is a language for describing the structural elements of software architectures. The addition of structural information to behavioural specifications greatly increases our ability to construct and analyse models of systems. Darwin can be used to describe a system's structure, and from this a composition expression can be generated to compose processes describing the behaviour of individual components. This technique was described in Chapter 5 in relation to generating and analysing models of plugin systems.

Incorporating a Darwin to FSP compiler/translator as an extension to LTSA, allows both aspects of a system to be easily modelled. In a similar way to the MSC plugin, an extension was developed that adds an editor pane to the LTSA (a text based one this time). The compiler for the Darwin specification is encapsulated inside the plugin, and generates FSP as its output. This is written in to the editor pane as described before.

The combination of Darwin and FSP provides useful expressive power, but it was realised that there was scope to do more if both the Darwin plugin and the MSC plugin were used together. Rather than relying on the user writing FSP descriptions of components' behaviour by hand, behavioural descriptions could be taken from an MSC specification if available. Certain generalisations are made to generate component models from several instances, and to map action names on to ports. The techniques for doing this are described in detail in a paper published in the proceedings of FSE 2004 [76].

The ability of the plugin mechanism to connect arbitrary pairs of components, rather than all extensions being made to a central core, meant that communication could be performed directly between the Darwin plugin and the MSC plugin. By exposing the same interface in both of these components, the plugin platform will create a new binding allowing them to communicate if both of them are present. In the case that

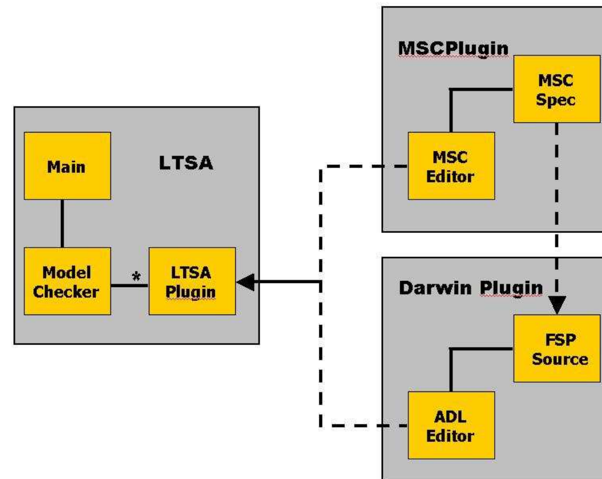

Figure 7.3.: A UML style diagram showing the structure of plugin LTSA.

one or both of the components are not present, this functionality is not available, but this change of configuration requires no changes to the source code, and can be updated dynamically by adding or removing components.

The diagram in Figure 7.3 shows how the various classes and interfaces within each of the components realise the provided and required interfaces. The dashed arrows denote interface implementation. The Darwin plugin can request behavioural information about a particular component from the MSC plugin through the `FSPSource` interface. The core of LTSA can accept any number of `LTSAPlugins`, denoted by the star on the association, but the Darwin plugin only accepts one `FSPSource` at any one time. The system as a whole comprises about 75,000 lines of Java code, with most of the code being in the core component and each of the plugins consisting of a few thousand lines of code.

7.3. Web Animator

Tools based on formal methods, such as the LTSA, are often seen as being difficult to use. They are not particularly suited to validating designs with clients or end users in the software development process, as they typically are not able to relate to the

concepts used in the model.

In order to more successfully use formal models at the design stage, and validate designs with the user, animation techniques can be used. Animation allows the behaviour of the model to be rendered in a fashion that is recognisable by the user.

The Web Animator extension to LTSA allows behaviour models to be animated through a sequence of interactive web pages [15, 16, 17, 75]. We have carried out several experiments modelling the user's progression through the pages of web applications using labelled transition systems. By using a web-based animation technology, the user can be shown the current state of the system in terms of a web page with particular links or buttons that they can click on the trigger transitions in the LTS model and move to a new state. As the animation is generated dynamically from the model based on a declarative set of visualisation rules, changes in the model are immediately reflected in the visualisation. If, during the interactive exploration of the model, the user finds something that should be changed, such a change can often be incorporated very quickly. This is in contrast to standard prototyping techniques, where making changes in response to user comments and requests could take several days or even weeks, depending on how much code needed to be written to implement the change.

7.4. NASA Assume-Guarantee Reasoning plugin

Work has been done by Cobleigh, Giannakopoulou and Pasareanu at the NASA Ames research centre, to develop techniques to learn assumptions for use in assume-guarantee reasoning [19].

They implemented their techniques as a plugin to LTSA. By programming against a well defined interface, they were able to develop extensions for LTSA without having to have access to the source code. In addition, they were able to integrate their extensions without having to release their source code to anyone else (which would have caused licensing problems). NASA can distribute their extension freely, and anyone can install it into their version of LTSA, without having to upgrade to a new version of the core. Meanwhile, development can continue on the core of the LTSA tool, and this can be compiled separately as the decoupling of components means that

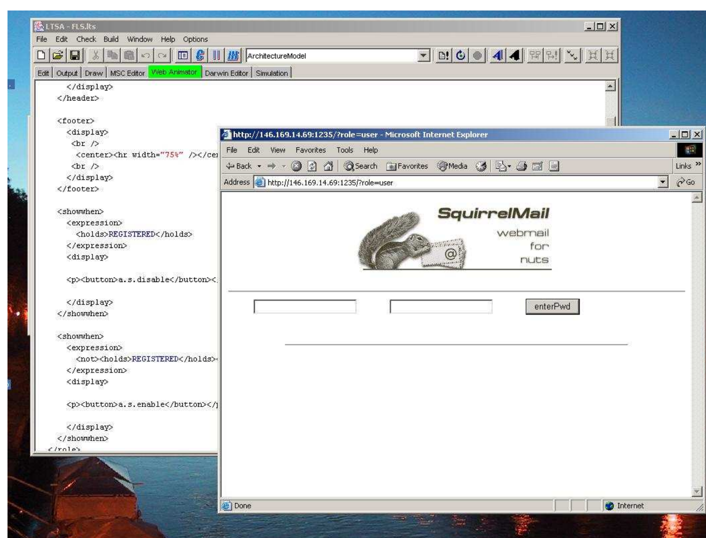

Figure 7.4.: The LTSA running with the Web Animation plugin.

no mention of specific classes from NASA is made in the source code of the core. This is beneficial as the intention is to make the core of the tool open source, but not all of the extensions will be open source.

7.5. Other plugins

Some students have used the plugin mechanism to develop extensions for LTSA as part of research and project work. The modularity afforded by using plugins meant that their development and experimentation could be carried out without affecting development of the core of LTSA. It also meant that they could easily create their own custom version of LTSA by installing their module, and pass it to anyone else who wanted to try it (for example their supervisor), without having to send a customised version of the whole tool.

BPEL to FSP translator The BPEL language can be used for specifying business processes and workflows. A technique was developed by Howard Foster, a PhD student at Imperial College London, to translate BPEL specifications into FSP models [30]. We have implemented this as a further plugin to LTSA.

Petri nets to stochastic FSP Wallace Wong, an undergraduate student at Imperial College London, built a plugin for LTSA to translate petri-net models into stochastic FSP models [83].

7.6. Constraints for LTSA plugins

As different plugins for LTSA are developed by different parties, to ensure that different configurations work together correctly, some behavioural constraints can be imposed. Such constraints are used to ensure that only plugins that allow interaction with the core component through the expected protocol will be added to the system. The protocol that is required is that all plugins follow a particular sequence of actions for initialisation, normal operation and cleanup.

We consider a scenario where the system as a whole comprises the core LTSA component, the MSC plugin and the Darwin plugin. Both of the plugins provide an implementation of the `LTSAPlugin` interface, and are bound to requirements of the LTSA core. The Darwin description that would be generated as a snapshot of this configuration is as follows:

```
interface LTSAPlugin { init; run; cleanup; }

component System {
  inst core:LTSACore;
       msc:MSCPlugin;
       dwn:DarwinPlugin;
  bind core.plugins[1] -- msc.LTSAPlugin;
       core.plugins[2] -- dwn.LTSAPlugin;
}
```

The generated Darwin description for the core LTSA component features an array port `plugins`, which is sized to be big enough to contain all of the bindings connected to this port. The special constant `PLUGINS_MAX` is generated, which corresponds to the size of the `plugins` array. We use this in the specification of the behaviour of the LTSACore component. Here we show the Darwin and FSP combined so that the correspondence can clearly be seen.

```
component LTSACore {
  require plugins[2]:LTSAPlugin;
  /%
  LTSA_PLUGIN = (init -> run -> sync -> cleanup -> END).
  ||LTSACore = forall [i:1..PLUGINS_MAX]
                  plugins[i]:LTSA_PLUGIN/{sync/plugins[i].sync}.
  %/
}
```

The parallel composition uses the `forall` operator to compose the behaviours of all of the currently connected plugins. The relabelling at the end causes synchronisation of all of separate processes by relabelling any particular plugin's `sync` action to one common `sync` action.

The core of LTSA follows the following protocol, operating in two phases: all currently available plugins are initialised and then run, later they are shut down. These phases do not overlap; all plugins are initialised and run before any are shut down.

The property that we desire to hold for a particular possible configuration of LTSA is that it is possible for each component to perform a `cleanup` action, so that the system as a whole can be cleanly restarted.

```
assert ALLCLEANUP =
      forall [j:1..PLUGINS_MAX] <>(core.plugins[j].cleanup)
```

This property, specified in linear temporal logic (LTL) [65], is parameterised by the `PLUGINS_MAX` constant, so that it can specify that for all installed plugins, it is possible (specified using the diamond operator) for them to perform the `cleanup` action.

The behavioural specification that we have written for the MSC plugin is, in FSP:

```
MSCPlugin = ( init -> run -> cleanup -> MSCPlugin ).
```

This indicates that the MSC plugin follows the protocol that we expect of each plugin, and can cycle through these operations repeatedly. The Darwin plugin has a very similar process description:

```
DarwinPlugin = ( init -> run -> cleanup -> DarwinPlugin ).
```

The complete Darwin/FSP description for this configuration can be compiled into a labelled transition system as described in Chapter 5. The LTL property can then be checked. In this case, no violations of the property are detected by the model checker, and so this is a valid configuration.

We now consider the same configuration, but with the Darwin plugin replaced with a proposed new version, which we will call DarwinPluginV2. This has the same interfaces as the original version considered above, but a slightly different behavioural specification. Notably this version does not expect to be asked to clean up before being reinitialised.

```
DarwinPluginV2 = ( init -> run -> DarwinPluginV2 ).
```

Including this component in the model and checking the property reveals a violation, as it is not possible for all plugins to complete the `cleanup` operation that was specified. The output trace from the tool is:

```
LTL Property Check...
-- States: 7 Transitions: 16 Memory used: 7615K
Finding trace to cycle...
Depth 6 -- States: 19 Transitions: 45 Memory used: 8055K
Finding trace in cycle...
Violation of LTL property: @ALLCLEANUP
Trace to terminal set of states:
        core.plugins.1.init
        core.plugins.1.run
        core.plugins.2.init
        core.plugins.2.run
        core.sync
        core.plugins.1.cleanup
Cycle in terminal set:
LTL Property Check in: 521ms
```

This property violation shows that the new version of the Darwin plugin is not suitable to replace the old version in this configuration, as although it does expose the same

interfaces, it does not expect the same calling protocol. This reconfiguration should therefore not be carried out.

7.7. Discussion

Refactoring LTSA to use a plugin architecture based on MagicBeans has been a successful exercise from a number of standpoints, but also uncovered a few difficulties of working this way. Previously, any changes to the LTSA to support new research projects, for example the Scenebeans animator [67] or the translator from LTL formulae to Buchi automata [38], had involved changing the core of the LTSA code, creating an experimental version. It was difficult to maintain consistency between versions and to find one working version that contained all of the different extensions.

There was some initial overhead in refactoring the core of the application to take advantage of the plugin system, but this time was paid back in the ease with which further extensions could be created and integrated. The core of the system, which is well tested, can still be used by itself as a teaching tool. The normal configuration that is used in undergraduate laboratory exercises does not include any of the research project extensions, and can be maintained separately.

Different researchers can now use the latest version of the core of the system along with their own collection of extensions. If an upgrade is made to the core, just this component can be distributed and used without affecting any of the work done on extensions, as long as the exposed interfaces are kept the same.

We have found the ease of reconfiguration particularly useful when doing demonstrations, as in order to focus the audience's attention on the particular part of the functionality of the tool that is being presented, a configuration can be constructed that does not include any plugins that will not be used in the demonstration, removing any unnecessary clutter from the interface.

Our collaborators at the NASA Ames laboratory were able to easily write and deploy an extension to the tool given a set of interfaces to implement or use, a small example plugin and some instructions showing how to use the plugin framework. This way of working was beneficial to the NASA researchers as when they experienced a fault in

their program, they were able to look only within the relatively small scope of the code of their plugin, as opposed to searching through the entire code of the core of the tool.

There have also been a number of extensions written by students at Imperial, notably the work of Wallace Wong [83], where the plugin was developed entirely independently of the core LTSA development. This shows that well defined interfaces and an intuitive programming model make it easy for components to be developed by independent teams and then integrated.

One difficulty experienced by the NASA team was that the exposed interface was somewhat limiting. They did not necessarily have access to the the features of the core that they required. In some situations, where there was no alternative, this caused them to request a change in the interface to reveal extra functionality. This did not cause too many problems as there were not very many users of the plugin version of LTSA at that time, and the changes that were requested were able to be handled by adding methods to the interface. Adding methods is a binary compatible change, and so should not cause any problems with backward compatibility of the component. However, this exercise did point out the issue that in providing an extension point allowing another party to provide a component to work with a system, the developer must anticipate the functions that the third party will want to call. With a large component this can be difficult as a large amount of functionality may be available, but we want to present clean and concise interfaces.

In other cases, rather than requesting a change in the interface, the NASA researchers found a way to work around the problem by accessing the core in a different way. For example, rather than passing objects representing LTSs backwards and forwards between components, they passed textual representations, as this is the interface presented by the core. This led to some performance problems as every time the textual description was edited, it had to be reparsed and recompiled. This was an expensive operation for large models. At the time of writing the textual representation is still being used, but it is likely that the processing time will soon become prohibitive and the interface will have to be changed. It should be noted that the performance problems described were not caused by the overhead of the loading and binding mechanisms used in the plugin framework, but owed to restrictions imposed by programming to

interfaces. There is a trade-off between the flexibility of deployment and ease of development offered by the use of plugins, and the difficulties of not being able to customise every part of the application in order to optimise performance.

7.8. Summary

This chapter described the use of the MagicBeans plugin technology to provide a dynamic extension mechanism for the Labelled Transition System Analyser tool. A number of different extensions that have been developed as plugins were described.

The plugin mechanism allowed third parties, for example the researchers at NASA, to develop, test and deploy extensions to LTSA without having to have access to the source code of other components, or provide others with access to their confidential source code.

The fact that bindings can be made between any pairs of compatible components was used to build plugins that can use features provided by other plugins directly, if they are available, without having to communicate through the core of the application.

In order to ensure correct behaviour when different sets of plugins are combined, behavioural constraints were employed, specifying the expected behaviour of components.

8. Conclusions

In this final chapter we summarise and evaluate the contributions of this thesis, and discuss some possibilities for future work in the area.

8.1. Contributions

It is well accepted that the requirements for software systems change over time. Different users or customers may require specific customisations to create variations on an application. In many situations different configurations of components may be deployed at different sites to construct different software systems. If each custom solution is individually crafted by the software developer then this will lead to them having a large number of different builds to support and maintain. This thesis has investigated how software systems can be assembled by the end user or deployment engineer, rather than the developer, by plugging together sets of compatible components, and how the architecture of such a system can be allowed to evolve over time.

Two key questions have been addressed in this thesis. Firstly, what sort of programming model can be used to support this sort of application assembly? Secondly, how can the behaviour of such a system be predicted so that only systems that behave correctly are assembled?

We have shown how using a plugin architecture can allow systems to be created by the end user or deployment engineer by simply selecting sets of components and plugging them together without writing any extra program code. Support for runtime reconfiguration allows evolution of a system through the addition, removal or replacement of components. Plugin architectures are commonly associated with applications like web browsers. However, where these typically only allow extensions to a central

core, we permit systems to be formed from complex graphs of components. We have identified that the sort of evolution and reconfiguration that plugin architectures allow is good for a number of different types of software system. A plugin architecture is most effective where it is anticipated that there may be a need for some sort of extension at some future time, but where the exact details and configurations of how the architecture will evolve cannot be anticipated when the system is initially deployed.

We have presented a programming model for constructing plugin components in a lightweight fashion, without some of the extra machinery required by other component-based approaches. We have mapped concepts familiar from object-oriented programming to the provision and requirement of services. Our middleware platform, MagicBeans, supports programming in this model in the Java language. When new components are loaded the platform examines the code of the component to determine provided and required interfaces, matches these with the interfaces of other components present in the system and creates bindings between relevant pairs.

In dynamically assembling and evolving a system using components from different sources, it is difficult to know that the system will behave as expected after a reconfiguration. Adding a new component may cause a violation of a desired system property. We want to check this before making the reconfiguration rather than experiencing the results afterwards.

This thesis has presented techniques for automatically generating models of a running system, and of proposed reconfigurations of that system, in a form that can be analysed using model-checking. The automatic nature of the model generation means that this analysis can be performed even in the situation where the deployment agent is not skilled in formal modelling or analysis. We have presented techniques for creating a model in the Darwin ADL describing the structure of a system, extracting information from the code of the components and the runtime system. By combining this with behavioural information for each of the components, in the FSP process calculus, a model of the complete system can be formed that can be analysed using the LTSA model checker. If the analysis succeeds and no property violations are uncovered then the reconfiguration can proceed, otherwise it should be aborted.

8.2. Evaluation

In Chapters 1 and 2 we identified a number of different problems that needed to be solved in order to provide a technology that effectively allows systems to be composed and evolved dynamically while maintaining confidence that those systems will behave correctly. The main aims of the work presented in this thesis were to develop a system that allowed applications to be dynamically reconfigured by adding, removing and replacing modules, while maintaining a system that will preserve desired system properties, and hence not malfunction. Concerns relating to the dynamic aspects of software composition should be factored out into a reusable framework. Use of this framework for the application programmer should be as simple as possible. We will now consider how well the work addresses these issues. Each section is prefaced by the associated requirements as set out in Chapter 2.

8.2.1. Encapsulation

We require a system that will allow the creation of applications to be managed by the deployment engineer. This requires the coordination of components by a managing framework, without the deployer needing to write "glue code" to mediate between components.

As seen in Chapter 2, the problems of continually evolving software to update the available functionality can be addressed by providing software modules, or components, that can be added in to an existing software system. To make these modules easy to install, it is desirable that components can be integrated with a system without having to write any new code (commonly referred to as "glue code") to mediate between components. A component should encapsulate everything that it needs in order to be used.

After our experiences working with extensible systems, we support the proposition that the component level is the most appropriate level of granularity for adding functionality and reconfiguring systems. Object-oriented programming focusses on the class as its unit of encapsulation and modularity, but in general we have found that in

order to supply useful functionality, it is normally the case that sets of cooperating classes are required. Working with Java, we have used Jar archive files as our units of modularity, bundling together sets of classes to form deployable components. It should be easy to adapt the techniques that we have described to work with other component formats, for example .NET assemblies.

Using these deployable units as our components for constructing software systems allows systems to be assembled by the deployment engineer. The notion of *plugin* components allows modules to be added without the deployment engineer needing to have access to the source code of components, or to write any code themselves.

To construct the models that we use for the analysis of systems requires a specification of each component's behaviour. By including the behavioural specification within each component we maintain the possibility of deploying units that contain everything that is required for them to be used with the system. The component encapsulates the specification, which can be extracted by the runtime framework and combined with the generated structural model to create a full model of the system, without the deployment engineer needing to be aware of the details of how behaviour is specified.

The same mechanism is used to allow the framework to access a component's specification as is used for one component to access a service provided by another. This requires including within the component a class that implements a particular interface. This can mean that manipulating the specification is somewhat cumbersome as source code needs to be altered. However, this is not seen as a great problem, as it is expected that the specification will be written by the component's developer, who will be working with its source code anyway. Also, appropriate tool support can allow the boiler plate code in the specification class to be generated automatically.

8.2.2. Supporting Framework

We require a framework for plugin components that will support the construction of extensible applications in a generalised and reusable way.

We have implemented support for dynamism in plugin based systems in the Mag-

icBeans framework. This forms a middleware platform on top of which applications using plugin components can run.

The platform supports programming extensible applications using a third-party binding paradigm. The identification of components and the matching of interfaces is managed by the component framework, removing code for this concern from the user components. When a component becomes available that provides a service that a client can accept, the client will be notified of this by the runtime framework. One consequence of this is that the client programmer does not have explicit control over when components are connected or disconnected. It is not possible for user code to trigger a reconfiguration of the system. Although this may seem to be a limitation in some cases, for example if the application is monitoring its own performance in some way, we believe that encapsulating coordination and reconfiguration functionality in a lower layer of the system makes user level code cleaner and more maintainable.

8.2.3. Simplicity

We require integration with a familiar programming language, following its idioms, to make the use of a plugin system intuitive for the developer. This will also allow more use to be made of the compiler for static checking of component code.

We have aimed to make it easy for an application programmer to use the MagicBeans framework to integrate plugin components into their application. The programming examples in Chapter 4 show that the concepts from the plugin model presented in Chapter 3 fit quite naturally with the concepts familiar to a programmer using an object-oriented language. Service provision is expressed through interface implementation. Service requirement is expressed by writing a callback method whose parameter is of a type that indicates the service required. If, as in other approaches, services are located by querying a registry using a service name that is just a string, it is difficult to check type correctness statically. The use of language level types enables more checking of the program to be done by the compiler. This gives greater assurance of the correctness of code within a particular component.

All of the algorithms for matching provided and required services, creating bind-

ings and managing references between components are encapsulated in the plugin framework. The developer does not need to manage components explicitly, only to include methods to be executed on the arrival or departure of a required component. The third-party binding paradigm that has been adopted means that all responsibility for searching for components that may provide particular services is passed to the framework. Components do not need to behave in an active fashion, they will be informed of events relevant to them when they occur.

We do not prescribe the mechanism by which components are added to or removed from the system. There are a number of different possibilities for this, from locating plugins in a known file system location to using a dynamic network discovery service such as Jini [45]. In applications where the user administers the application's configuration themselves, for example most desktop applications used on personal computers, location of new plugins can be left to the user, who can provide the location of the component that they wish to install, perhaps in the form of a URL. Any mechanism that can detect the arrival or departure of a component can be used with our plugin framework. Support for different mechanisms can be added by using the plugin mechanism to extend the framework itself.

As our system requires inspection of a component's bytecode to identify provided and required interfaces, this can have an effect on the amount of time taken to load a component. Large components containing many classes may take several seconds to load. However, in practice we have not noticed any particularly detrimental effects on the performance of applications. Typically we have observed that applications composed from large components tend to be reconfigured less frequently than those composed from small components. Because of this, the load time overhead has not proved a problem. A few seconds delay is typically acceptable in a reconfiguration that only occurs once every couple of weeks.

8.2.4. Re-use

We require that component management code should be factored into a reusable framework that is not tied to any particular application. It should not be necessary

to duplicate component management code in multiple components.

The MagicBeans system is sufficiently general to allow many different applications to be built using it as their extension mechanism. This is in contrast to the technologies used to provide extension mechanisms for popular web browsers such as Netscape Communicator, which use a specialised mechanism (see Section 2.3.2). Internet Explorer uses Microsoft's ActiveX technology to allow plugins to be written. Although this is more general than Netscape's approach, and ActiveX is used in a number of different Microsoft products, in order to accept another component, a component must be, *i.e.* it must include the code for, an ActiveX container. What MagicBeans provides is a reusable platform that includes all of the code for matching interfaces and creating and managing bindings between components.

In the earlier discussion of encapsulation (Section 8.2.1), it was stated that it was desirable for a component to be a unit containing everything that that component required in order to be used. The factoring of component and system management code into a reusable layer seems to break this, as components cannot be completely autonomous. However, there is another angle to the encapsulation issue. The platform encapsulates the management code, meaning that code relating this concern does not need to appear in each of the client components. If we had opted for a first-party binding protocol, as seen in COM (see Section 2.1.2), then each component would have had to manage binding to its collaborators.

8.2.5. Dynamism

We require a generalised dynamic architecture that will permit these three types of change: dynamic addition, removal and replacement of components.

As discussed in Chapter 1, Oreizy *et al* identify three types of architectural change that are desirable at runtime [62]: addition, removal and replacement of components. The plugin model presented here, and its implementation in the MagicBeans framework, supports all of these different forms of evolution.

Components in a software system are, ideally, loosely coupled. This makes it particularly appropriate to support dynamic reconfiguration at the architectural level [23].

In this work we have presented a system where the separation between components is maintained by ensuring that all of the references across component boundaries are managed by the runtime framework. This allows us to perform replacement of components, as any binding or reference to a component that is to be replaced can be severed and reconnected to the replacing component.

We can generate models of the application during execution. This means that in deciding whether a component may safely replace another, we have knowledge beyond the interfaces that it provides and requires. It is possible to determine which of the component's provided services are actually being used in the current configuration. If components are considered in isolation, away from the system, then determining if one may replace another is a matter of checking whether it provides at least the services that are provided by the component it is to replace. With plugin architectures we consider all requirements to be optional, so it is not necessary to check that the requirements of the replacement are at most those of the component to be replaced, as would be the case in other component architectures, where requirements indicate strict dependencies. The use of runtime information allows us to be less conservative about deciding whether a component can safely replace another, without loss of confidence.

The rules that we have defined for the operation of a plugin system, in terms of which services are bound and how reconfiguration proceeds, are in a sense *laisser-faire*. Existing bindings are never broken during the addition process. Consider the situation where a system is running with two components. One requires a service and the other provides it, and hence they are bound. When a new plugin is added to the system, if that plugin provides the same service, the binding will not change. The new component is not considered to be "better" than the old (if the new component is intended to replace the old one, a replacement operation rather than an addition should be triggered).

We have tried to adopt a level of dynamism that allows for evolution while maintaining consistency, avoiding unnecessary change. Our experience in building applications using plugin architectures has shown that this approach is effective in enabling the development of applications that can be extended and reconfigured dynamically, while remaining stable and consistent from a user perspective.

Opting for a more aggressive reconfiguration strategy might allow existing bindings to be broken when a new component is connected to the system. This would allow for changes of configuration that we do not allow, for example inserting a component into the middle of a pipeline. Performing this change would require breaking and reconnecting the chain. At the present time we believe that allowing these more extensive architectural changes to occur could lead to inconsistent behaviour. For example, if a chain of components are passing data up and down, inserting a component into the chain in the middle of a transaction could cause unexpected processing, or the lack of expected processing. Perhaps in future the analysis techniques we have presented could be extended and refined to be able to deal with such situations.

8.2.6. Predictability

We require that a proposed configuration can be analysed for desired properties before it is realised, so that the deployer can gain confidence that the proposed reconfiguration will result in a correctly functioning system. It must be possible to abort the change if the result will violate system properties.

Assembling systems from sets of components with typed interfaces is akin to assembling object-oriented programs from collections of objects, whose available operations are expressed through types. We assemble applications from components by connecting ports with matching types. This ensures that applications are well typed; no action can be requested of an object that does not implement that action. This guarantee is attained as each component is well typed in itself (given that the code compiles), and the matching mechanism that is used in constructing plugin applications relies on the underlying language (Java in the MagicBeans implementation) type system. Only well typed applications will therefore be constructed.

However, having a program or application that is correctly typed does not guarantee that the program will behave in the desired fashion. The agreement of types says nothing about whether operations will in fact be invoked, or the ordering of those invocations. It is often the case that components must interoperate in accordance with a particular protocol in order for them to function correctly. To ensure correctness at this

level requires some further information and testing. We have demonstrated techniques through which candidate configurations of components can be analysed by generating models automatically at runtime. Structural information about the configuration of the system, including bindings and port type information, is combined with behavioural information for each component. This allows a behavioural model of the system as a whole to be created. Using model-checking tools we can verify that the components in the given configuration will fulfil required system properties. The LTSA tool allows us to easily check for deadlock and liveness in models. Although actual deadlock, experienced as an application freezing, is not very common in practice, the check for deadlock can also reveal other problems. For example a finite capacity buffer composed with a perpetual writer will eventually reach a deadlock as the buffer will become full and the writer will not be able to proceed. In this way we can use the deadlock check to detect possible buffer overruns.

Assessing the correctness of software is a difficult problem for programmers. Formal analysis techniques require a higher level of specification to be written than the program code. We have shown how some parts of a model can be generated automatically by the runtime system, but we still rely on the developer of each component providing a specification of that component's behaviour. While process calculus is not a widespread tool, and still considered "difficult" compared to programming in several spheres, we have provided tool support to help developers to write specifications for their components.

With larger models there can be significant computational overhead associated with model-checking. It may take a long time to check a large state space. As model-checking is a key part of our reconfiguration process this may mean that it is not possible to quickly swap components in and out while maintaining confidence in the system's correctness. At this point an engineering compromise needs to be reached concerning the importance of ensuring correctness versus the computational expense. In some cases, the user may decide to forego the model-checking stage if they are not overly concerned with maintaining behavioural properties. In other cases where correctness is critical, *e.g.* business critical servers or software for space probes, the cost of performing complex analysis is more likely to be deemed acceptable.

8.3. Future Work

We will now consider some possible future research directions arising from the work presented in this thesis.

8.3.1. Automatic Discovery of Behavioural Descriptions

The approach that we have presented extracts structural information from the code of a set of components, but relies on a developer writing and including a specification for their component in order to make use of behavioural information. Components that do not provide a behavioural specification can still be used with the system, but we are limited in the guarantees that can be made about the correctness of the resulting system.

Manual construction of such a model is both time-consuming and error prone. If it were possible to extract behavioural models automatically from the code of components, all components would be able to be included in the behaviour model. The Bandera project [20] aims to extract process models directly from Java code, so that models can be built and checked directly, without human intervention.

Future work could involve using techniques from Bandera to build analysable models directly from the code of components. The input to Bandera is Java source code, where we would prefer to work with the executable Java byte code that is found inside our components.

8.3.2. Use of Assume-Guarantee Reasoning

In the work that we have presented here, we combine the behavioural descriptions for all components and then check for a property. It is clearly possible that as the number of components in the system increases, the size of the model to be checked could become very large.

A possible approach for dealing with this problem might be the use of assume-guarantee reasoning. Work at NASA has produced techniques that can be used to learn assumptions that must hold for a component being added to a system in order

for a property to be preserved [19]. The component can then be checked against the assumption separately from the system, reducing the size of the state space.

With plugin systems, we are often considering the addition of just one component (rather than a complete reconfiguration of the system), so such a technique seems likely to yield good results in some cases.

8.3.3. Distribution

The MagicBeans system as described in Chapter 6 currently works only in a single address space. Many component based systems are designed to operate in a distributed environment, where different components operate in different address spaces, possibly distributed across a number of different hosts.

The model of plugin based systems that was described in Chapter 3 abstracts from the physical location of the components, describing them only as being inside or outside "the system". Therefore, no changes would need to be made to the abstract model in order to adapt the techniques described here to work in a truly distributed environment. All that would need to be altered would be details of the implementation. Some sort of messaging infrastructure would be required so that hosts could communicate in order to create a model of the complete system and make decisions about reconfigurations. The binding mechanism would need to be made more complex, so that calls could be made between components hosted on different machines. There are several possible mechanisms for providing such a facility, for example Java RMI.

8.3.4. Hierarchical Composition

Currently it is only possible to construct systems by adding one component at a time. There are situations where it would be preferable to be able to add a complete subsystem consisting of a number of components which are already bound together. The modelling system that we have presented already deals with the analysis of composite components, the system as a whole is currently modelled as a composite component, and so this could be extended to cater for varying levels of hierarchical composition in different subsystems. Doing this would also allow the behaviour of

subsystems to be analysed in isolation, which might be less computationally expensive than analysing the complete system.

8.3.5. Translation to Other Platforms

Throughout this thesis, the examples and implementation details given have been in the Java language. The plugin model presented could easily be translated to other platforms, a particular candidate at the present time being Microsoft's .NET platform. From an implementation perspective, .NET assemblies (components) have many similarities with Java's Jar archives, *e.g.* there is facility for inspecting their contents, and using reflection techniques to discover implemented interfaces. C# is a very similar language to Java, and so it should clearly be possible to use plugin concepts in C# programs as we demonstrated for Java in Chapter 4. However with .NET, as all source languages are compiled to a common intermediate language, there would also be the increased flexibility of being able to write plugin components in different languages, as long as they were able to provide functions that the plugin framework could call, and having them interoperate.

It might also be possible to use the analysis techniques developed in this thesis to check the correctness of other systems of components. Component models such as OSGi (see Section 2.2.1) are being adopted widely, and so if it were possible to produce models from sets of OSGi components as well as sets of MagicBeans components, then the range of systems that could be analysed would be greatly increased.

8.4. Closing Remarks

We will never reach the stage where all the requirements for a software system can be predicted at the outset of a project. There will always be changes in requirements over time. Modern software systems are large and are responsible for many facets of daily life. The costs associated with rebuilding them from scratch in response to new requirements are high and often prohibitive. Provision for change is therefore a

fundamental requirement in the design of such systems. The need for software that is adaptable and upgradeable, and for technologies to help manage change, is growing day by day.

Adaptability in software can lead to every installation of an application being slightly different. While this has the benefit of allowing systems to be tailored to a particular user's needs, it also moves control of the configuration out of the hands of the software developer and further towards the user. This limits what use traditional software engineering techniques can be in ensuring the correctness of applications, as they are no longer built by experts in Software Engineering, but by system administrators and end users.

The work presented in this thesis shows that the conflict between the goals of adaptability and correctness can be surmounted by providing runtime support for the modelling and analysis activities that more traditionally occur at design time.

Bibliography

[1] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[2] F. Barbieri. Avalon Planet. Technical report, Apache Software Foundation, http://avalon.apache.org/planet/components/index.html, 2004.

[3] M. Barnett. private communication, 2003.

[4] M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tillmann, and A. Watson. Serious specification for composing components. In *6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, 2003.

[5] M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, Nov. 2001.

[6] M. Barr and S. Eisenbach. Safe Upgrading without Restarting. In *IEEE Conference on Software Maintenance (ICSM 2003)*. IEEE, Sept 2003.

[7] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(4):386–426, 2002.

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

[9] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *Proceedings of the first workshop on Self-healing systems*, pages 9–14. ACM Press, 2002.

[10] H. Cervantes and R. S. Hall. Automating service dependency management in a service-oriented component model. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering*, 2003.

[11] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings of the 26th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, may 2004.

[12] R. Chatley, S. Eisenbach, and J. Magee. Modelling a framework for plugins. In M. Barnett, S. H. Edwards, D. Giannakopoulou, and G. T. Leavens, editors, *SAVCBS*, volume #03-11 of *Technical Report*. Iowa State University, 2003. http://www.cs.iastate.edu/∼leavens/SAVCBS/2003/papers/SAVCBS03.pdf.

[13] R. Chatley, S. Eisenbach, and J. Magee. MagicBeans: a Platform for Deploying Plugin Components. In *Second International Working Conference on Component Deployment*, May 2004.

[14] R. Chatley, S. Eisenbach, and J. Magee. Predictable Dynamic Plugin Systems. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2004)*, Barcelona, Spain, March 2004.

[15] R. Chatley, J. Kramer, J. Magee, and S. Uchitel. Model-based Simulation of Web Applications for Usability Assessment. In *Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, May 2003.

[16] R. Chatley, J. Kramer, J. Magee, and S. Uchitel. Visual methods for web application design. In M. Burnett and J. Grundy, editors, *Proceedings of 2003 IEEE Symposium on Visual and Multimedia Software Engineering October 28-31 Auckland, New Zealand*, 2003.

[17] R. Chatley, S. Uchitel, J. Kramer, and J. Magee. Fluent-based web animation: Exploring goals for requirements validation. In *Proceedings of the 27th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2005)*, St Louis, Missouri, USA, may 2005.

[18] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[19] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning Assumptions for Compostional Verification. In *Proc. of TACAS 2003*. LNCS, April 2003.

[20] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[21] D. Crawford. *Commun. ACM - Special issue on Aspect Oriented Programming*. ACM Press, October 2001.

[22] M. Dahm, J. van Zyl, and E. Haase. The Byte Code Engineering Library (BCEL). Technical report, Apache, http://jakarta.apache.org/bcel/, 2002.

[23] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26. ACM Press, 2002.

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

[25] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is java binary compatibility? In *OOPSLA*, pages 341–361, 1998.

[26] S. Eisenbach, V. Jurisic, and C. Sadler. Feeling the way through DLL Hell.

In *The First Workshop on Unanticipated Software Evolution (USE 2002)*. http://joint.org/use2002/, June 2002.

[27] S. Eisenbach, C. Sadler, and S. Shaikh. Evolution of Distributed Java Programs. In *IFIP/ACM Working Conference on Component Deployment (COCD 2002)*, volume 2370 of *LNCS*. Springer-Verlag, June 2002.

[28] S. Eisenbach, C. Sadler, and S. Shaikh. Evolution of Distributed Java Programs. In *IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *LNCS*. Springer-Verlag, June 2002.

[29] H. Evans. DRASTIC and GRUMPS: design and implementation of two run-time evolution frameworks. *IEE Proceedings - Software Engineeering*, 151(02):30–48, march 2004.

[30] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montral, Canada, Nov. 2003. IEEE Computer Society Press.

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

[32] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 175–188, 1994.

[33] D. Garlan, D. Kindred, and J. Wing. Interoperability: Sample Problems and Solutions. Technical report, Carnegie Mellon University, Pittsburgh, 1995.

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

[35] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32. ACM Press, 2002.

[36] I. Georgiadis. *Self-Organising Distributed Component Software Architectures*. PhD thesis, Imperial College London, London, United Kingdom, Jan 2002.

[37] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM Press, 2002.

[38] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Bchi automata. In *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, November 2002.

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2 edition, June 2000.

[40] D. Green. The Reflection API. Technical report, Sun Microsystems, Inc., http://java.sun.com/docs/books/tutorial/reflect/, 1997-2001.

[41] R. S. Hall. Oscar. Technical report, ungoverned.org, oscar-osgi.sourceforge.net, 2003.

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

[43] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *Proceedings of the 22nd Conference on Software Engineering (ICSE'2000)*, pages 730–733, Limerick, Ireland, May 2000. ACM Press.

[44] Javabeans. The Only Component Architecture for Java Technology. Technical report, Sun Microsystems, Inc., java.sun.com/products/javabeans/, 1997.

[45] JINI. DJ - Discovery and Join. Technical report, Sun Microsystems, Inc., wwws.sun.com/software/jini/specs/jini1.2html/discovery-spec.html, 1997-2001.

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

[47] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.

[48] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng.*, 21(4):336–355, 1995.

[49] Macromedia. Macromedia Shockwave Player. Technical report, Macromedia, Inc., www.macromedia.com/ software/shockwaveplayer/, 1995-2003.

[50] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Conference on Software Engineering, Sitges, Spain, 1995*, pages 137–154. Springer Verlag, 1995.

[51] J. Magee and J. Kramer. *Concurrency – State Models and Java Programs*. John Wiley & Sons, 1999.

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

[53] N. Medvidovic, D. Rosenblum, and R. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99*, 1999.

[54] Microsoft Corporation. The Component Object Model: A Technical Overview. Technical report, Microsoft Developer Network, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn-comppr.asp, 1994.

[55] Microsoft Corporation. How to Write and Use ActiveX Controls for Windows CE 2.1. Technical report, Microsoft Developer Network, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnce21/html/activexce.asp, 1999.

[56] H. Muccini. Detecting Implied Scenarios analyzing non-local Branching Choices. In *Proc. of FASE 2003*. LNCS, April 2003.

[57] G. C. Necula and P. Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[58] Netscape. Netscape Communicator Plug-in Guide. Technical report, Netscape, http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm, 1998.

[59] B. Nuseibeh. Weaving together requirements and architecture. *IEEE Computer*, 34(3):115–117, March 2001.

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

[61] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

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

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

[64] OSGi. Open services gateway initiative specification. Technical report, OSGi, http://www.osgi.org, 2001.

[65] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, 1977.

[66] N. Pryce and S. Crane. A model of interaction in concurrent and distributed systems. *Lecture Notes in Computer Science*, 1429:57–??, 1998.

[67] N. Pryce and J. Magee. Scenebeans: A component-based animation framework for java.

[68] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Using critics to analyze evolving architectures. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 90–93. ACM Press, 1996.

[69] Sun. Java platform debugger architecture. Product documentation, Sun Microsystems, Inc., java.sun.com/j2se/1.4/docs/guide/jpda/index.html, 2002.

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

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

[72] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.

[73] The Mozilla Organisation. Firefox - the browser, reloaded, 2004.

[74] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In *Proc. of TACAS 2003*. LNCS, April 2003.

[75] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. Fluent-based animation: Exploiting the relation between goals and scenarios. In *Proceedings of the IEEE International Conference on Requirements Engineering (RE 2004)*, Kyoto, Japan, September 2004.

[76] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. System architecture: the context for scenario-based model synthesis. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, Newport Beach, CA, USA, November 2004.

[77] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 74–82. ACM Press, 2001.

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

[79] W3C. Scalable Vector Graphics (SVG) 1.0 Specification. Technical report, W3C, http://www.w3.org/TR/SVG/, 2001.

[80] P. Waesawangwong. A constraint architectural description approach to self-organising component-based software systems. In *Proceedings of the Doctoral Symposium at the 26th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, May 2004.

[81] D. Watkins, M. Hammond, and B. Abrams. *Programming in the .NET Environment*. Addison Wesley, April 2004.

[82] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering*, pages 314–323. ACM Press, 2000.

[83] W. Wong. Analysing Clocked Process Algebras via Stochastic Petri Nets. Technical report, Imperial College London, http://www.doc.ic.ac.uk/ ajf/Teaching/Projects/Distinguished04/WallaceWong.pdf, 2004.

[84] D. Zowghi, A. Ghose, and P. Peppas. A Framework for Reasoning about Requirement Evolution. In N. Y. Foo and R. Goebel, editors, *Proceedings of*

the 4th Pacific Rim International Conference on Artificial Intelligence, Cairns, Australia, 1996, pages 157–168. Springer Verlag, 1996.

A. 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
  no start.holes => no bindings
  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 }
}

fun canBind( cl:Class, c1:Component, i:NumInterface, c2:Component ) {
  i in c2.holes && cl in c1.pegs && (c1 != c2) &&
  no cl.abstract && i in cl.implements
}



fun 1-1-binding( s : System ) {
  all b1 , b2 : Binding { b1 in s.bindings && b2 in s.bindings &&
                    (b1.to = b2.to) && (b1.peg = b2.peg) => (b1 = b2) }
}

fun 1-n-binding( s : System ) {
  all b1 , b2 : Binding { b1 in s.bindings && b2 in s.bindings &&
                    (b1 != b2) && (b1.to = b2.to) &&
                    (b1.peg = b2.peg) => (b1.from != b2.from) }
}

fun load( s , s' : System , c:Component) {
  s'.start = s.start
  s'.bindings = s.bindings
  s'.comps = s.comps + c
}
```

B. Implementing a NullObject factory

```
public class NullObjectFactory {

  /**
   * This class automatically generates null implementations of
   * a given Java interface on the fly - implementing the
   * NullObject pattern.
   * Calling a method on the NullObject should either:
   *  - do nothing
   *  - return 0 or a similar value
   *  - return a NullObject representing the return type of
   *    the method
   */

  public Object newNull( Class p_intf ) {

    /**
     * Set up a proxy for the given interface, with method
     * calls to be handled by a NullInvocationHandler
    **/

    InvocationHandler x_ih = new NullInvocationHandler();

    return = Proxy.newProxyInstance( p_intf.getClassLoader() ,
                              new Class[] { p_intf } ,
                              x_handler );
  }
}


class MyInvocationHandler {

   /**
   * Depending on the return type of the method, either
```

```
  * do nothing, return a sentinel 0 or ' ' or similar,
  * or, for reference types, reuse the factory to create
  * a null implementation of the interface type, and return
  * that.
  **/

  public Object invoke( Object p_proxy ,
                Method p_meth ,
                Object[] p_params ) {

   if ( p_meth.getReturnType() == Void.TYPE ) { return null; }
   if ( p_meth.getReturnType() == Integer.TYPE ) { return new Integer(0); }
   if ( p_meth.getReturnType() == Double.TYPE) { return new Double(0.0); }
   if ( p_meth.getReturnType() == Float.TYPE) { return new Float(0.0); }
   if ( p_meth.getReturnType() == Short.TYPE) { return new Short(0); }
   if ( p_meth.getReturnType() == Long.TYPE) { return new Long(0.0); }
   if ( p_meth.getReturnType() == Byte.TYPE) { return new Byte(" "); }
   if ( p_meth.getReturnType() == Character.TYPE) { return new Character(' '); }

   if ( p_meth.getReturnType().isInterface() ) {

    return newNull( p_meth.getReturnType() );
   }

   return null;
  }
}
```

C. Full Compressing Proxy Model

```
interface Filter { data; }
interface Proc { packet; end; }

component Source {

   require next:Filter;
   /%
   Source = ( next.data -> Source ).
   %/
}

component BasicFilter {

   require next:Filter;
   provide Filter;

   /%
   BasicFilter = ( data -> next.data -> BasicFilter ).
   %/
}

component Adapter {
   require next:Filter;
         out:Proc;
   provide Filter;
         Proc;
   /%
   Adapter   = BasicFilter,
   BasicFilter = ( data -> next.data -> BasicFilter
              | pluginAdded.proc -> Adapt
              ),
   Adapt     = ( data -> out.packet -> ToProc ),
   ToProc    = ( out.packet -> ToProc | out.end -> FromProc ),
   FromProc = ( packet -> FromProc | end -> next.data -> Adapt ).
   %/
```

```
}

component GZip {

   require out:Proc;
   provide Proc;

   /%
   GZip2   = ( packet -> put -> In ),
   In      = ( packet -> put -> In | end -> Zip ),
   Zip     = ( zip -> Out ),
   Out     = ( get -> out.packet -> Out | out.end -> GZip2 ).

   Buffer(N=2) = Count[0],
   Count[i:0..N] = ( when (i<N) put -> Count[i+1]
                 | when (i>0) get -> Count[i-1]
                 | when (i==0) out.end -> Count[i] ).

   ||GZip = ( GZip2 || Buffer ).
   %/
}

component PacketAdapter {

   require next:Filter;
         out:Proc;
   provide Filter; Proc;

   /%
   PacketAdapter = ( data -> out.packet -> out.end -> packet
                 -> end -> next.data -> PacketAdapter ).
   %/
}
\end{verbatim}
\clearpage
\begin{verbatim}
//system with deadlock
component System {

   inst s:Source
      b:BasicFilter
      a:Adapter;
      g:GZip;
      b2:BasicFilter;

   bind b.Filter -- s.next;
      a.Filter -- b.next;
```

```
      b2.Filter -- a.next;
      a.Proc -- g.out;
      g.Proc -- a.out;
}

//working system
component System {

   inst s:Source
      b:BasicFilter
      a:PacketAdapter;
      g:GZip;
      b2:BasicFilter;

   bind b.Filter -- s.next;
      a.Filter -- b.next;
      b2.Filter -- a.next;
      a.Proc -- g.out;
      g.Proc -- a.out;
}
```

D. Full LTSA Model

```
interface LTSAPlugin { init; run; cleanup; }
/%
const PLUGINS_MAX = 2
%/

component LTSACore {

  require plugins[2]:LTSAPlugin;
  /%
  LTSA_PLUGIN = (init -> run -> sync -> cleanup -> END).
       ||LTSACore = forall [i:1..PLUGINS_MAX ] plugins[i]:LTSA_PLUGIN/{sync/plugins[i
            ].sync}.
  %/
}

component MSCPlugin {

  provide LTSAPlugin;
  /%
  MSCPlugin = ( init -> run -> cleanup -> MSCPlugin ) + LTSAPlugin.
  %/
}

component DarwinPlugin {

  provide LTSAPlugin;
  /%
  DarwinPlugin = ( init -> run -> DarwinPlugin ) + LTSAPlugin.
  %/
}

component System {

  inst core:LTSACore;
       msc:MSCPlugin;
       dwn:DarwinPlugin;
```

```
  bind core.plugins[1] -- msc.LTSAPlugin;
       core.plugins[2] -- dwn.LTSAPlugin;
}

/%
assert ALLCLEANUP = forall [j:1..PLUGINS_MAX] <>(core.plugins[j].cleanup)
%/
```

E. HotSwap Experiments

Hotswapping is a technique for replacing class definitions inside a running Java Virtual Machine. It was designed by Mikhail Dmitriev, and more information on using it is available in Dmitriev's technical report [24].

It is not possible to swap in a new definition for a class which has arbitrary differences to the original version. Here we see what is possible, what is not, and whether everything that is possible will work correctly.

E.1. A framework for testing

To make it easy to run different tests, we provide a managing application which does the following:

- Loads a jar file old.jar containing a set of classes that form a program. One of the classes must be called Main and have a method called run().

- Instantiates the Main class and runs the program by.

- Loads a jar file new.jar containing a set of classes (a subset of the classes in old.jar) that should replace the original class definitions.

- Calls HotSwap to redefine the classes.

- Calls the run() method on the existing instance of Main.

E.2. Tests

For each experiment we present the source code for the old (already loaded) and new (to be upgraded to) versions of the class, and the results of running the experiment, showing whether or not such a change is a possible upgrade under the current (1.4.0) Hotswap implementation.

E.2.1. Changing a method body

Old version:

```
class Main {
   void run() {
      System.out.println(``old version'');
   }
}
```

New version:

```
class Main {
   void run() {
      System.out.println(``new version'');
   }
}
```

Result:

```
old version
new version
```

Changing the implementation of a method body is a possible change.

E.2.2. Adding a method

Old version:

```
class Main {
   void run() {
      System.out.println(``old version'');
   }
}
```

New version:

```
class Main {
   void run() {
      System.out.println(``new version'');
   }
   void m1() {}
}
```

Result:

```
old version
That upgrade is not supported
old version
```

Adding a method is not possible.

E.2.3. Adding a field

Old version:

```
class Main {

   void run() {
      System.out.println(``old version'');
   }
}
```

New version:

```
class Main {
   int i;
   void run() {
      System.out.println(``new version'');
   }
}
```

Result:

```
old version
That upgrade is not supported.
old version
```

Adding a field is not a possible change.

E.2.4. Changing the order of methods

Old version:

```
class Main {
   void run() {
      System.out.println(''old version'');
      m1();
      m2();
   }
   void m1() { System.out.println( ''m1'' ); }
   void m2() { System.out.println( ''m2'' ); }
}
```

New version:

```
class Main {
   void run() {
      System.out.println(''new version'');
      m1();
      m2();
   }
   void m2() { System.out.println( ''m2'' ); }
   void m1() { System.out.println( ''m1'' ); }
}
```

Result:

```
old version
m1
m2
new version
m1
m2
```

Changing the order of the methods in a class is a possible change.

E.2.5. Changing the order of fields

Old version:

```
class Main {
   int i;
   int j;
   void run() {
      System.out.println(``old version'');
   }
}
```

New version:

```
class Main {
   int j;
   int i;
   void run() {
      System.out.println(``new version'');
   }
}
```

Result:

```
old version
That upgrade is not supported
old version
```

Changing the order of the fields in a class is not a possible change.

E.2.6. Changing the order of methods in a superclass

Old version:

```
public class Main {
   A a;
   public Main() { a = new B(); }
   void run() {
      System.out.println(``old version'');
      a.m1();
   }
}

public class A {
   void go(){}
   void m1(){ System.out.println(``m1''); }
   void m2(){ System.out.println(``m2''); }
}

public class B extends A {
   void go() { m1(); }
}
```

New version:

```
public class A {
   void go(){}
   void m2(){ System.out.println(``m2''); }
   void m1(){ System.out.println(``m1''); }
}
```

Result:

```
old version
m1
old version
m1
```

Changing the order of the methods in a superclass still seems to work. (The result prints “old version” twice because we only upgrade class A). We suspected this might not work, but it does seem to...

E.2.7. Calling a method in a new class

Old version:

```
class Main {
   void run() {
      System.out.println(``old version'');
   }
}
```

New version:

```
class Main {
   void run() {
      System.out.println(``new version'');
      new AnotherClass().m1();
   }
}

class AnotherClass {
   public void m1() {
      System.out.println(``m1'');
   }
}
```

Result:

```
old version
new version
m1
```

The HotSwap succeeds, and if AnotherClass can be found on the classpath, then it is loaded an m1() is called as required.

E.2.8. Calling a method in a new class that is missing.

Old version:

```
class Main {
   void run() {
      System.out.println(``old version'');
   }
}
```

New version:

```
class Main {
   void run() {
      System.out.println(``new version'');
      new AnotherClass().m1();
   }
}
```

Result:

```
old version
new version
ClassDefNotFound : AnotherClass
```

The HotSwapping of the method proceeds ok, and the linking fails when AnotherClass cannot be found. It is not possible to HotSwap in a class that does not already exist inside the virtual machine.