Imperial College London
Department of Computing

A Rigorous, Architectural Approach to Extensible Applications

Andrew McVeigh

Abstract

An extensible application is one which allows functionality to be added, replaced or removed without requiring the source code of the application to be revealed or modified. The aim is to enable developers to add features and customise a substantial base application for new requirements, without the direct involvement of the application creators.

A recurring theme of existing extensibility approaches is that an application must provide pre-planned extension points to accommodate expected future extensions. This results in a tension between keeping the architecture simple and potentially inextensible, or providing many predictive extension points that cannot be guaranteed to cover all future requirements despite best intentions.

The Backbone component model is presented as an architectural approach which addresses these issues. By augmenting an architecture description language with a small set of constructs for modelling structural change, extensibility is naturally built into an application as it is elaborated into a compositional hierarchy. An extension can then restructure any part of the application architecture it builds on, to meet new requirements.

The key contribution of this work is the consideration of both planned and unplanned extensibility in a hierarchical component model. A formal specification is given, describing the way that extensions can alter an architecture, and how extensions can be combined in a way which resolves any structural conflict.

Tool support is provided by a UML-based modelling workbench and runtime platform, developed from the specification. Integration with existing implementation components and their subsequent evolution is fully supported.

For evaluation, the model is compared and contrasted with other approaches, specifically plugin architectures and product lines. Backbone is also used to restructure and extend a mature system. These studies demonstrate that Backbone supports unplanned extension with the proviso that if existing leaf components are not granular enough, then some reimplementation may be required.

Acknowledgements

I would like to express my gratitude to my supervisors, Jeff Magee and Jeff Kramer, for their guidance and constant encouragement, and the sharing of their deep insights into component structures and software architecture. Their work and research has inspired, educated and humbled me. They have also helped me to grasp the deep and sometimes subtle relationship between intuitive understanding and formal specification in software engineering. Their helpful critique has improved my writing style immeasurably.

I would also like to thank Susan Eisenbach, who has supported my lectures on the background to this thesis and on compositional modelling. Special thanks to Naranker Dulay, who spent much time explaining to me the details and history of the Darwin implementations.

Thanks must be extended to my work colleagues at the various companies that I did contract work for during my studies. In particular Jitendra Shah, Kevin Smith, Julie Evans, Maria Buckley, Steve Ashcroft and Jeremy Wilks have kindly allowed me to work whenever I needed or wanted to. Special thanks to Markus Tesch for agreeing to the amazingly flexible arrangement in the first place.

My family have been a source of constant encouragement throughout. My wife Terri has supported me immensely during this time and has always realised how important this work has been to me. My two children, Maddie and Belle, have been wonderful and very understanding when the studies meant that I could not spend so much time taking them to the park or playing on the trampoline with them!

Last, but not least, I would like to thank Ian Hodkinson for his helpful words of advice in my first year.

Dedication

This thesis is dedicated to my lovely wife Terri, and my two wonderful daughters Annabelle and Madeline. Thanks to God for an answer to prayer, and for the fortuitous appearance of a small book on software architecture in an obscure second-hand bookshop in Perth.

This thesis is also dedicated to my mum, who was so proud of my work. Rest in peace.

Contents

  3.1 The Three Key Extensibility Concepts 37

  3.2 The Backbone Component Model 41

    3.2.1 Core Constructs 42

    3.2.2 Extensibility constructs 46

    3.2.3 Summary 49

  3.3 Modelling the Audio Desk Scenario in Backbone 50

    3.3.1 Relationships in an Extension Setting 50

    3.3.2 Using Strata to Model Ownership and Relationships 51

    3.3.3 The Base Application 52

    3.3.4 The CD Player Extension 55

    3.3.5 The Turntable Extension 56

    3.3.6 The Upgraded Desk 59

    3.3.7 Combining All Extensions to Form a Coherent Application 61

  3.4 Summary 64

4 An Outline of the Backbone Formal Specification **67**

  4.1 Organisation of the Formal Specification 68

  4.2 Key Concepts of the Base Layer 69

    4.2.1 Stratum and Dependency Rules 69

    4.2.2 A Strata Dependency Example 71

    4.2.3 Elements 73

    4.2.4 Deltas 75

  4.3 Key Concepts of the ADL Layer 78

    4.3.1 Components 78

    4.3.2 Interfaces 79

    4.3.3 Human Readable Names and Renaming 80

List of Tables

List of Figures

Chapter 1

Introduction

It seems impossible to predict change, and very often, a successful system will change in ways which are inconceivable to the original designers.

Keith Bennett et al [BBHL04]

Any time you build an extensible system ... you can't predict in advance what changes your users will want to make to your system. You can go to any length — even expose every single line of code in your system as its own virtual function — and your users will inevitably run into something that they can't specialize.

Steve Yegge [Yeg09]

1.1 Motivation

An extensible application is one which allows functionality to be added, replaced or even removed without requiring modification or access to its source code [KF98]. The benefit of architecting a system in this style is that it allows developers to create an extension, which adds features to a base application and customises it, without further involvement from the original creators of the application. The extensible system forms a platform on which a family of applications can be built, servicing a much larger market than would otherwise be possible.

Eclipse [Obj09f, Obj09h], Firefox [Moz09a, Moz09b], Emacs [Bla09] and Excel [BBG05] are prominent examples of this approach. These applications support extensibility via (sometimes subtly) different mechanisms, but the basic concept is the same. They allow third party developers to add to (and sometimes remove from) a base application in order to extend it, and each has spawned a significant set of extensions. The extension concept has shown itself to be of significant value: the original application creators can service a broadened market, extension developers gain a mature base to build on, and end users get increased choice in order to meet their needs. Interestingly, even though the implementation source code is freely available for three out of the four example applications mentioned at the start of the paragraph, this does not diminish the need for, or effectiveness of, an extensibility approach. Extending a substantial application is usually simpler, and incurs less maintenance overhead, than directly altering the source code of the system.

Unfortunately, however, existing extensibility approaches have limitations and unwelcome side effects. Most of the approaches assume that pre-planned hooks or extension points will be built into the base application in order to accommodate for any possible future extension requirements. Predicting these points in advance can never cover all eventualities [BBHL04], and building excess points into the application "just in case" pollutes the architecture and reduces clarity and manageability. Other approaches construct an application as a fine-grained arrangement of components, allowing arbitrary replacement of any part, but end up exposing great complexity to the application creators and extension developers alike.

Developers building on an extensible base application may also wish to remove or replace existing functionality in order to simplify and streamline the system and improve performance [Par78], even as they add other features. Current approaches either forbid this or only support this implicitly via hiding the presence of features or by allowing end users to perform deletion of previously installed parts of the application.

There is a further, more pernicious, problem. The ability to extend an application leads naturally to the requirement to combine two or more independently-developed extensions (and base application alterations) into a single application. However, as extension developers are given more flexibility to adapt and customise the base, the potential increases for extensions to conflict when they are combined. This tension is often resolved by placing unwelcome limitations on extensions, in part because conflict between them is hard to detect and correct.

These issues and limitations present real difficulties when creating, extending and upgrading an extensible application. We rephrase these concerns as the following research question.

> Can we devise an architectural approach to software that naturally builds extensibility into a system as it is constructed, which also respects the underlying forces and constraints between extension and base application developers?

We investigate this question by presenting an example development scenario, and using it to discern the underlying issues that arise between developers in an extensibility setting.

1.2   Example Scenario

The Audiosoft company develops "desk version 1.0", an extensible application for controlling digital audio devices via a software mixer[1]. The application is used by radio studios to control a set of devices for on-air transmission, and comes configured with two microphone devices by default.

Other developers can extend this base application to add further features, as shown in figure 1.1. The reason for developing the application as extensible is that Audiosoft does not have the resources to add

---

[1]This example is simplified from a commercial product that the author previously worked on.

a software controller for every audio device on the market, and wishes to allow third party developers to accomplish this task instead.

Audiosoft does not wish to release the implementation code for the product due to several proprietary algorithms used in the mixer. For the sake of the example, suppose that the application is complex and contains many components at varying levels of abstraction.
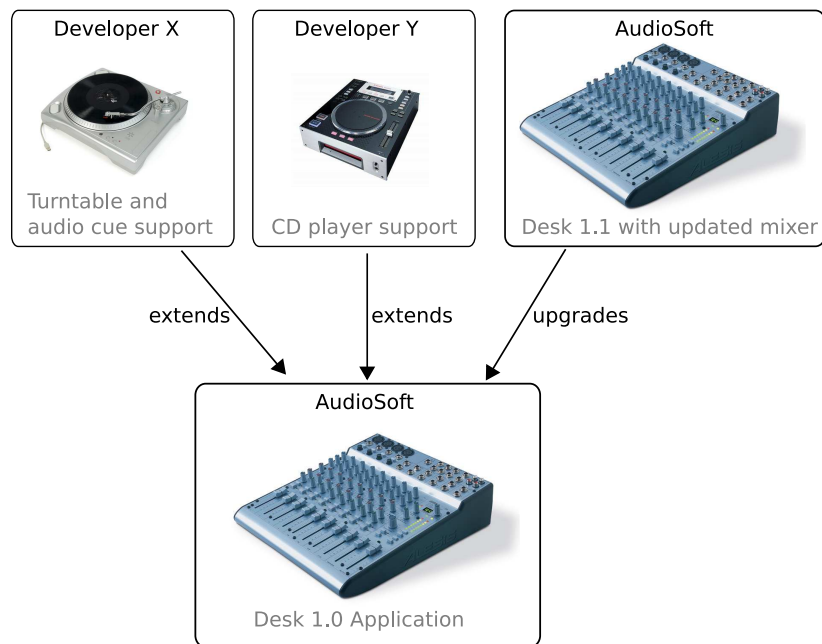
Figure 1.1: The desk application with various extensions

Developer X is a third party developer contracted to add support for a turntable to version 1.0 of the desk application. However, there is a problem. The turntable requires audio cuing, where audio can be sent to an off-air bus so that the start of a track can be found. Desk version 1.0 does not provide a cue feature, and as a consequence X must somehow adjust the base application to add support for this before integrating in the turntable device controller.

Another developer (Y) decides to extend the application by adding a software device controller for a CD player to desk 1.0. In the meantime, Audiosoft upgrades the desk application to version 1.1, which features an improved mixer.

Finally, a radio station (R) wishes to use version 1.1 combined with the turntable and CD devices in a single desk.

1.3 Requirements for Extensibility

Even though this scenario is relatively simple, it presents clearly the challenges that an extensibility approach must deal with. This section analyses the scenario in order to distill requirements for an ideal extensibility mechanism.

Firstly, even though the desk application doesn't provide audio cuing, developer X must add this in order to allow the turntable device to be fully supported. In this case, the existing extension points and parametrisation in the desk 1.0 application cannot support this. It is very common for an extension developer to be faced with the situation where the design of the base simply has not foreseen the needs of the extension. This leads to our first requirement:

> Requirement: **ALTER**
> It should be possible to alter a base application to accommodate new features introduced by an extension. The changes to the base application may be unplanned, and therefore the ability to make arbitrary changes is required. The extension developer should be afforded the same power to amend the architecture that the developers of the base application had when they created it. Further, the alterations required for an extension should be proportionate to the functionality added or changed.

The key point is that the required changes may be unanticipated by the base developers (unplanned), and will require (possibly extensive) base changes which could involve addition, deletion or replacement of parts of the base. As an example of the need for deletion, consider that an extension developer adding output limiting to the desk may wish to remove the mixer's existing frequency equalisation capability.

Although we need to make potentially deep changes to accommodate an extension, as per the earlier definition of extensibility the system must be able to be extended without requiring modification or access to its source code. In some cases the driver for this is that the implementation is proprietary. Another goal is to shield extension developers from the full complexity of the application at the source code level.

> Requirement: **NoSource**
> It should be possible to extend the base application without requiring modification or access to its implementation source code.

It is worth unpacking this requirement, as it may seem at first glance that we are unnecessarily constraining ourselves. By modifying the source code, it is true that any required alterations could be made for an extension. However, giving extension developers full access to the source code and allowing them to make the requisite changes (via source code control branching or simply "copy and paste") creates many variants of the base application, causing maintenance and ongoing evolution issues.

Furthermore, the ability to make extensions without modifying the base source code is important even in organisations which have full rights to modify the source. This is because despite best intentions, producing a new variant quickly for time-to-market concerns usually involves a fork of the codebase to prevent impact on existing clients. By allowing extension without base source code modification, we prevent the need for proliferation of base variants and the difficulties involved with the management, upgrade and and later merging of these.

Another potential alternative, given that the base source is not available to extension developers, is to ask the base developers to evolve the application to incorporate a new set of extension points. However,

this causes a problem for the base application as the architecture gets polluted with extension points of each and every extension. This is not tenable if there are many extensions, particularly as it pushes all of the maintenance and variant management work onto the base developers. It also results in an ongoing stream of evolutions to the base, which are visible to all parties even if they are not interested in the extension that necessitated the changes. Existing users of a base will not want to have to upgrade and retest on a new base with changes that have only been made to support a feature needed for another user.

These points lead to the next requirement, which is that those who do not care about a particular base alteration required for an extension should neither be required to have visibility of the change, nor be constrained by it.

> Requirement: **NoImpact**
> The changes made to the base for an extension should not constrain the ongoing development of the base, nor the development of any unrelated extensions. An extension should not place the maintenance burden for any part of the added features onto the developers of the base application.

To illustrate the above requirement via our scenario, consider that Audiosoft does not wish to see the changes that X introduces for audio cue support, as these are not planned for incorporation into the general desk product. Similarly, Y does not wish to see or have to adapt their extension for the changes that X wants to make to the base application.

However, although we require that those who do not wish to see changes are not affected by them, at some point we may wish to combine multiple extensions into a single application. In our scenario, the radio station wishes to combine the CD player and turntable support. This leads to a further requirement.

> Requirement: **Combine**
> It should be possible to combine multiple, independently developed extensions to a base application, so as to form a unified system. It should be possible to detect and correct any conflicts between extensions, including conflicts between the base alterations required for each extension.

Although all of the extensions discussed so far have been built on the desk version 1.0 application, we cannot ignore the evolution of the base. As stated earlier, Audiosoft eventually decides to upgrade the base application with an improved mixer. It may do this by evolving the application using traditional source modification techniques, or it may structure the upgrade as a further extension. Either way, we would like to be able to use existing extensions with the new upgrade, leading to our next requirement.

> Requirement: **Upgrade**
> It should be possible to combine existing extensions with an upgraded or evolved version of

the base application. It should be possible to detect and correct any conflicts caused when existing extensions are combined with the upgraded base.

Both COMBINE and UPGRADE require the ability to detect and correct any conflicts between independently developed extensions to the system. In order to emphasise this important notion, we elevate it into a separate requirement.

Requirement: **DETECTANDCORRECT**

It should be possible to detect and correct any conflicts that occur when upgrades, and independently-developed extensions (and their alterations to the common base) are combined into a single application.

Our final requirement is that extensions can be added at the appropriate level of abstraction, without necessarily exposing the extension developer to the full complexity of the system. If a complex application consists only of very coarse-grained components, then making a simple unplanned change targeted at a low level of abstraction may involve an inordinate amount of work and be very difficult to achieve: we will need to replace a lot of functionality for a small change. On the other hand, if an application contains many fine-grained components, it will be simpler to add a small extension, but the architecture will be difficult to manage and understand.

Requirement: **APPROPRIATELEVEL**

Extension developers should be shielded from the full complexity of the underlying application, and only exposed to the relevant parts of the base architecture, at the appropriate abstraction level, for the alterations they wish to make.

At first glance some of these requirements seem to be in direct conflict, and it will be shown that this tension causes issues for existing extensibility approaches. As previously pointed out, the ability to make fine-grained alterations (ALTER) apparently conflicts with the aim of keeping the application architecture manageable (APPROPRIATELEVEL). The ability to make substantial alterations to the base seems to conflict with the requirement that no base source be modified (NOSOURCE). Further, NOIMPACT says that any changes to the base for an extension must not be visible unless desired to the base application or other extension developers. Shielding those not interested from these changes implies the creation of many base application variants, leading seemingly to an inability to upgrade (UPGRADE).

1.4   Research Contributions

This thesis describes (and formally specifies) a design-time technique for manipulating the hierarchical structure of a base application architecture using structural deltas. This allows an extension developer to adjust any part of a base application architecture, permitting a system to be restructured even for requirements that were not considered when the base was initially developed.

This architectural manipulation is economically enabled by adding three concepts to a conventional architecture description language. By keeping the structural deltas of an extension separate from the base, the concepts resolve or ameliorate the unwanted constraints between developers in an extension setting, and hence address the requirements outlined earlier in this chapter.

A secondary contribution is the development of a modelling tool and runtime platform, based on the formal specification, to support the approach. The modelling tool integrates our structural delta approach with conventional UML2 composite structure diagrams. Component structures are shown as fully expanded at all times, even whilst deltas are recorded in the underlying architecture, providing a uniform approach to both construction and extension.

1.5 Thesis Structure

The requirements are used to define and justify the Backbone extensibility model presented in this thesis. The essence of the approach is to allow extensions to make alterations to the compositional structure of the base application. This allows any part of the system to be remade to accommodate an extension, and provides a structural foundation for checking and correcting conflicts when extensions are combined. The following chapters discuss how and why Backbone is suitable for creating, extending and upgrading applications.

The remainder of this thesis is structured as follows. Chapter 2 presents existing work in the area of extensible applications and architectural variants and examines how these approaches match up against the requirements.

Chapter 3 presents the Backbone component model and architecture description language (ADL). By working through an expanded form of the example scenario, we show how Backbone deals with the structural aspects of extension and conflict resolution. Backbone is assessed against the requirements in the context of the example.

Chapter 4 presents an outline of the Backbone formal specification. This is written in Alloy [Jac02, Jac09], a logic language well suited to model finding. Some of the more interesting properties to emerge are discussed, particularly in relation to tool support for the approach.

Chapter 5 outlines the Backbone modelling tool and runtime platform. The relationship between these and the formal specification is explained, showing how architectural alterations are handled in an intuitive way. The UML2 [Obj09e] composite structure diagram form is used as the graphical modelling notation, and a critique of the UML2 model is provided in the context of extensible systems. A mapping between Backbone and UML2 is provided.

Chapter 6 builds on the foundation of the the extensibility approach to demonstrate several advanced modelling techniques. Stereotypes are introduced as a way of expanding the structure of elements automatically, providing a convenience that allows us to omit common connectors and parts when defining a component. Several component-based variants of object-oriented design patterns are presented, focusing particularly on improved extensibility.

Chapter 7 evaluates Backbone against prominent approaches to building extensible systems and application variants. In particular, plugin architectures [Vol99, Bir05] and product lines [BLHM02] are examined in depth. As a case study, we introduce the Backbone approach into a mature application that has spawned a number of variants, demonstrating that a system does not have to be modelled from the ground up in Backbone to obtain the benefits of extensibility.

By way of conclusion, chapter 8 summarises the contributions of this thesis and suggests further work that builds on the structural foundations of our approach.

Appendix A discusses some areas of the Backbone formal specification in more detail and catalogues the structural and well-formedness rules. The specification itself is divided into two parts: a base that describes the extensibility concepts (appendix B) and a component model built on this (appendix C). The specification is written using Alloy, and this is briefly introduced in appendix D for readers unfamiliar with this logic.

Finally, appendix E describes how to obtain the software and models discussed in this thesis, including the Evolve modelling tool and the Backbone runtime platform.

Chapter 2

Background

This chapter presents a survey of existing work that is relevant to the creation of extensible applications, including application variant and large scale software reuse techniques.

We start by reviewing existing approaches to extensibility in section 2.1, indicating how these fare against the requirements of the previous chapter. When an implicit reference is made to a requirement, its name will be put in brackets after the text. E.g. (ALTER).

Section 2.2 and beyond focus primarily on a component-based and architectural perspective, along with the issues that prevent components from being reused and their architectures from being extended. The link between components and this thesis is that our approach relies on replaceable components to permit principled alteration of a base application.

Detecting unwanted interference, when combining extensions, has a structural and also a behavioural perspective. We focus primarily on the structural side in this literature review and thesis. Explicit discussion of behavioural concerns is deferred to section 8.3 where we consider further work, which builds on the structural foundation we establish.

2.1   Approaches to Extensibility and Large Scale Reuse

2.1.1   Scripting

In an influential article by Ousterhout [Ous98], scripting is presented as a way to integrate components implemented in conventional compiled languages. The argument is that scripting languages are more flexible and faster to develop in than their statically typed and compiled counterparts. This approach can be used to produce systems which are extensible via customisation of scripts. Scripting has also been used as glue code to ease the integration of components [SN99].

A number of prominent extensible applications have followed this architectural style, whereby blackbox implementation components are glued together with scripts. Firefox follows this model and uses

extensions which can consist of JavaScript code and optional implementation-level components or plugins[1]. These scripts are used to customise the user interface and can call into the underlying application or added implementation plugins.

Other systems have opted for a pure scripting approach in order to provide an extensible base. An important and complex example is Emacs, a feature-rich text editor which uses a Lisp-like scripting language for writing extensions [Bla09]. Much of the Emacs system is also developed in the same scripting language.

Scripting approaches have the limitation of requiring pre-planned extension points to facilitate extensibility. Emacs calls these "hooks", and scripts can register with these to be called at certain points in the application processing logic. If these hooks are not sufficient, then it is possible to copy and edit the script source code for the underlying base application, as much of this is often also expressed as scripts. However, this violates the NoSource requirement. Emacs allows modified scripts to live in a parallel installation-specific directory to avoid overwriting the original script sources, but although this allows freedom to make changes (Alter), it can also cause issues with upgrading and conflicts can ensue if multiple scripts make changes to the same part of the base (Combine, DetectAndCorrect, Upgrade).

Scripts from different extensions can conflict, and the interference can be subtle and difficult to detect. Conventions and ad-hoc rules for scripts have been proposed to ameliorate conflicts in extensible systems such as Firefox, however it is commonly the case that extension conflicts are detected first by users and cannot be foreseen in advance by extension creators [Bar09].

Excel [Mic09b] can be extended by a mix of Visual Basic scripts and compiled implementation COM components [BBG05]. This approach has the same limitations as other scripting systems. Further, if a change is required to a COM component, then it must be replaced or evolved, leading to a conflict if different scripts require different versions (Combine, NoSource). These versioning issues are discussed in more detail in section 2.3.3.

2.1.2 Object-Oriented Frameworks

Frameworks are a large scale reuse technique allowing a semi-complete application to be customised and extended into a complete application [FS97].

A framework allows objects implementing specified interfaces to be registered at pre-planned extension points, and it calls out to these objects at various points in its processing logic. This is known as inversion of control [Joh97b, Joh97a], where program flow is handed over to the framework. Framework extension points are also referred to as "hooks" [FHLS97]. The underlying mechanism of a framework has been codified as the open-closed principle [Mey97, Mar00]: frameworks are open to extension via inheritance, but closed to modification of the source code of the framework.

---

[1]The use of plugins in Firefox is covered in section 2.1.3.

Frameworks are able to be extended without having the implementation source code, as long as the extension has been accommodated by the provision of the appropriate extension points. Many complex graphical interaction frameworks are distributed in this way [VL89, HWL$^+$02]. As long as the specification of the interfaces is available, the framework can be delivered in binary object form.

Frameworks offer an effective and large-scale form of reuse, but in practice a number of fundamental issues limit the approach [CDHSV97]. Many of these issues stem from ownership of changes to the base framework. Alterations must be performed by the framework (base) developers, or else the framework's source code must be copied and independently modified leading to subsequent maintenance problems (NoSource, NoImpact). In many cases, timescales for application delivery will bias towards the latter solution. The net effect is that version proliferation and architectural drift of the base framework tends to occur [CDHSV97], making ongoing maintenance of the framework extremely difficult.

Evolving a framework as a single, unified version is difficult because the base framework must be aware of the different uses that it is being put to before it can be reliably upgraded. Various techniques have been proposed to mitigate this situation [CFL03, HH01, MB97], however these rely on placing constraints or conditions on the framework and preventing it evolving in a direction which breaks existing usage by clients [HH01, CFL03]. This limits the type of change that can be introduced for reuse (Alter) in order to minimise impact on existing clients (Upgrade, NoImpact).

In the process of creating a complex application, it is often necessary to combine and integrate two or more independently developed frameworks (Combine). This is problematic because each framework assumes under the inversion of control paradigm that it will have complete control of the "event loop". As pointed out succinctly in [MB97], frameworks are designed for extension and not composition, and source code access and modification is often required to integrate the control loops of different frameworks (NoSource). Further, the frameworks may overlap, requiring some functionality to be removed from one (or perhaps both) in order to ensure deep integration between them.

Some guidelines are issued in [vGB01] to allow frameworks to be more easily composed and extended. One of the suggestions is to use small components at a lower level of abstraction. This recommendation makes the assumption that larger components are necessarily monolithic, and violates the AppropriateLevel requirement.

The above issues show that there are many problems involved in reusing, extending and combining frameworks, particularly as they require that extension points be pre-planned and built into the original design. Further extension may require modifications to the source code to the framework, and this will either place an undue burden on the framework developers or result in a variant management problem due to the many different framework versions.

2.1.3 Plugin Architectures

Plugins are pre-packaged units of software that can be "plugged into" a base application to extend its functionality. This architectural style provides a lightweight way to reduce the size of a base system and simultaneously cater for a wider spectrum of requirements via extensibility [Vol99].

In its simplest form, a plugin can take the form of a compiled class conforming to an interface. The application discovers all plugins at startup time, and calls out to them via existing extension points at appropriate stages in its processing logic [MMS03]. The need to pre-plan and design extension points into the base application leads to limitations on the types of extensions that can be accommodated (ALTER).

Advanced plugin systems allow plugins themselves to also offer extension points [CEM03, Obj09f]. This turns the concept of a base application into a relative one, meaning that the original application and any collection of extensible plugins could be considered a new base to build on. Taken to its logical extreme, the base for a pure plugin architecture is simply a plugin discovery and load mechanism, which is how the Eclipse system is structured [Obj09g].

A recurring problem with plugin architectures is that independently developed plugins can interact in unforeseen ways when combined into the same base [Bea09, Des09]. This problem is exacerbated because testing a group of plugins for interference involves a combinatorial explosion of manual checks [Bea09].

By providing a versioning scheme for plugins, Eclipse allows parts of an application to be replaced to accommodate a new feature. This resolves some of the ALTER limitations regarding pre-planned extension points. It also leads to a greater likelihood of conflict when plugins are combined. In essence, plugin architectures reveal a tension between being allowed to make changes to the base (ALTER) and preventing structural and behavioural conflicts when plugins are combined (COMBINE). Granting the power to make extensive changes to the base increases the likelihood of conflict and exposes the weakness of the style regarding the lack of automated checks.

MagicBeans does provide a way to check if plugins are behaviourally compatible, by allowing each plugin to specify its actions via an finite state process (FSP) expression [MK06, CEM04], although it provides no way to repair any conflicts detected.

Plugin approaches are non-hierarchical: plugins cannot be composed of other plugins. This violates the APPROPRIATELEVEL requirement, where as the number of plugins becomes large it is not always possible to replace or modify base functionality at the correct level of abstraction. A tension becomes evident between making plugins fine-grained at a low level of abstraction to allow ease of replacement for extension, and making them coarse-grained at a high level of abstraction to allow the architecture to be managed and understood more easily. To indicate the scale of the problem, consider that a typical Eclipse environment contains over 200 plugins, and an enterprise development environment based around Eclipse is known to contain over 500 [GB03].

The availability of an application's source code does not reduce the need for, or the effectiveness of, a plugin architecture or other extensibility approach. Many prominent and successful open-source applications use plugins for extension, including Firefox [Moz09b], Eclipse and Wordpress [Wor09]. Although the source code for each of these applications can be freely modified and redistributed, the effort involved in copying and maintaining a variant of these applications would be large and dispro-portionate for all but the largest extensions. As such (and somewhat surprisingly), the extensibility

requirements presented in section 1.3 are not fully addressed by complete source code availability. Instead, the requirements are more a product of the constraints and interactions present in common extension scenarios rather than an inevitable outcome of the lack of source code access.

The Backbone approach is compared and contrasted with a plugin architectural style in section 7.1.

2.1.4   The Common Lisp Metaobject Protocol

We examine the Metaobject Protocol (MOP) not as a way of creating an extensible system, but as a useful exercise in how an important extensible system has been constructed.

The goal of the MOP is to allow the Common-Lisp Object System (CLOS) and language to be extended, without compromising performance and other features [KdRB91, KAJ$^+$93]. This is achieved through a set of interfaces (protocols) which allow the runtime system to be accessed and augmented. Augmentation is achieved through a set of hooks that allow user-level programs to participate in the runtime of a program.

This approach is related to the way that extensible systems use scripting [Ous98, Bla09], but provides a deeper and more powerful way of expressing system changes due to the fact that the full runtime and extensions are in the same language. The line between the runtime and user-level is blurred.

Kiczales and Lamping further examine the underlying issues facing object-oriented systems in [KL92] and partially conclude that a facility for the replacement of units in a system is necessary for extensibility. There is again the predictable tension between having coarse-grained replaceable units for making large changes whilst also allowing the system to be managed easily, and making the units smaller for ease of customisation. Layered protocols are proposed as a way of addressing this tension. In effect, this is a hierarchy of facilities where the user can change behaviour at the correct level of abstraction (APPROPRIATELEVEL). However, they note that arbitrary replacement will almost certainly lead to chaos, and seek to constrain what can be replaced (ALTER). Unconstrained alteration of the system is not permitted – instead system addition is primarily covered.

2.1.5   Aspect-Oriented Programming and Separation of Concerns

Aspects offer a way to weave cross-cutting concerns into an application, allowing it to be extended without modifying the source code.

Following from metaobject protocols, it was observed that the features added via such protocols cut across the base level computation of a program [KLM$^+$97]. Aspect-oriented programming (AOP) is a language, built on top of the facilities of a MOP, to specify cross-cutting features to a base program. These features are specified separately from the base and then woven together into the application at a later stage. The underlying intuition is that cross-cutting features are usually spread throughout a conventional codebase, and this creates maintenance, reuse and clarity issues: aspects allow these features to be untangled and represented in a single place.

Aspects have been proposed as a mechanism to improve the extensibility and reusability of frameworks and product lines [KAG$^+$06, FCS$^+$08]. Aspects are also sometimes used to patch and adapt code that cannot be modified [Rob07], partly addressing the ALTER requirement.

Before assessing AOP fully against our extensibility requirements, consider how AspectJ [Asp03], an AOP approach for Java, works. AspectJ uses the "join point" model. A join point identifies the type of language construct that can have additional functionality woven into it, such as a method or field modification. A pointcut then selects instances of a join point according to a lexical pattern match. e.g. `call(void Point.set*(int))`. Finally, an advice adds functionality to a pointcut and can be invoked before or after the call, or be sandwiched around the call.

AspectJ relies on lexical conventions such as prefixing all methods which cause mutation to an object with "set*". This is a fragile approach which could easily select more or less methods than desired through failure to follow the convention or when additional code is added (COMBINE). The base code may not have the convention required to support a new feature, which limits unplanned extension. Further, despite a reasonable set of join point possibilities, only certain alterations can be performed (ALTER). Subsequent upgrading of the base is also problematic as the lexical structure of selected join points may alter in a new version (UPGRADE).

Combining independently developed aspects into a single base is also troublesome. Although AspectJ offers a default combination order for advices based on a set of rules, and also supplies a precedence operator, the order can inadvertently change by adding extra advices to the application. Conflict detection is difficult and subsequent repair involves modifying the advices and their precedence (COMBINE, DETECTANDCORRECT). Further, aspects are not composable and cannot directly refer to each other [OT01].

In an empirical study looking at using aspects to evolve a software product line, it was found that the source code of the base had to be changed in certain situations to allow aspects to be introduced, that the lexical specification of aspects was fragile, and that the use of aspects harmed the modularity of features in the base [FCS$^+$08].

Multi-dimensional separation of concerns (MDSOC) is a generalised form of AOP where there is no principal axis. This provides an extension developer with the same power as a base developer, unlike in the AspectJ model. This approach has been used to create, evolve and extend systems [OT01]. Hyper/J [TOS02] uses hyperslices where each slice contains the full code for a particular cross-cutting concern, in such a way that it can exist as a program in its own right. Hyperslices can be composed by specifying the order of composition and instructions on how each slice maps onto a unified class hierarchy. As one hyperslice may selectively replace methods in another slice, order is important and no guarantee can be made when combining slices that the result will be correct (COMBINE, DETECTANDCORRECT). This ordering may require modification as other slices are composed in. Further, modification of methods is necessary when the concern does not cover the entire body of the method [LMW00]. Overlapping features between slices can lead to complex interactions which cannot be understood in isolation, leading to inconsistencies and undesired interference when slices are combined [Nel03].

2.1.6   Product-Line Architectures

A product-line architecture focuses on a set of reusable components that can be shared by many systems in a product family [EBB06]. The development of components is driven by a hierarchical feature model which specifies fine-grained variants which deal with different use cases. Products are created by adding to the feature graph if required, and then choosing the appropriate set of features to assemble into the new application.

Concurrent evolution of single components in a product line is problematic [SB99], and work to merge and propagate changes from one branch to another is still required if the code for a component is branched [CCG$^+$04] (COMBINE). In addition, this approach usually involves maintaining a single feature graph for all product variants, which makes decentralised development difficult (NOIMPACT).

GenVoca is an approach and environment for generating product lines [BLHM02]. The primitive element of composition is called a *gluon* and these are arranged in layer-like structures called *constants*. GenVoca allows class extension, which is a where a subclass assumes the name of its parent class and this allows existing functionality to be extended via inheritance and replacement. This approach shares many of the features and limitations of the MixJuice approach outlined in section 2.1.7.

A related product line approach is AHEAD [BLS03]. This is evaluated in more detail in section 7.2 where it is compared and contrasted with Backbone.

2.1.7   Difference-Based Modules

MixJuice adds a module system to Java, where modules describe the difference between the base application and the desired application [IT02]. The intention is to allow a Java program to be extended in unplanned ways.

MixJuice relies on a total module loading order to be specified, although it can use implicitly included "complementary" modules to resolve several common types of conflict. Inheritance features prominently, and any alterations and corrections for conflict are constrained by the limits of this construct. In particular field deletion or replacement, and method signature change and removal cannot occur (ALTER).

Because it relies on the Java class model, MixJuice also does not offer a true architectural approach with separate levels of abstraction (APPROPRIATELEVEL). MixJuice uses a naming system of *module::class* to prevent accidental name collisions in independently developed modules.

A graphical approach to depicting MixJuice architectures has been proposed [Ich02]. The issues and limitations involved in using MixJuice to refactor an existing system are described in [Che06].

2.1.8   Viewpoint-Oriented Approaches

Viewpoints allow multiple perspectives on a single system to be constructed separately and combined at a later point [FKN$^+$92]. This enables independent development of different aspects of an application, supporting the different roles and expertise within a team. A viewpoint is not restricted to the structural side of a system, and can encompass requirements, architecture and also functional and non-functional concerns.

Viewpoints encode partial knowledge about a system and are based around a representational schema of a particular domain. Consistency checks are used to ensure that the combined viewpoints represent a coherent system. Viewpoints represent a compelling approach to the construction of a system with multiple facets, but are not explicitly oriented towards the creation of extensible applications. In particular, a constructive approach is taken, which prevents the modification of existing features (ALTER).

A more structurally-oriented approach is taken in [EHTE97]. In this model, views also encode partial knowledge about a system, based on a reference model. In practice, these structural views are object instance graphs which are combined at a later point into a unified model.

2.1.9   Parametrisation and Product Diversity

Koala [vO00] is a component model for embedded systems based on Darwin [MDEK95] that allows for extension of an architecture through variation points and parametrisation. Component variants can be plugged into the variation points, supporting a family of applications. The points must be decided in advance, limiting this to a technique for planned extension. This does not fully satisfy the ALTER requirement.

Parametrisation is used in Koala to capture options supported by a component [vO00]. This approach can result in a combinatorial explosion of options if the parameters of the constituent parts of a composite component are also exposed. Further, any changes need to be factored into a new version of the base application, violating the NOIMPACT requirement and reducing the clarity of the architecture.

Koala features HORCOM, a software bus which mirrors the functionality of an electronic hardware bus [vO03]. Components are decoupled from each other by the bus, and understand a standard protocol. Additional components can be plugged into the bus, extending the behaviour of the system in an additive way only (ALTER). Feature interaction is not handled explicitly (COMBINE).

2.1.10   Mixins

Mixins are a language feature for expressing abstract subclasses that can be reused in different parent classes [BC90]. Any number of mixins can be combined into a parent class, and methods of the mixin may invoke methods of that class. This implies that the mixin must make assumptions about the names

of the parent methods that it will call, which presents an integration issue. Further, multiple mixins may conflict or interact in unforeseen ways when combined into the a single parent class (COMBINE, DETECTANDCORRECT).

Scala is a language with mixins which aims to support the combination of independently developed extensions [ZO04]. It provides two dimensions of extension: data extension for the object-oriented view, and behavioural extension for the functional view. Scala does not resolve or ameliorate the name collision problem suffered by mixin and multiple inheritance approaches.

Units and mixins are used in [FF98] to separate component creation and dependency binding, providing a significant level of reuse for large units or modules. However, once a binding has been made, it cannot be replaced or removed, which limits this approach for extensible systems. The result is a variant of the abstraction problem [GSC⁺04] where buried abstractions cannot be altered (section 2.2.2).

2.1.11   Virtual Classes and Nested Inheritance

The BETA language allows virtual patterns, which are a type of virtual class [LMMPN93]. A pattern can specify other nested patterns, and sub-patterns can refine nested types through inheritance, allowing large patterns to be reused and extended.

Nested inheritance provides a similar facility, which allows a class to override any inherited nested types [NCM04]. This extends the BETA facilities by guaranteeing type safety through static analysis. The example scenario of an extensible parsing system demonstrates considerable reuse for a compiler family.

In practice, both approaches are limited by the need for pre-planned extension points (where appropriate classes have been explicitly nested or marked as virtual) and are subject to the limitations of the inheritance construct (only additions and compatible overrides).

2.1.12   Feature Composition

Feature composition has been proposed to allow independently developed features to be added to a telecommunications system, where existing services must not be affected in an adverse way by any newly added functionality [HA00]. This approach uses a series of relational assertions to model states and events, along with invariants which characterise the intended effects of the added features. It is possible for a newly added feature to adjust another feature already present in the system.

Feature interaction is detected at runtime by examining whether one feature has invalidated the invariants of another. A lower priority feature is not allowed to violate the invariants of a higher priority feature.

This work does not explicitly model an architecture, and limits the types of application extension allowed to ensure that unwanted feature interaction can either be prevented or contained (ALTER).

2.2   Component-Based Software Engineering (CBSE)

A longstanding goal of software engineering has been the ability to efficiently and reliably construct software systems from prefabricated components. An early reference to the general concept and vision was outlined by M. D. McIlroy at the 1968 NATO Software Engineering Conference [McI68]. In this influential white paper, software production techniques are compared unfavourably to industrial manufacturing techniques in electronics and hardware. A key element is found to be the idea of a component, or interchangeable part, which provides a level of modularity. The lack of support (at that time) for a component industry where component producers provide catalogues of parametrised components was taken as a sign of the lack of maturity of software production techniques relative to other fields.

Following from this, a vision of the production of systems through the customisation, "transliteration" and assembly of parametrised software components is proposed. The proposal outlines options such as space-versus-time trade-offs for algorithms, and considers the need to automatically translate algorithms into different languages for different operating systems.

Apart from the generation and translation perspective, this vision of components is similar in many ways to the modern concept of a class library. The examples chosen are based very much around algorithmic concerns, and parametrisation and generation are proposed as ways to handle the level of choice required. The wider concerns of a component-based architecture, such as dynamic structures and multiple levels of architectural abstraction, are not discussed although dynamic memory allocation is briefly mentioned.

The modern definition of a software component varies depending on the context, with some literature emphasising the notion of third party deployment and composition [Szy02], whereas other models focus on inter-language communication and interoperability [Obj09a, Obj09b]. The minimal consensus that we adopt for this thesis, however, is far simpler:

> *A component is a unit of software that can be instantiated, and explicitly declares the interfaces (or services) it provides and requires.*

This definition is at the heart of most component approaches [MDEK95, CH01]. It does not mention implementation language, architectural hierarchy, distributed systems or the how components are assembled to form a system. These aspects are important to many component models, and can be layered on top of this definition. Of particular interest, is that a software component can be represented by an object-oriented class as long as some convention or mechanism is used to explicitly denote provided and required interfaces, and constrain communication between components to those interfaces.

2.2.1   Issues Preventing Component Reuse, Extension and Integration

In [CN91], CBSE is compared to the production of muskets in the pre-industrial era, where hand crafting of parts was routine. The lack of both interchangeability and standard methods of measuring component specification compliance are cited as being key impediments to the development of component catalogues. The aim of these catalogues is to facilitate large scale reuse, and allow rapid construction of applications via component composition.

However, a more fundamental problem preventing reuse and integration is an inability to specify component alterations to accommodate for new requirements (ALTER). As pointed out in [Hol93], even minor syntactic mismatches prevent component integration. The full spectrum of issues ranges from simple naming issues through to more complex behavioural interactions and testability concerns [KL92, Szy06, Stu05]. Component versioning and deployment approaches have been proposed as a way of mitigating some of these issues [MS02], where multiple versions of a component may be deployed into in a running system. This does not address the issue of migration to the newer component version, or the problem when a single version of the component must always be enforced. The latter situation occurs when a component is managing a resource that requires a single controlling entity in a system, or when the component is managing shared state.

A common theme of several approaches is to solve the integration issue by wrapping the original component and delegating selectively to it. Wrapping is proposed in [Hol93] in order to adapt an interface for naming mismatches, although it is pointed out that this introduces a performance problem. It also introduces a problem with identity as both the wrapped object and the original need to assume the same identity in some situations. The desire to wrap components is more focused on solving the problem at the implementation level, where maintaining compatibility with existing languages and paradigms is either implicitly or explicitly considered to be of paramount importance. Superimposition [Bos99] is a variation on this theme which aims to address the identity problem (referred to also as the "self problem"). The aims of adaptation, in this case, are that the changes can be applied transparently, and that the wrappers can be reused in other contexts. This does not solve the fundamental problem that wrapping is a blackbox reuse technique [BW99, Szy02] which cannot adjust fundamental characteristics of a component, only hide them. Other wrapping approaches are outlined in [Jor04, TVJ$^+$01].

Another approach to adaptation is to use glue code written in a scripting language to handle any component integration issues [SN99]. This is a blackbox technique also, and suffers from the limitations outlined above.

In a far reaching article, Szyperski lists some of the challenges of extensibility in a component context [Szy06]. Extensible systems are not open to total program analysis due to their open nature and therefore all variants cannot be exhaustively tested (COMBINE). The (possibly mutual) interdependence between extension providers and the interaction between independently developed extensions must also be considered (DETECTANDCORRECT). Another problem noted is the tension between fine-grained components for ease of extension, and coarse-grained components for ease of architectural management (APPROPRIATELEVEL). A system that can be extended with independently developed extensions and

remain verifiably correct is known as *independently extensible*. Interestingly, Szyperski proposes that modules be used as the unit of extension in a component system. Feature removal is not discussed.

2.2.2   The Abstraction Problem

In a hierarchical model, composite components can be assembled out of instances of other components, forming a composition hierarchy. Reuse of components in this type of model is progressively more complex if a composite requires a change to a component deep in its hierarchy, as shown in figure 2.1. It is difficult to replace the single component requiring change (X in composite A) as it is deeply buried. Replacing X globally is not necessarily a solution, as it is also being used elsewhere and it may be desired that B be unaffected by the change.

The strength of hierarchical models is that they can deal with multiple abstraction levels, satisfying requirement APPROPRIATELEVEL. However, because abstractions can be deeply buried in a compositional hierarchy, alterations may be difficult (ALTER). This is a variant of the abstraction problem [GSC+04]: components are more valuable when they represent higher-level abstractions targeted at a particular domain but this specificity and the hierarchy implied limits their potential for reuse.
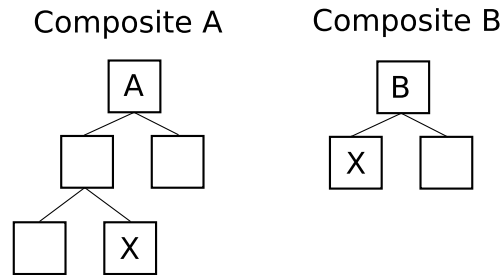
Figure 2.1: Two composition hierarchies, showing a buried abstraction

Using hierarchy, an entire system can be modelled as a single composite component [GMW97, KMS89]. As such, component reuse and extensibility are closely related to system reuse and extensibility, and problems at component level often translate to analogous problems at system level.

2.3   Component Technologies

A number of component runtime, packaging and deployment technologies exist. These are widely used in industry, and in some cases have led to significant commercial marketplaces for components adhering to their respective models. The following technologies are reviewed along with their support for extensibility based on the requirements in section 1.3.

2.3.1   CORBA and the CORBA Component Model (CCM)

CORBA [Obj09a] provides a platform-independent communications format and distributed component model. Interfaces are specified using an interface definition language (IDL), which can then be compiled

for a specific language and platform choice into stubs and skeletons. Component references can be passed at runtime as object references or turned into a textual form as inter-operable references (IORs). Bindings between components are formed via service location [Fow09] and are not specified using connectors. A hierarchical model is not supported.

The CCM is a component model for CORBA [Obj09b]. Components may provide and require interfaces (called facets and receptacles respectively) through ports, and an event model for asynchronous communication is supported (sources and sinks). The model is non-hierarchical, and does not feature connectors in the standard specification [RRS+05]. There is a considerable amount of similarity between the CCM and the Enterprise JavaBeans specification (section 2.3.2).

Neither CORBA nor CCM are particularly suitable for building extensible systems. Alterations to existing components are only possibly via source code modification or component wrapping (NoSource, Alter). Further, as a non-hierarchical approach, extensions cannot modify or replace components at varying levels of abstraction (AppropriateLevel).

2.3.2 Java Component Models

JavaBeans [SUN09b, O'N98] is a minimalist client-side technology for the creation and configuration of reusable Java components. Fundamentally, a bean is just a class that conforms to a lexical convention for getting and setting attribute[2] values. This approach is primarily focused on graphical components, and supports an event model where clients are notified of changes to bean attributes. Connectors are not explicit, but are instead modelled as references from a bean to clients interested in changes in its state. JavaBeans supports a limited form of hierarchical composition through object aggregation. It does not support explicitly required interfaces.

Enterprise JavaBeans (EJB) is the distributed server-side Java component model [SUN09a], and despite the naming similarity does not build on the JavaBeans specification. Component hierarchy is not supported in any version of EJB. In EJB version 2.1 and below, there was no way for one component instance to be declaratively connected to another. Connections are instead implicitly formed by each component using a registry called JNDI (Java Naming and Directory Interface) to locate required services, which effectively forces the embedding of connector information inside component instances. This severely limits the extensibility of EJB models, as vital connection information is embedded in the code making it difficult to change without altering the implementation (Alter, NoSource). The embedded service location code also makes testing of EJB systems problematic, as it is difficult to replace the components in the environment of the component being tested.

As an alternative to the EJB model, so-called "lightweight frameworks" have become popular for implementing enterprise systems. The Spring framework [JHA+05] uses XML configuration files to connect JavaBeans together declaratively. Provided interfaces are denoted by the class type of the bean and any implemented interfaces. Required interfaces are specified by stretching the notion of a bean attribute to refer to required bean instances. Used in this way, JavaBeans provides a functioning

---

[2]An attribute is also known as a bean property.

component model and this style has become known as dependency injection, in contrast to service location where each component uses a registry or directory to locate services [Fow09]. Hierarchical composition is not explicitly supported in Spring and connectors are implicitly defined through bean references. Despite these limitations, the Spring XML configuration facility can be used in an ADL-like fashion [McV09].

The binding of components using service location is known as first-party binding because the components themselves play an active part in resolving the services required (section 2.4.1). Dependency injection is an example of third-party binding, so named because an external third party creates the connections. Much of the evolution of EJB and enterprise Java frameworks can explained in terms of the progression between these different styles of binding.

A Spring child bean definition can inherit from a single parent bean definition, and can selectively replace some parts of the configuration giving a level of configuration reuse analogous to class-based inheritance. This allows a new Spring component to be defined in terms of an existing one, where existing references can be overridden and new references added. Each bean is associated with one implementation class, so any configuration evolution must generally be matched at the implementation level via source code evolution (ALTER).

Spring can also use aliases, which function as replaceable names. Aliases are part of a flat, global namespace. Through careful organisation of included configuration files, it is possible to use this to provide extension points. This, along with overriding, offers a form of pre-planned extension similar to the use of variation points in Koala (section 2.1.9), and suffers from the same extensibility limitations.

Spring provides aspect-oriented programming facilities, which can be used to customise the behaviour of a system without directly modifying the structural configuration or implementation source code. This suffers from the same issues as other aspect approaches (section 2.1.5). Extension in the Spring system is not uniform, sometimes involving adjusting the structural configuration and sometimes involving the development of aspects.

Spring does not provide support for analysing or resolving conflicts caused through combining independently developed extensions (COMBINE, DETECTANDCORRECT).

EJB version 3.0 [DK09] has adopted many of the dependency injection ideas from lightweight frameworks, bringing JavaBeans and EJB closer together conceptually.

2.3.3   Component Object Model (COM)

COM is a component model and runtime infrastructure built into the Windows operating system [Box97]. It forms a powerful extensibility mechanism for many applications, including influential programs such as Excel [Mic09b]. COM supports a hierarchical model, and composition of instances is via a registry based approach for indirectly locating service providers. Components must explicitly declare provided and required interfaces.

COM and its successor ActiveX [Mic09a] have successfully produced a sizable commercial marketplace for Windows user interface components.

In COM, component versions are held in a global registry, which leads to a situation called "DLL Hell", when multiple applications require different versions of the same component [SE04, Stu05]. This approach restricts evolution to always guaranteeing backwards compatibility, although techniques such as providing both the original interface and also an upgraded interface in the same component allow some mitigation against this. Registration-Free COM [Tem09] improves this situation by allowing a single application to maintain and control its own local registry thereby avoiding conflict with other applications. However the problem of version conflict still occurs when combining multiple components into a single application as different extensions may require different versions of a subcomponent (COMBINE).

The COM model does not focus on supporting or resolving extension conflicts, and the obscure and indirect nature of the registry-based configuration makes models difficult to visualise and evolve architecturally (UPGRADE, COMBINE, DETECTANDCORRECT).

2.3.4   Open Services Gateway Initiative (OSGi) and the Eclipse Plugin Framework

OSGi defines a module-based environment for the deployment of network-enabled services [OSG09]. It is used as the basis of the Eclipse plugin architecture [Obj09g].

OSGi provides facilities to manage the deployment and life-cycle of bundles (analogous to modules) and services, and these may be updated or removed at runtime without restarting the system. Bundles may depend on and also provide interfaces, although OSGi has not provided any form of explicit connection facility until very recently [OSG09, EH07]. Generally, services are found via a locator pattern [Fow09].

The OSGi model does not offer support for bundle composition, and any architectural hierarchy must be inferred through the names of the bundles which conform to Java package naming conventions. This lack of hierarchy and the difficulty in managing a large "flat" architecture tends to bias the developers towards more coarse grained bundles, increasing the footprint of potential conflicts when these must be replaced or evolved (COMBINE, APPROPRIATELEVEL).

Eclipse uses an implementation of the OSGi standard for its plugin architecture, along with a registry based approach for matching up extension providers with extension points. In this approach, bundles are synonymous with plugins. Bundles therefore form the unit of versioning and grouping, and also the unit of replacement. However, classes rather than bundles form the unit of instantiation and are passed around as references. This pushes much of the architecture into service location code contained within classes, obscuring the clarity of the system. The registration approach has similar limitations to the COM registration model (COMBINE, DETECTANDCORRECT).

The lack of architectural hierarchy and the subsequent coarse-grained nature of components in Eclipse interacts badly with the plugin versioning system, meaning that some changes require inordinate effort

as well as source code modification. A full critique of the Eclipse plugin model is provided in section 7.1.

Extensibility platforms based around OSGi demonstrate that modules (information hiding, grouping, coarse-grained dependency management) and components (instantiation, explicit service dependencies) are both useful in their own right. The need for both and their interplay is clearly expressed in [BPV98].

2.4 Architecture Description Languages (ADLs)

ADLs evolved out of the desire to describe the architecture of a software system as a set of explicitly connected components. Many ADLs exist, and the general consensus is that they must support hierarchical composition of components, explicit connectors and an underlying formal model which can be analysed [MT00].

The use of connectors and declarative connector bindings sets ADLs apart from earlier module interconnection language approaches [AG94], as well as the component models in section 2.3.

In this section we review some of the influential ADLs and examine their support for building extensible systems. Tool support and graphical component representations are also reviewed.

2.4.1 Darwin

Darwin [MDEK95, KMND00] is an ADL for specifying the architecture of distributed systems. It focuses on the structural side of a system, but also offers support for multiple views and allows information to be added to the specification in order to aid analysis. The separation of the structural description from the implementation is referred to as a configuration-based approach. Koala [vO02] is a variant of Darwin used for describing the software architecture of embedded electronics in appliances such as television sets (see section 2.1.9).

Darwin components can be either primitive or composite. Primitive (or leaf) components have no further decomposition, and can be supplied with a behavioural specification as opposed to a structural description. Composite components are used to compose and connect instances of other components into a new structure. An architectural hierarchy follows from this. Figure 2.2 shows the Darwin graphical depiction of a leaf filter component along with the textual definition [MDEK95]. Figure 2.3 shows a composite pipeline component. The circles refer to input or output ports, and all interaction with an environment is conducted through these gateways.

Darwin supports four different mechanisms for creating connections between the provided and required services of components [CDF+95]. In *first-party binding*, the components themselves play an active role in establishing their own connections by utilising an external registry or name server to look up any required services. The drawback of this is that the information is now encoded implicitly inside the components themselves, constraining their reuse in different scenarios. In contrast, *third-party*

binding is where a external party creates the connections, allowing the connection information to reside in a configuration and be subject to architectural analysis. *Second-party binding* separates the configuration into two parts: the core architecture (which can specify virtual components at points in the architecture) and the final deployment target (which substitutes the virtual components for specific components available in the target environment). The final binding type is known as *dynamic invocation binding*, whereby a component reference is passed as a parameter between components. This type of binding is generally discouraged, as it can lead to complex, dynamic architectures which cannot be described by a static configuration.

Darwin supports dynamically instantiated components in one of two ways. Lazily instantiated components are created upon receipt of their first message. Secondly, components may be programmatically instantiated via a *dyn* port, which allows for multiple instantiation subject to some limitations [CDF+95].

The $\prod$-calculus [Mil99] is used as the underlying formalism to describe what a structural configuration means, and the formal specification uses channel names to represent Darwin port addresses [KME95]. This precisely describes how a configuration is flattened and instantiated in a distributed setting.


```
component filter{
  provide output<stream char>;
  require input<stream char>;
}
```
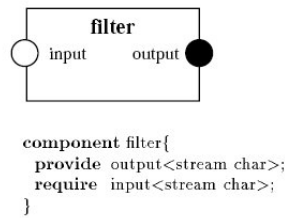
Figure 2.2: A Darwin primitive component


```
component pipeline(int n){
  provide output;
  require input;

  array F[n]: filter;
  forall k:0..n-1 {
    inst F[k] @ k+1;
    when k < n-1;
      bind F[k+1].input -- F[k].output;
  }
  bind
    F[0].input -- input;
    output -- F[n-1].output;
}
```
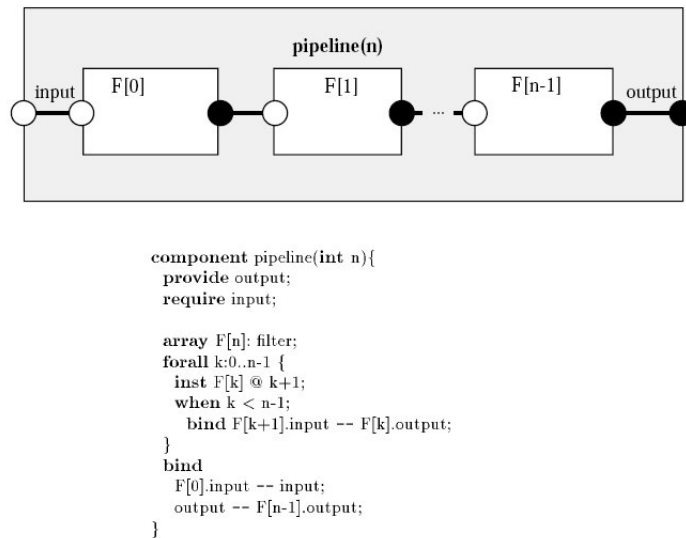
Figure 2.3: A Darwin composite component

The Software Architect's Assistant (SAA) is a graphical design environment for the development of

Darwin programs [NKMD95, NK96]. It understands the hierarchical structure and distributed nature of Darwin systems, and provides integrated graphical and textual views. A composition hierarchy view is supported as well as component diagrams. Libraries can be created, facilitating a compositional approach to system building which involves constructing new components from a set of existing ones. A key focus is on supporting the distributed nature of Darwin programs, and components can be mapped onto machine nodes in a topology.

Behavioural information in Darwin is specified using finite state processes (FSP), a process calculus [MK06]. The Labelled Transition System Analyser (LTSA) tool can process FSP into labelled transition systems, allowing deadlock checking, the checking of safety and progress properties, and other types of analysis to be performed.

Darwin supports a form of port type inference, where the interfaces of ports of composite components do not have to be explicitly specified, but can instead be determined from the internal connections. Type parametrisation is also featured [DSE97].

Dynamic reconfiguration of architectures specified in Darwin is explored in [KM90, MK96]. This work considers how to accommodate evolutionary change to the structure of an architecture, whilst the system is running. Configuration changes are modelled as deltas: component and link creation and removal are supported. The property of system quiescence, indicating that a given component is not currently involved in a transaction, is used to determine when to transfer application state from the old components to the new ones.

Darwin can be used to model extensible systems through the use of type parametrisation. This is a form of pre-planned extension that relies on the correct types being parametrised in order to accommodate extension needs, and limits the alterations and types of extensions that are possible (ALTER). Although the work in [MK96] develops the intuition that all change in a system can be accomplished through deltas affecting the components and connections between them, these change facilities were not explicitly added back into the Darwin design language.

The work in this thesis started with the insight that adding these architectural change facilities into a Darwin-like design language, in a way that respects hierarchical component specification, would allow system reuse and extensibility according to the requirements listed in section 1.3.

2.4.2   The C2 Family

C2 [TMA$^+$95] is an ADL designed to support the explicit requirements of graphical user interface (GUI) software, including the reuse of GUI elements. It provides support for the separation of graphical views and underlying data using the model-view-controller paradigm [Ree09, TMD10].

A component has a top and bottom domain. The top domain specifies the notifications that the component can accept and the requests that can be issued to the rest of the architecture. In essence, the domain concept models both the notions of provided interfaces for the handling of notifications, and also required interfaces for issuing requests.

The bottom domain indicates the notifications that will be emitted, essentially modelling the required interfaces for sending events to the rest of the architecture. Each domain may be connected to only one connector, but connectors can accept links from many components. A key principle is one of substrate independence, where the component knows of the components that are connected to its top domain, but does not know which components are connected to the bottom domain. An architecture can be reused by slicing it horizontally at a certain level and taking the components and connections above this level as a reusable set. Each C2 component may be active, with its own thread.

C2 SADL [Med96] is a variant of C2 designed to express the dynamic instantiation of components. It supports upgrading components and the removal of unwanted components via reconfiguration in a running system. C2 SADL also defines the concept of *placeholder* components [MORT96], for representing conceptual entities which have not been fully elaborated. This provides support for top-down design in addition to bottom-up construction of an architecture.

Later work in this area developed C2 SADEL [MRT99] which explicitly provides support for expressing and analysing the architectural evolution of a system at specification time. The protocol of a component is modelled via state, invariants, and pre- and post-conditions. Further, this approach is used to implement the design environment and language, demonstrating its applicability.

Much of the work on C2 implies a non-hierarchical[3] architecture, although C2 SADEL specifically mentions that composite components are possible by eliding the internal structure of a complex configuration and exposing any required top and bottom domains [Med99]. C2 SADEL does not include facilities for expressing the evolution or extension of composite components (ALTER). As much of the structure of a complex system is encoded in these components, any architectural extensibility approach must provide support for reuse and alteration of composite components and configurations.

Work on C2 runtime software evolution describes operators which can remake an architecture [OMT98]. Architectural hierarchy is not mentioned or dealt with, limiting the scope of this work.

2.4.3 ROOM

ROOM [SGW94b] is an ADL for modelling and constructing real-time software systems. Components are called actors, and must specify any required or provided interfaces via ports. Each actor is concurrent, as per the original model specified in [Agh86]. Actors are hierarchical, and protocols are described using extended state machines (ESMs), which are a type of automata allowing variables.

The presence of variables makes it difficult to analyse protocols using model checking, as the state space can be prohibitively large. From an engineering perspective however, ESMs are more attractive to design with because the number of specified states is usually far smaller than in the equivalent finite automata. ROOM also includes a concept of structural inheritance that allows for component reconfiguration in a sub-actor. Features to handle conflict due to multiple actor inheritance and conflict

---

[3]The C2 literature sometimes uses the term "hierarchy" to refer to the connection of component instances in a flat, non-compositional setting. This thesis avoids this usage, and only uses the word in conjunction with a true compositional hierarchy.

resolution are not described in the ROOM literature. No component replacement facilities are offered, limiting the types of extension possible to pre-planned (ALTER).

ROOM outlines a pragmatic and wide-ranging vision of CBSE which includes a virtual machine for model debugging and execution, and tools to translate models into implementation languages. This vision led directly to the ObjecTime toolset which eventually was developed into the Rational Rose Realtime toolset, and later renamed as Rational Rose Technical Developer [Rat09]. This work has had a far reaching impact on graphical modelling techniques, and is one of the key influences that led to the vision of Model Driven Architecture (MDA) [Obj09c]. The ROOM approach and similar techniques have been successfully used in the real-time software arena for many years.

2.4.4 The Unified Modelling Language and the Stereotype Extensibility Mechanism

The Unified Modelling Language (UML) is a graphical language for describing the structure and behaviour of object-oriented systems [Obj09e]. UML was standardised by the Object Management Group (OMG), which subsequently evolved it from version 1.0 through to version 2.0.

UML2.0 (abbreviated to UML2) introduced a number of component-oriented diagram types which allow it to be feasibly used as the basis of an ADL [GA03]. The design of these diagram types and underlying model have been heavily influenced by Darwin, ROOM and several other ADLs. Composite structure diagrams can be used to model composite and leaf components [OL06]. Component diagrams are also provided, but these are essentially a syntactic variant of composite structure diagrams and are not considered further in this thesis.

Figure 2.4 shows the leaf filter component (cf. figure 2.2) and composite pipeline component (cf. figure 2.3) depicted using the UML2 composite structure notation. Of particular note is that UML2 does not include textual facilities for model representation or computational instantiation as per the Darwin example, and hence only a fixed-length pipeline can be shown. Ports are shown as square rather than round, and interfaces are explicitly shown as circles (provided) or half circles (required).
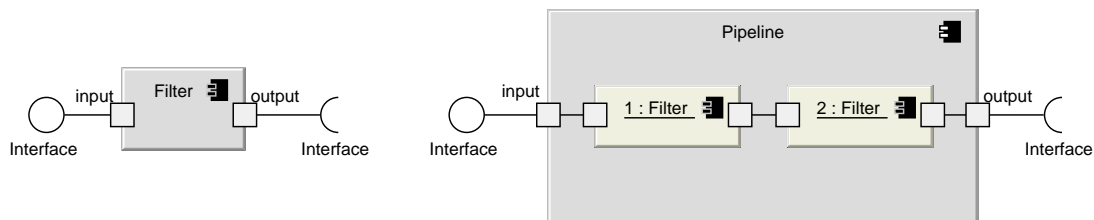
Figure 2.4: UML2 depiction of a leaf and composite component

Component protocols in UML2 are modelled either as sequence diagrams [Sel03] or activity diagrams. Essentially, this amounts to modelling protocols as ESMs, with the same engineering advantages and limitations on their formal analysis. UML2 sequence diagrams can feature looping and alternation as well as other operators.

UML2 introduced package merge which provides a way to specify extensions to a package via other packages. This was provided to allow the UML2 specification itself to be phrased as a set of layers where upper layers can add to and amend lower layers in a backwards compatible way. As pointed out in [ZA06], this construct can only add features to a model and contains a number of issues which prevent its use as a general approach to creating extensible applications.

The UML2 language can be extended through stereotypes. Consider that a "heavyweight" extension is made by editing the underlying metamodel (schema) of the UML2 language. To allow a "lightweight" form of this type of extension, stereotypes are provided which allow "virtual subtyping" of the meta-model [HSGP06]. Stereotypes are effectively a class, and can have named attributes. This powerful approach allows a user model to extend the metamodel for the purposes of a particular domain. A collection of stereotypes is known as a profile.

UML2 represents an amalgam of loosely integrated and overlapping techniques and graphical diagrams from many areas, showing its heritage as standard designed by a committee with the associated politics. The metamodel is complex, featuring over two hundred separate elements. Part of the challenge of applying the UML to a software system is to choose an appropriate subset of the language and give it a more precise meaning.

2.4.5 Extensible Architectural Interchange Languages

ACME is a (non-XML) ADL which can represent component and connector architectures [GMW97]. A key feature of ACME is that it provides a core which can capture the set of architectural structural concepts, whilst also permitting the inclusion of other information via annotations which refer to externally defined sub-languages. It provides an interchange format, which allows a common set of tools to be built which work with multiple architectural styles and representations.

In a similar vein, xADL (Extensible ADL) is an XML schema for representing architectural descriptions [DAH+07]. It uses the XML schema facility and its associated extension features to allow additional architectural information to be added for a particular context. An example is the adding of variant information for product line architectures to the core schema of xADL 2.0 [Das09].

The core xADL schema represents a generic architecture description language, biased towards the C2 style. A toolset has been built around this language, using data interchange in xADL to facilitate loose-coupling and tool extensibility [KGO+01]. ArchStudio is the modelling environment at the heart of this suite.

The extension facilities in xADL are comparable to the so-called heavyweight extension facilities of UML2 where the metamodel is directly edited in such a way as to remain compatible with the original language. xADL does not provide a facility analogous to the lightweight approach of UML2 stereotypes where a model may actually "virtually" extend the metamodel (section 2.4.4).

2.5   Architectural Evolution

Various trends in approaches to architectural evolution are discussed in [AZB06]. Topics include graph rewriting approaches, generation-based approaches, ways of assessing architectural stability in the presence of new requirements and the safe integration of new concerns through aspect-oriented techniques applied at the architectural level.

A box and line representation of architectural evolution and branches, based on an underlying formal model, is covered in [Erd98].

2.5.1   File-Based Configuration Management Approaches

Conventional file-based CM (configuration management) systems have been successfully used to manage and evolve component architectures [Stu05], and offer a lowest-common denominator approach to extending a system by modifying the source code.

Combining structurally conflicting extensions in such an approach has the disadvantage of requiring the developer to fully understand the source code and any changes made in order to perform a sensible source code merge. It also offers no guarantees that the properties of the each extension will be preserved in the combined system or that apparently independent extensions will not interfere after merging.

2.5.2   MAE

MAE addresses architectural evolution by integrating architectural concepts into a centralised CM system [vdHMRRM01, RVDHMRM04]. This permits the explicit evolution and merging of architectural configurations via version control, at the level of architectural concepts such as connectors and components.

This approach provides an overarching CM system which can support the creation of variants in branches using architectural deltas. For instance, to accommodate a new extension, a branch of the architecture could be produced which introduces the new variation points required. Like in a source code CM system, the changes will be held as deltas against the architectural configuration and multiple variants will eventually be merged to form a new baseline with no variants. Patches can be issued from these deltas to update a system to a new configuration.

The ability to introduce variants, however, does not fundamentally solve the NoIMPACT requirement: we are still over time introducing many variation points and baselining into a notionally unified base architecture. This leads eventually to a complex, overly generic architecture with many extension points to satisfy all extension developers.

The approach also assumes that all components are available via a unified and consistent CM system, which is not feasible in a decentralised environment with many (possibly commercial) component

providers. It has been suggested that the centralised CM system can be decentralised using inter-file branching which requires a map between original and derived artifacts [Sei96]. This is a little known technique which has not been evaluated by the literature. Further, it is unlikely that industrial practices and robust commercial CM systems will be displaced by the MAE CM system.

MAE does not understand or handle the mapping from architecture to implementation or the implementation evolution, which prevents this approach from being used as a general extensibility mechanism [NEHvdH05].

2.5.3 Easel

Easel is an extensibility and modelling approach for product line architectures. It builds on the ArchStudio toolset [DAH$^+$07], and allows a set of architectural deltas, called a change set, to be expressed against an architecture [HvdH07]. The base architecture can actually be expressed as a change set also. Different change sets can then be associated directly with features, and boolean guards can be used to indicate which change sets (and hence features) can be combined to produce a given variant.

This is a powerful architectural approach which allows change layers to be overlaid on top of an architecture, creating variants of a system. This works for both planned and pre-planned alterations, fulfilling the ALTER and NOIMPACT requirements at the architectural level. Change sets can correct architectural errors caused by the combination of different variants (COMBINE, DETECTANDCOR-RECT), although no support is provided to include these automatically, as per the complementary modules in MixJuice [IT02]. No structural checks are present in Easel, although these are provided in the underlying ArchStudio approach and associated tools.

Easel allows change sets to be edited at any point, and uses globally unique identifiers to track modified architectural elements over time. As long as the same unique identifier is always associated with the same architectural element, merges can always be performed with a predictable result. Currently, a single merge algorithm is defined, although others may be provided in the future.

The addition and removal of components, component instances and connectors are supported. Unfortunately, a replace must be synthesised by a deletion and an addition. This destroys the identity of any component instance being replaced, requiring all connectors binding to that component instance to also be deleted and re-added. As will be shown in chapter 3, an explicit replace construct retains logical identity which preserves existing connectors and other artifacts. This minimises the alterations required for an extension.

Change sets in Easel have been compared to a module facility [HvdH07]. However, although they support some of the features of modules (a level of grouping, coarse-grained dependency management), they do not support exporting of publicly accessible elements or hiding of private ones. As such, modules [BPV98] are still required to manage medium sized and large architectures with multiple levels of abstraction (APPROPRIATELEVEL). Although it would be possible to add modules to the

underlying ADL, these would have to be integrated in some way with change sets, with subsequent overlap. xADL has a group construct which offers namespace features but no dependency management or export control.

Easel and MAE are closely related approaches. In the MAE system, branches hold variants as architectural deltas and branches are merged to combine these. In Easel, change sets hold variants as architectural deltas and combining change sets merges these and the features they represent.

Easel, like MAE, also does not understand or handle the mapping from architecture to implementation or underlying implementation evolution. This prevents the approach from being used as a general extensibility mechanism, although it has many of the required characteristics at the level of architectural configuration.

2.5.4   Merging Approaches for Combining Extensions and Variants

An approach to merging UML models is presented in [AP03]. This considers only the structure of the models at a basic level, and does not address well-formedness or conflict at an architectural level.

Product line architectures have to use merging when it is required that changes in one part of the product line are propagated to another part. Automated support for merging has been provided in [CCG$^+$04], which works by determining the delta changes between different parts of the product line and applying these changes elsewhere. Much of the work involves computing differences between two architectural variants. For example, lexical names are used to establish commonality between variants, which means that merging renamed elements is problematic and must rely on structural heuristics (UPGRADE, COMBINE).

Critical pair analysis, a technique derived from graph rewriting, has been used to detect incompatibilities between parallel branches of a system that each refactor the same elements before being merged [MTR05]. The technique is currently restricted to a fixed set of refactorings, and relies on having changes expressed in graph form.

2.5.5   TranSAT: Using Aspects to Evolve Architecture

TransSAT (Transform Software Architecture Technologies) allows architectural concerns to be specified independently from a core software architecture as aspects, and woven into a coherent whole at a later stage [BCD$^+$04]. The aim is to allow further architectural concerns to be added in a modular fashion. The aspect model used is very much in line with the AspectJ [KLM$^+$97, Asp03] terminology and approach.

Each aspect in TranSAT can modify the underlying architectural structure (components, connectors, interfaces) using a small language where join points are specified declaratively as a set of structure and behaviour matches [BLMDL06]. The set of architectural instructions for remaking the architecture are similar to those expressed in [MK96]. Existing behaviours cannot be removed, which limits this

as a general approach to extensibility or evolution. The encapsulated concerns can be reused between architectures, and structural and behavioural verifications are performed after weaving.

This approach shares limitations with other aspect approaches (see section 2.1.5). The definition of aspects (architectural instructions) is less intuitive than the definition of the primary axis (components and interfaces). Application of the aspects can also lead to architectural errors. Alternatively, structural and behavioural checks may rule out application of an aspect due to interference reasons, despite it being notionally otherwise applicable to a particular situation.

2.6   Enforcing Implementation and Architecture Consistency

It is important that an architectural representation is an accurate description of the system that it represents. If this is not the case, then analysing the architectural specification will not produce results which are representative of the actual system. This area is relevant to this thesis because an architectural approach to extensibility must connect to the implementation level and be able to track its evolution.

2.6.1   ArchJava and AliasJava

An architectural description of a system aims to specify all legal communication paths between components. We have communication integrity when a component may communicate directly only with those components that it is connected to in the architecture. This property is notoriously difficult to enforce as it is possible to bypass most integrity mechanisms by something as simple as passing component references as parameters in method calls to other components.

ArchJava (supplemented with AliasJava) is a type system and language extension for Java that ensures that implementation components only communicate via the specified communication paths, guaranteeing that they cannot use unspecified mechanisms or illegal aliases [ACN02, Ald08]. The type system allows objects to be divided into hierarchical ownership domains. Each object belongs to one domain, and the approach supports hierarchical aliasing. Domain policies specify whether objects in another domain can be accessed.

ArchJava enhances the Java language with architectural constructs. The architectural specification information is added to classes as they are elaborated, ensuring the architecture and implementation are kept fully synchronised. Although the type system is powerful, a downside of this approach is that the implementation language has been modified. As a consequence, ArchJava programs do not work with existing Java integrated development environments.

ArchJava adds no extensibility constructs to Java. A framework approach must therefore be used for extensibility, with the subsequent limitation that only pre-planned extension will be catered for.

2.6.2   ArchEvol

ArchEvol allows an architectural specification to hold version information about the implementation components used in the application. The mapping between architectural and implementation entities is maintained by integrating Eclipse [Obj09g], ArchStudio [DAH+07] and a conventional CM system. Implementation components are each assigned a separate subdirectory in a conventional CM repository. The architectural specification is held in a single xADL file which is versioned.

It is possible to navigate from an architectural representation of a component in ArchStudio to the implementation code in Eclipse via a URL held in the ArchStudio model. Upon modification of the source, a representation of the component's structure is passed into ArchStudio and merged back into the architecture.

Because each component is identified via a separate URL, this approach can use multiple, possibly geographically separated underlying repositories. However, this is not a true decentralised approach and does not permit multiple independent repositories to be separately evolved and synchronised later as per Mercurial [O'S09] or Darcs [Rou05]. For instance, all parties using the system must check out the same component from its owning repository and sharing is further complicated by the fact that the entire architecture is kept in one file.

ArchEvol does not explicitly support extensibility, although this could potentially be combined with the variant xADL extension [Das09] to allow product lines to be expressed and synchronised with the implementation. This approach would have the same limitations as product lines, in that only pre-planned extension would be supported. ArchEvol does not integrate with Easel but conceptually this seems possible.

2.7   Summary

Many existing extensibility approaches require pre-planned extension points, violating the ALTER and NOIMPACT requirements and limiting the types of extensions that can be added. Other approaches avoid this problem by allowing replacement of individual elements in the base, but do not support a compositional hierarchy. This causes a tension between making the architecture fine-grained for easy extension via replacement, or coarse-grained for easy management.

Architectural CM approaches allow variants to be created via architectural deltas in the design stage, permitting unplanned modifications to an architecture. Although satisfying the ALTER and part of the COMBINE requirements they only cover the architectural level, and provide no way to express or understand the evolution of the underlying implementation. As such these cannot be assessed against the NOSOURCE or UPGRADE requirements. Various other approaches allow the architecture and implementation to be synchronised which allows for implementation evolution to be tracked, but no explicit extensibility facilities are provided.

Several of the approaches utilise existing CM systems or provide their own. The latter is a problem

in any commercial setting as existing, mature CM systems are well entrenched and not likely to be displaced. These centralised CM-based extensibility approaches also offer a rigid view of change ownership and control: a full extensibility solution needs to provide more flexible ways of sharing and distributing subsets of the architectural definitions and implementation. For instance, it is feasible that an extension developer may never want to share back alterations of the base application into a central system. The solution must recognise these possibilities, and still provide facilities to handle upgrading and combination of extensions.

In summary, no existing extensibility approach satisfies all of the requirements outlined in section 1.3.

Extensibility approach	Strengths	Weaknesses
Scripting	Limited ALTER. Pre-planned extension via hooks.	Further ALTER requires source modifications, leading to issues with NOSOURCE, COMBINE, DETECTANDCORRECT, and UPGRADE.
Frameworks	Limited ALTER. Pre-planned extension via extension points	Further ALTER requires new framework version, leading to issues with NOSOURCE, NOIMPACT, COMBINE, and UPGRADE.
Plugin architectures	Full ALTER Pre-planned extension via plugin extension points, unplanned extension via plugin replacement.	Plugins are not hierarchical, biasing towards coarse granularity and revealing tension between ALTER and APPROPRIATELEVEL. COMBINE and DETECTANDCORRECT issues.
Aspect-oriented programming	Limited ALTER. Some unplanned extension via aspects.	Many ALTER changes cannot be made using aspects. NOSOURCE, COMBINE, UPGRADE and DETECTANDCORRECT issues.
Product lines	Subset of ALTER via refinement, limited by inheritance-like constructs used.	Limitations of refinement means full ALTER requires source alterations. Issues with NOSOURCE, NOIMPACT, DETECTANDCORRECT, UPGRADE.
Difference-based modules	Subset of ALTER via extends, limited by inheritance-like construct.	Lack of architectural hierarchy leads to APPROPRIATELEVEL issues. Full ALTER violates NOSOURCE.
Viewpoints	Additive subset of ALTER.	Full ALTER is limited by additive approach taken, violating NOSOURCE and others.
Parametrisation	Limited ALTER. Pre-planned extension only.	New parameters violate NOSOURCE, NOIMPACT.
Mixins	Limited ALTER.	Full Alter violates NOSOURCE. Poor support for COMBINE, DETECTANDCORRECT.
Virtual classes	Subset of ALTER, limited by use of inheritance.	Full ALTER violates NOSOURCE and others.

Feature composition	Limited ALTER, based on critical runtime requirements.	Full ALTER is not catered for. Lack of explicit architecture.
JavaBeans and Spring	Bean inheritance gives a limited form of ALTER.	Full ALTER violates NOSOURCE and has issues with COMBINE, DETECTANDCORRECT.
Microsoft COM	Full ALTER via component versions.	Registry approach leads to UPGRADE, COMBINE, DETECTANDCORRECT issues. COM model is limited architecturally, with no explicit connectors.
Darwin	Subset of ALTER via type parametrisation and interface inference.	Adding parameters violates NOSOURCE. Limited support for COMBINE, UPGRADE.
C2SADEL	Extensions must be pre-planned. Support for description of leaf evolution.	No true hierarchical architectures, violating APPROPRIATELEVEL. No way to express evolution of composites (ALTER).
ROOM	Subset of ALTER via structural inheritance.	Full ALTER violates NOSOURCE, as no way is provided to evolve components without source.
File-based CM	Mature approach.	ALTER always violates NOSOURCE. COMBINE, UPGRADE and DETECTANDCORRECT must be performed at the textual level.
MAE	Full ALTER via branching. Works at architectural level.	Adding extension points is notionally creating versions of a component, causing problems with NOIMPACT. Requires consistent use of MAE CM across extenders. No implementation mapping.
Easel	Full ALTER via change sets. Works at architectural level. Supports NOIMPACT, COMBINE, UPGRADE, DETECTANDCORRECT.	Change sets overlap with modules, causing issues for APPROPRIATELEVEL. No explicit architectural hierarchy. No mapping to implementation means NOSOURCE cannot be assessed.
TranSAT	Limited ALTER. Some unplanned extension catered for via aspects.	Full ALTER violates NOSOURCE. Issues with combining aspects (COMBINE, DETECTANDCORRECT).
ArchJava and AliasJava	Extensions must be pre-planned. Support for architectural integrity.	Full ALTER violates NOSOURCE and others.
ArchEvol	Limited ALTER. Supports mapping from architecture to implementation.	Full ALTER violates NOSOURCE and others. No hierarchy.

Table 2.2: Extensibility approaches

Chapter 3

The Backbone Architecture Description Language

The Backbone ADL is a conventional, hierarchical ADL that has been supplemented with three key extensibility concepts. These concepts have been distilled from experience and insights gained working on several industrial systems, and have been formulated to address the requirements of section 1.3.

3.1   The Three Key Extensibility Concepts

In keeping with the definition in section 2.2, a Backbone component is minimally defined as a unit of software that can be instantiated, and which explicitly declares the interfaces that it provides and requires. Building on this foundation, a composite component is one which is made up of instances of other components. As discussed in section 2.2.2, an entire application can be structured as a complex composite component.

Consider a composite component representation of the Desk 1.0 application (figure 1.1), configured with two microphone devices as shown in figure 3.1.
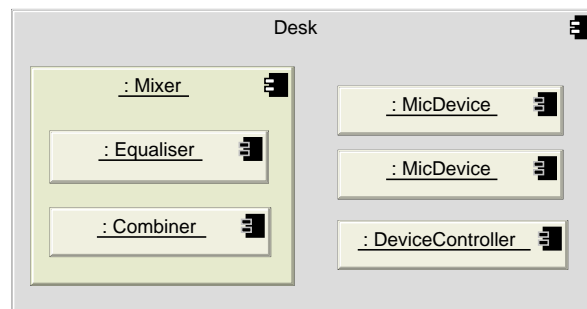
Figure 3.1: The compositional structure of the desk application

This is a simplified UML2 composite structure diagram showing components and instances only. The

`Desk` component is composed of two separate instances of the `MicDevice` component (each instance has a colon in front of its name), one instance of the `DeviceController` component and one instance of the `Mixer` component. Diving down a further level in the compositional hierarchy, the `Mixer` is composed of an instance of the `Equaliser` component and an instance of the `Combiner` component.

The underlying insight of the Backbone approach is that we can make any alterations required to extend an application, by modifying the compositional structure of a base application expressed as a composite component.

Rather than directly editing the composite components to effect any required structural changes, however, we instead provide constructs to express alterations as deltas in an compositional hierarchy. Our alterations are then kept separately from the base application and can be selectively combined with the base at program startup to produce the required changes required for the extension.

Suppose that we now wish to extend the base application of figure 3.1 to support cuing, a turntable device and a CD device as per the desk scenario. To achieve this, we must alter the compositional hierarchy as shown in figure 3.2. The top of the figure shows the hierarchy of the base, and the extended application is shown underneath. The mixer is replaced with an improved, integrated version and a further mixer is added for cue support. One of the microphones is replaced with an instance of the `TurntableDevice` component, and a `CDDevice` is added. We also wish to retain all of the other aspects of the `Desk` component, including any connectors and attributes (not shown), the existing `DeviceController` and one of the existing `MicDevice` instances. Note that the extension in this case has compressed the hierarchy.
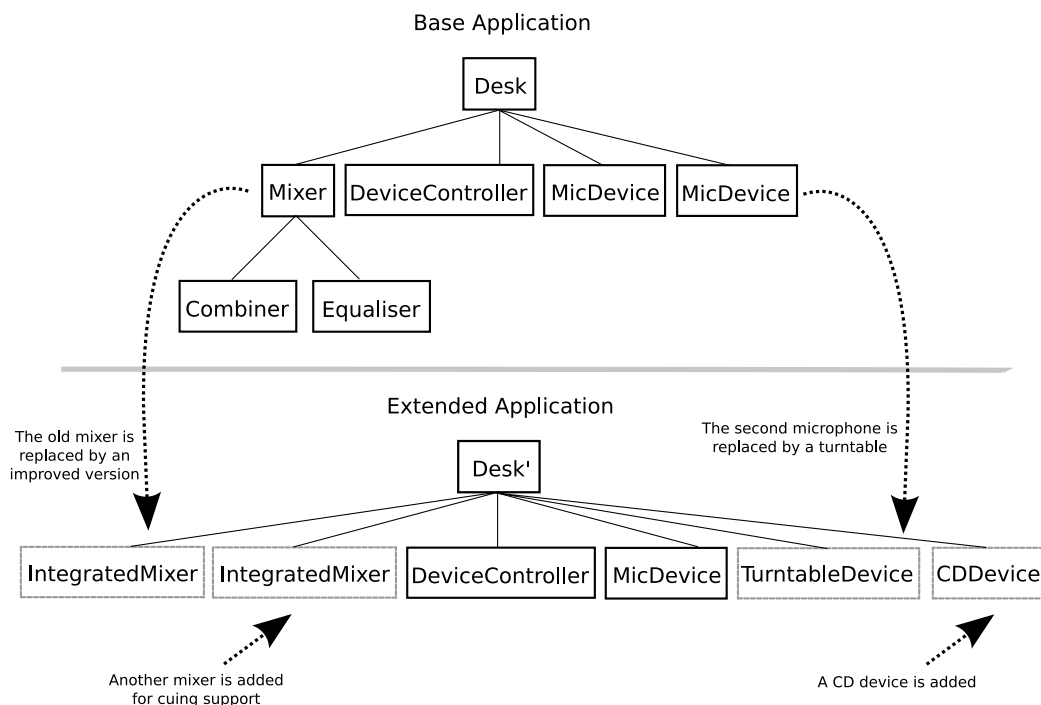
Figure 3.2: Extending the desk application by adjusting its composition hierarchy

Backbone provides the facilities to remake a composition hierarchy, in this manner, by adding three extensibility concepts to an underlying "core" ADL. The core is a conventional, Darwin-like language which has been adjusted to map more closely onto UML2 and object-oriented terminology and concepts.

The three extensibility concepts provide a way to apply architectural deltas to a design. The concepts are listed below.

- Resemblance
  Resemblance allows a new component to be incrementally defined in terms of deltas (add, delete, replace) from the structure of one or more existing components. For instance, `Desk'` could be defined using resemblance from `Desk` in terms of four delta instructions: one to replace the `Mixer` instance with an instance of `IntegratedMixer`, a second to add another mixer, a third to replace the `MicDevice` instance with an instance of `TurntableDevice`, and a fourth to add a `CDDevice`. The other `MicDevice` instance and the `DeviceController` instance will be inherited[1] from the original `Desk` definition. Resemblance also allows structural constituents, such as connectors, to be adjusted in a similar fashion.

- Replacement
  Replacement allows a component to be globally substituted for another in the architecture. For instance, we could use replacement to substitute `Desk'` for `Desk` in the architecture. Anything that previously referenced `Desk` would now reference `Desk'`. The combination of resemblance and replacement allows us to evolve components incrementally without destructively editing the original definition.

- Stratum
  A stratum is a module in the ADL, and this can be used to hold a base application or extension. It groups definitions, declares explicit dependencies on other strata, and controls the visibility of any nested strata. In our example, the `Desk`, `Mixer`, `Combiner`, `Equaliser`, `DeviceController` and `MicDevice` components would be defined in one stratum (`base`), and the `TurntableDevice`, `IntegratedMixer`, `CDDevice` and `Desk'` components would be defined in another (`extension`) which depends upon the `base` stratum (figure 3.3). If we include both strata in our application, we get the extended system. If, however, we only include the `base` stratum, we get the original application desk application without the turntable, CD or cuing support.

In essence, the Backbone approach builds extensibility into an application by encouraging it to be structured as a hierarchical component architecture, at the same time providing a way to rework the composition hierarchy and associated structural constituents. Any part of the hierarchical structure acts as an extension point, and these points arise naturally from system decomposition. Resemblance and replacement are used to modify the composition tree, at the appropriate abstraction level, to alter and extend the architecture (APPROPRIATELEVEL).

---

[1]Resemblance subsumes the notion of inheritance, and hence retains much of its terminology.
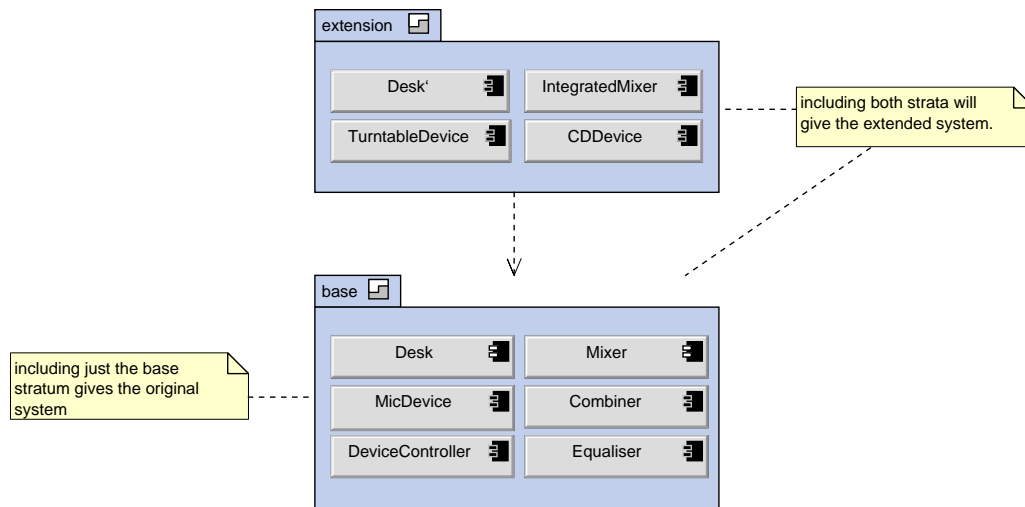
Figure 3.3: An extension stratum depending on a base stratum

Several points are worth noting in the example above. We have not destructively edited the definition of `Desk` itself, yet we have been able to profoundly change the structure of the application to accommodate our changes (ALTER). The replacement of `Mixer` with `IntegratedMixer` in the compositional hierarchy has somewhat flattened the hierarchy. In practice, it is common that both compression and expansion of the hierarchy may result from an extension.

The changes for the extension will only be applied if we form a system by including both the base and extension strata. If someone chooses to only include the base stratum, they will be left with the original base application (NOIMPACT). We can therefore choose the combination of features to include in a system by structuring and choosing strata with the appropriate dependencies.

By phrasing other elements as entities with structure, we are able to apply the extensibility concepts to these also. In this way, resemblance and replacement are used in Backbone to describe the evolution of interfaces and their substitutability in an architecture.

3.2 The Backbone Component Model

This section explains the Backbone component model in detail. UML2 composite structure diagrams are used to depict the graphical form.

Table 3.1 shows the graphical symbols for the core Backbone ADL. Apart from *port links, aliased slots* and *placeholders*, these are standard UML2 constructs which we do not reinterpret.

Symbol	Backbone core construct
	Leaf component
	Interface
	Operation (inside an interface)
	Port
	Required interface (from a port)
	Provided interface (from a port)
	Attribute (inside a component)
	Primitive type (of an attribute)
	Port link (between ports inside a leaf component)
	Composite component
	Part (inside a composite component)
	Connector (between ports inside a composite component)
	Slot (inside a part)
	Aliased slot (inside a part)
	Placeholder component
	Delegation connector (between ports inside a composite component) or Dependency (between strata)

Table 3.1: The symbols for the Backbone core constructs

Table 3.2 shows the graphical symbols for the primary Backbone extensibility constructs, which correspond to the three extensibility concepts discussed earlier. These are not part of UML2.

Symbol	Backbone primary extensibility construct
	Resemblance
	Replacement
	Stratum

Table 3.2: The symbols for the primary Backbone extensibility constructs

Table 3.3 shows two further extensibility constructs, which are secondary in that they are "syntactic

sugar" for a combination, or selective application, of the primary extensibility constructs.

Symbol	Backbone secondary extensibility construct
🚀	Evolution (resemblance and replacement used together)
✖	Retirement

Table 3.3: The symbols for secondary Backbone extensibility constructs

Backbone has a textual form also, and this is sometimes shown to the right in the diagrams. A full mapping between UML2 and Backbone is outlined in section 5.2.11, including restrictions placed on the UML2 model to support extensibility.

3.2.1 Core Constructs

In keeping with the definition in section 2.2, a Backbone component is a unit of software that can be instantiated, and explicitly declares the interfaces that it provides and requires. A component can be either a composite, a leaf, a factory[2] or a placeholder.

Leaf Components ▤

A *leaf component* is atomic and not further decomposed into other component instances. Figure 3.4 shows two leaves along with the textual definition of the first. The resemblance between the interfaces will be explained shortly. A leaf is associated directly with an implementation class.
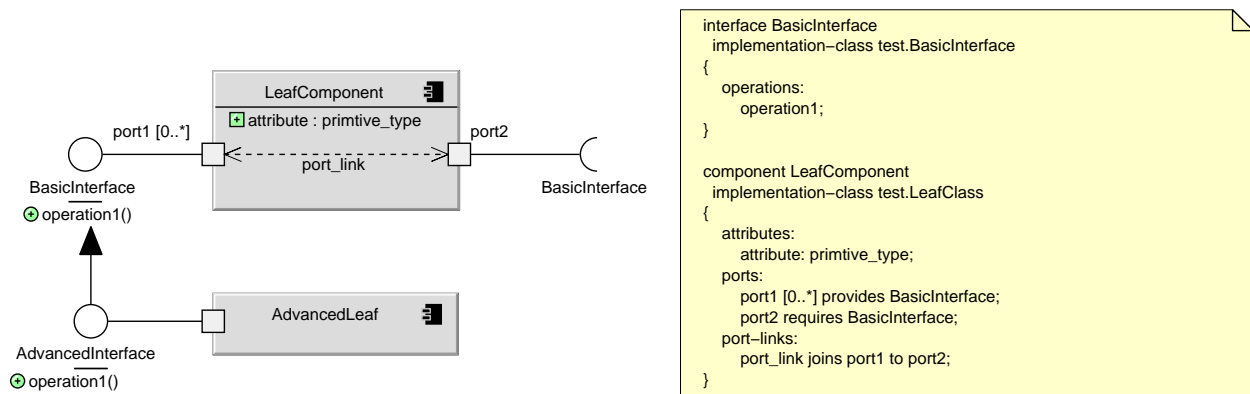
Figure 3.4: A leaf component

Interfaces ◯ **and Operations** ⊕

An *interface* denotes a contract between two parties for a service. Each interface can contain a set of *operations*, and is associated directly with an implementation interface, as shown in the textual form of figure 3.4. As explained in section 3.2.2, an interface may resemble other interfaces.

---

[2]Factories allow for dynamic component instantiation, and are discussed in section 6.3.2.

Ports ⬚, Required Interfaces ↵ and Provided Interfaces ◌

Ports serve to fully insulate a component from its environment, and can specify a number of *provided* and *required interfaces*. A port can be denoted as *indexed* by specifying a multiplicity of either a single fixed bound, or a lower and upper bound together. The default multiplicity of a port is [1]. A port with a multiplicity of [0..*] means that it can support any number of connections to it.

An indexed port can be marked as *ordered* by using the `is-ordered` keyword in the port definition, meaning that the order of connections to the port is significant. This is related to the subject of connector indices, and discussed further below.

Attributes ⊞ and Primitive Types ▤

A component may have *attributes* of *primitive type*, which represent a projection onto the internal state of the component.

Port Links ↳

Although a leaf component has no further decomposition in the architectural description, it may have internal logic and structure that relate the provided and required interfaces of several ports together. These internal linkages can be expressed by a *port link,* in lieu of explicit connectors, as shown in the leaf component of figure 3.4. Links propagate interface information between ports for the purposes of port type inference (see below), when an instance of the leaf is used inside a composite component. Port links can be between ports of different multiplicity, and are always optional.

Composite Components ▰, Parts ⊡ and Connectors ⤢

A *composite component* contains *parts*, which are instances of other components, as shown in figure 3.5. Parts can either be wired together, or back to one of the component ports, using connectors. A connector has exactly two *connector ends*. The parts of a component represent its initial configuration.
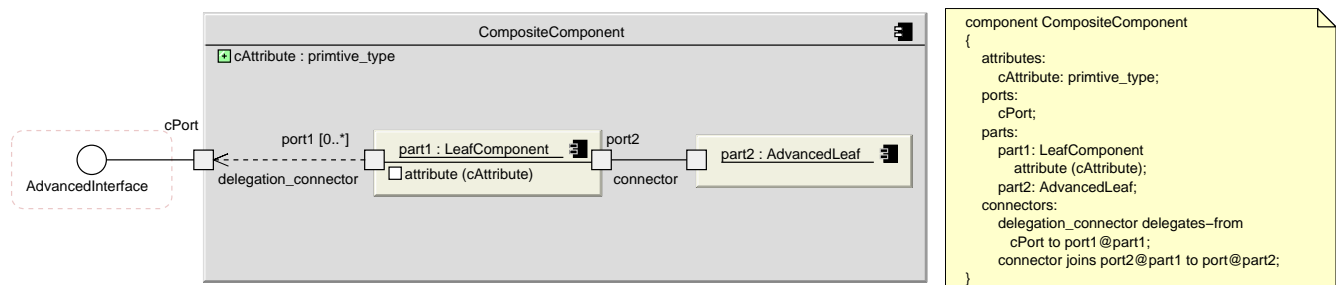
Figure 3.5: A composite component

As with leaf components, a composite may also specify ports and attributes. A composite has no need to specify port links, as the structure is fully described by the parts and connectors.

A composite component does not specify an implementation class; the composite exists only in the ADL and is a purely structural entity. In this sense, a composite differs from the conventional UML2 usage where it is denotes an implementation class that can have its own behaviour [OL06]. A composite in Backbone (and Darwin) is shorthand for a set of wiring instructions, and can always be flattened into a connected graph of leaf parts [KME95].

We refer to components and interfaces collectively as *elements*. We refer to the insides of components or interfaces (i.e. ports, parts, attributes, operations etc) as *constituents*.

Connector Indices: Numeric and Alphanumeric

When connecting to an non-ordered port, a connector end does not have to explicitly specify an index. In this case, the connections will be made in an arbitrary order reflecting the fact that order is not important to the port.

When connecting to an ordered port, however, an index must always be supplied. *A numeric index* (e.g. `[2]`) or an *alphanumeric index* (e.g. `[A1]`) can be used. The latter type is always preferred, as it is then possible for an extension to insert a new connection between any two existing connection ends by choosing an index that is lexically between the existing indices. For example, if the base architecture has connectors that use indices of `[a]` and `[b]` to connect to a port, then an extension could use `[Z]` to insert a connection before in the ordering, `[aa]` to insert between, and `[c]` to add the connection after the existing ones.

If numeric and alphanumeric indices are both used to connect to a given port, the numeric indices will be applied first. Using indices on a non-ordered port is allowed, but not mandatory.

Slots ▫ and Aliased Slots ☐

A *slot* is an attribute of a part that has been bound to a value, such as the slot `attribute` in `part` of figure 3.5. The value can be a literal (e.g. `100`) or a copy of an enclosing attribute (e.g. `= cAttribute`). A slot may also be *aliased* onto an enclosing attribute (as per the slot shown in the figure 3.5). If the same enclosing attribute of the composite is aliased onto several slots of different parts, they will all literally share the same variable. This allows buried instance state to be propagated up to the composite level and shared between parts.

Delegation Connectors ↗

A *delegation connector* establishes one port as an alias for another. In our example, `cPort` is an alias for `port1`. This tends to be used as a convenience to prevent the need for many explicit connectors

between ports with multiplicity.

Placeholder Components ⊟

A *placeholder component* represents a component that has not been fully elaborated in the architecture, supporting top down architectural specification. It cannot contain parts and is not associated with an implementation class, but may contain port links, attributes and ports.

Placeholders are useful for defining the general "shape" required of other components, and resemblance can be used to inherit constituents from a placeholder when forming a new concrete component. A placeholder can also be evolved into a concrete component using replacement. Parts of placeholder type can be used to represent unelaborated sections of a composite, with the caveat that a composite with placeholder parts cannot be instantiated as part of the composition hierarchy of an application.

Figure 3.6 shows a placeholder definition with the same shape as `LeafComponent`.
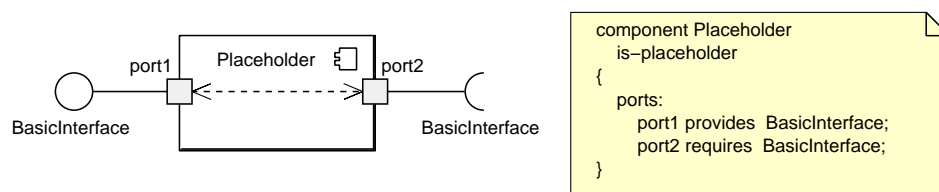
Figure 3.6: A placeholder component

Port Type Inference

The provided interface of `cPort` was inferred in figure 3.5 and did not need to be specified explicitly. The port of the `AdvancedLeaf` part provides a subtype of `BasicInterface`, called `AdvancedInterface` (figure 3.4) and this propagates through `part1` using that leaf's port link information.

Port type inference completely removes the need to specify provided and required interfaces for ports of composite components. This is useful because when even a small number of component structures are replaced using the extensibility constructs, many port interfaces can alter quite dramatically as the changes propagate to many component definitions in the composition hierarchy. With inference, updates to the port specifications are handled automatically and we do not have to manually adjust the affected composite component definitions.

Unique Identifiers

Although figures 3.4 and 3.5 show elements and constituents with names, these are purely descriptive. In actuality, each element or constituent has a universally unique identifier (UUID) allocated to it that remains constant even as the name changes, allowing Backbone to track logical identity even

when elements are evolved or moved between strata. UUIDs also provides a robust way to reference individual constituents of a component for the resemblance construct.

The Evolve modelling tool (section 5.2) handles the allocation of UUIDs and completely hides these from the user interface so that only descriptive names are presented. Although in the remainder of this thesis we will continue to show names for reasons of clarity, in all cases UUIDs would be allocated.

3.2.2 Extensibility constructs

Resemblance 🪁

Resemblance is used to define a new element in terms of similarity to other elements, using delta alterations (add, delete, replace) to add or change any inherited constituents. For instance, we can use resemblance to define a new component in terms of changes to the part and connector structures of existing components.

Consider figure 3.7, where we have defined `NewComposite` in terms of `CompositeComponent` from figure 3.5. We have deleted `part2`, added `cPort2` and replaced `connector`. Other constituents such as `part1` and `delegation_connector` were inherited. Notice how the removal of `part2` has caused the interface provided by `cPort` to change to `BasicInterface` via port type inference.
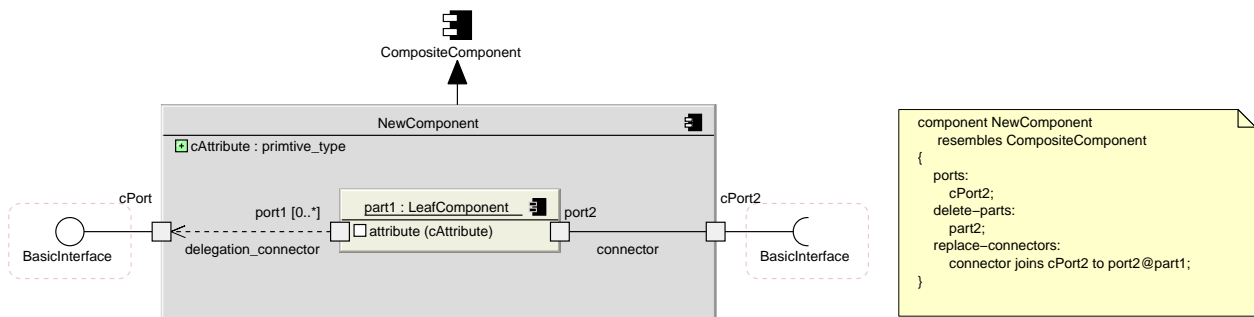
Figure 3.7: Using resemblance to define a new component in terms of deltas

The constituents of a component that can be added, deleted or replaced using resemblance are component name, parts, ports, connectors, attributes, port links, retirement status, implementation class and stereotype[3].

Resemblance can be used to define a composite component from a leaf (and vice versa). Defining a composite from a leaf involves deleting the inherited implementation class name constituent and adding parts and other structure. Defining a leaf from a composite involves deleting the parts and adding an implementation class name constituent via a delta.

The use of deltas even for descriptive names and implementation class mappings allows us to take a consistent approach to change, extension combination and conflict resolution for any constituent of a

---

[3]Stereotype usage in Backbone is explained in section 6.1

component or interface. Deltas are used uniformly in all cases[4].

An interface is also an element and can resemble other interfaces, as described earlier. Interface constituents are interface name, implementation class name and operations. Resemblance subsumes the inheritance construct completely, and a subtype relationship for interfaces is a resemblance relationship where no deletions or replacements of operations have been performed. For instance, `AdvancedInterface` in figure 3.4 is defined using resemblance from `BasicInterface`.

Replacement ⬈ and Evolution ⬈

Replacement globally replaces one element with another in the architecture.

Evolution is expressed by combining resemblance and replacement. Accordingly, it is regarded as a secondary construct and the graphical symbol is the overlay of the resemblance and replacement symbols. It defines a new element incrementally in terms of an existing one (resemblance), and then replaces the previous definition with the new one (replacement).

Although we previously used resemblance to define `NewComponent` in terms of changes to `CompositeComponent`, thereby incrementally altering the inherited structure, this did not affect the definition of `CompositeComponent`. If a component contained `CompositeComponent` parts, its structure would be unaffected by the presence of `NewComponent`. Replacement, on the other hand, allows us to affect existing usages by globally replacing the previous definition with our new definition.

Consider how this works in figure 3.8. We create another component, in the spirit of `NewComponent`, via resemblance from `CompositeComponent`. Simultaneously, we use replacement to replace the old definition. In this case we did not specify a new name, so the name is inherited. The component name is shown with a prime symbol after it to denote that it evolves the original.
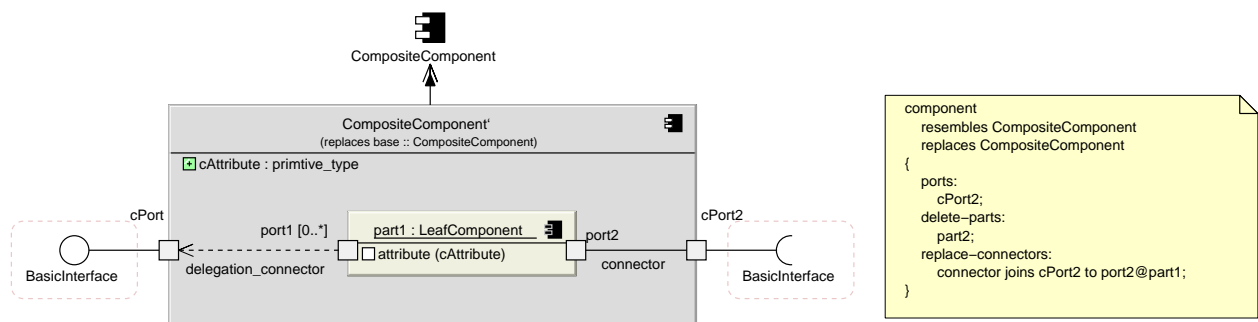
Figure 3.8: Evolving a component with resemblance and replace

Resemblance is useful as a construct in its own right for effecting component reuse [MMK06][5]. Replacement used on its own is possible, but rarely useful as it always introduces breaking changes because no structure or ports (including their UUIDs) will be inherited from the element being replaced.

---

[4]The textual and graphical forms hide the use of deltas for name and implementation class.
[5][MMK06] refers to *replacement* as *redefinition*.

Retirement ✖

Retirement allows an element to be retired, and it is an error to further refer to something that has been retired. Retirement is not a primary construct in the ADL per se, but is instead modelled as a boolean constituent which can be altered using deltas. As such, retirement is achieved by evolving the element we wish to retire, and creating a replace delta to set the retirement status to true.

The graphical and textual form are shown in figure 3.9.
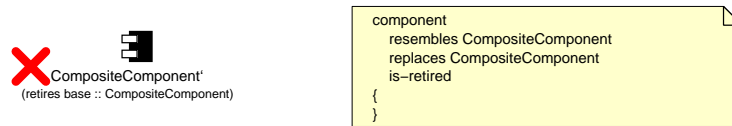
Figure 3.9: Retiring a component

Strata ▣ **and Strata Dependencies** ↗

A *stratum* is a nested module in Backbone. It groups related definitions corresponding to an extension or base application, and indicates their dependencies. A stratum must explicitly declare which other strata it depends on, and which child strata are exported as visible to others.

Combinations of strata are used to represent application variants. Figure 3.3 demonstrates how a base application can be placed in one strata, and an extension in another. If we take the {`base`, `extension`} strata combination, we get the variant corresponding to the extended application. If we just take the {`base`} combination, however, we are left with the original application.

A stratum forms the unit of extension in Backbone, and also the unit of ownership and sharing. Each stratum is owned by a single party that has modification rights to it. We can therefore use strata to model the ownership structure of an architecture, and map this onto a community of base and extension developers. The modelling tool and runtime environment are centred around the import and export of strata for distributing extensions and subsets of an architecture.

A system is assembled out of strata, and it is not possible to selectively include only some elements from a stratum. As such, it is not sensible to package both a definition and any replacements for it in the same stratum, as only the evolved version would then be visible. For this reason, replacement, evolution or retirement of an element must be in a different stratum to the original definition.

A stratum can either be *relaxed* or *strict*. If stratum S is relaxed then it propagates its dependencies: any stratum that depends on S also has visibility of the strata that S depends on, and so forth in a transitive manner. This allows us to use Backbone to model strict and relaxed layered architectures as described in [BMR+96].

A stratum that contains an element replacement (including evolution and retirement) must be marked as *destructive*, and strata with nested destructive strata must be also marked as destructive. The

symbols for the different strata types (table 3.4) allow a developer to quickly see which parts of the system involve replacement and could therefore cause conflict when combined with other extensions.

Symbol	Strata variant
	Relaxed (white / grey)
	Strict (white / black)
	Destructive and relaxed (red / grey)
	Destructive and strict (red / black)

Table 3.4: The symbols for strata types

The stratum concept is related to the UML2 package concept. Unfortunately, packages suffer from limitations which restrict their usage as a unit of import and export from a modelling environment [SW98]. This is explained in section 5.2.11.

3.2.3   Summary

The Backbone ADL consists of a set of three primary extensibility constructs, added to a conventional Darwin-like ADL which has been aligned to UML2 terminology and concepts.

The extensibility constructs derive from the insight of manipulating a component composition hierarchy in order to restructure it for an extension. Resemblance allows the reuse of existing composition hierarchies, by permitting a new element to be defined in terms of deltas from existing elements.

Used on its own, however, resemblance always defines a new component (and composition hierarchy) from existing ones. We also need the ability to remake the hierarchy of an existing component. At the same time, we want to prevent direct editing of the original definition by extension developers who are not the owners of that definition, to respect the NoImpact requirement. To accomplish this, we further add the replacement construct, which replaces an existing definition with a new definition. Used together, resemblance and replacement can express the incremental evolution of a component or interface in a hierarchical architecture.

Finally, strata can be used to group definitions for an extension and express the extension's dependencies on other parts of the system. Strata can also be used to model the ownership structure of the architecture. We use combinations of strata to express variants of an application.

3.3   Modelling the Audio Desk Scenario in Backbone

In this section, we work through the audio desk scenario from section 1.2, to demonstrate how the extensibility constructs satisfy the requirements outlined in section 1.3.

3.3.1   Relationships in an Extension Setting

Before modelling the desk application itself, let us examine the relationship between the parties involved in the extension setting, as shown in figure 3.10. The arrows show the flow of software between parties.
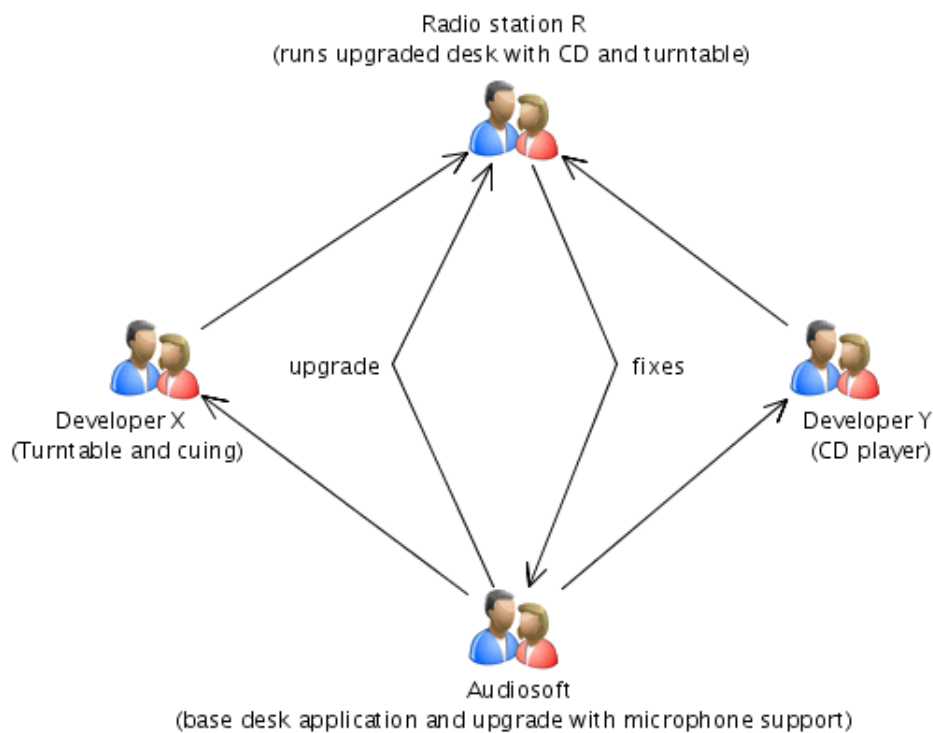
Figure 3.10: Software flows between parties in the desk scenario

The base desk application is developed by Audiosoft. Audiosoft distributes the base to Developer X who creates an extension to add turntable and cuing support. Developer Y also receives the base and creates an extension that adds CD player support.

Audiosoft have also recently produced an extension which upgrades the mixer. X and Y have not seen this upgrade or adapted their extensions for this version. Finally radio station R wishes to run an upgraded desk with a microphone, CD player and turntable.

To further complicate matters, consider that R wishes to potentially patch or fix the upgraded desk version and feed these changes back to Audiosoft.

The relationships between the parties reveal a number of issues which occur frequently in an extension setting. The extensions of X and Y are created independently, but must eventually be combined despite neither X nor Y having visibility of, nor a particular interest in adapting their extension for

the other. The job of combining these extensions and correcting any issues falls eventually to another party further towards the eventual user of the application; R in this case. Secondly, upgrades to the base may be packaged as an extension also[6], and not all parties may have access to the upgrade. It is common to want to combine a number of extensions, each originally developed against a different version of the base.

A further issue is that R wishes to send fixes back to Audiosoft. However, any fixes cannot reference the extensions from X and Y as Audiosoft has no exposure to these.

Finally, we cannot assume that a common configuration management (CM) system exists to allow all parties uniform access to the architectural definitions and implementations. Some software may be distributed by simply sending it to others. Other parties may use CM systems internally and control the evolution of their own software in this way, but will generally not share a repository with other parties.

3.3.2 Using Strata to Model Ownership and Relationships

We now turn our attention to structuring a set of strata and their dependencies, such that each party owns associated strata to contain the software that they produce. After reversing the direction of the arrows in figure 3.10 to represent dependencies rather than software distribution, we arrive at the structure shown in figure 3.11.

Each stratum contains both architectural descriptions and related leaf and interface implementations, and has a single owner. The owner is the only one permitted to do any direct modification of the definitions in the stratum. Strata can then be distributed to non-owning parties, in the reverse direction of the dependencies, who must treat them as read-only and use the extensibility constructs to effect any alterations required. This approach guarantees that each strata has a single definitive source and no merging or common CM system is required to manage parallel branches of a single stratum. Of course, this arrangement does not prevent parties from using a CM system to manage the evolution of strata they own, if this is desired.

In our strata graph, Audiosoft owns the `desk 1.0` stratum. X owns `turntable` and Y owns `CD`, each of which depend on `desk 1.0`. Audiosoft also owns a separate stratum `upgraded` which is used for the extension corresponding to the upgrade. In this way, both the base and upgraded desk are available via different strata combinations. The `combined` stratum is owned by R, and its dependencies combine all extensions into a single application. R is using the `fixes` stratum to hold any fixes to the upgraded desk to be sent back to Audiosoft, and the dependency structure ensures that this cannot contain references to the `turntable` or `CD` extensions. Other variants could be similarly structured as strata with the appropriate dependencies.

Parties do not need generally need to see strata upwards in their dependency graph, and do not have to concern themselves with the maintenance of these. An exception to this is stratum `fixes`, which

---

[6]Audiosoft could simply edit the desk application directly to produce the upgrade, but this may violate the NOIMPACT requirement for existing users.
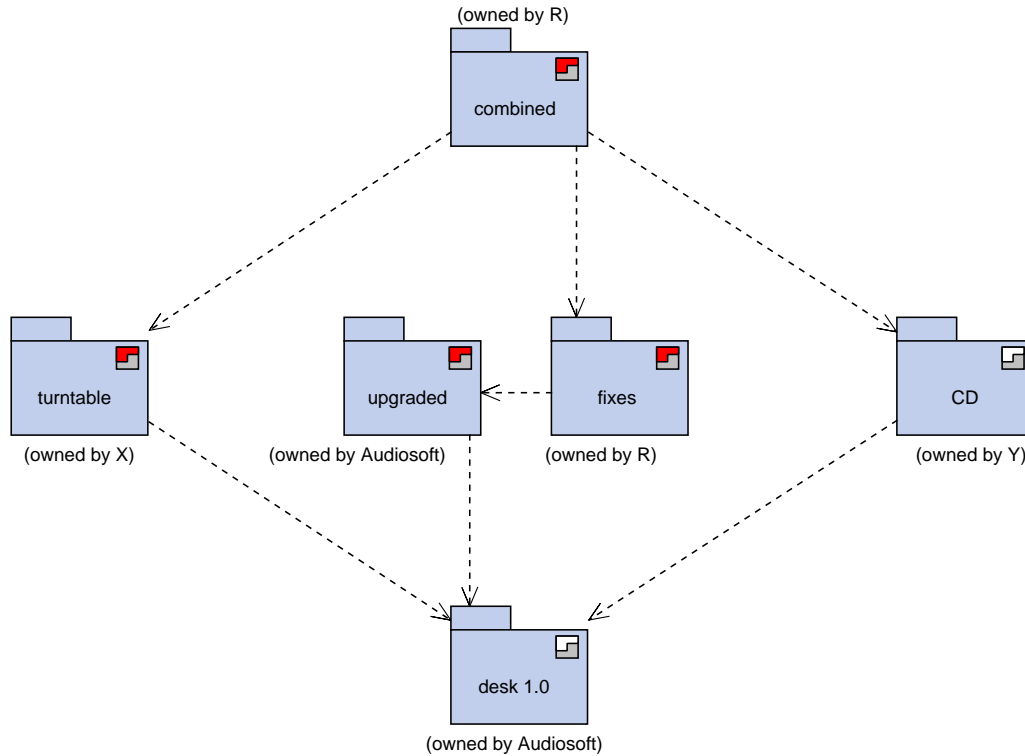
Figure 3.11: Strata dependencies reflect the relationships between parties

is used by R to send fixes back to Audiosoft.

3.3.3   The Base Application

Audiosoft defines the `Desk` component in the `desk 1.0` stratum, which it owns. This forms the base application that others will extend. In this section, we will elaborate the `Desk` component using a top-down decomposition.

The audio desk (`Desk` in figure 3.12) contains a number of volume knobs and switches (`Device-Controller`), which control a set of audio devices. The desk is configured with two microphone devices (`MicDevice`). The audio from the devices is fed into the mixer (`Mixer`), which combines them and produces a single audio output channel. Order is important for the `controllers` port of `DeviceController`, as this determines which knobs and switches control which device. As this port is marked as ordered, we need to specify an index for each connection end bound to it (`[a][b]`). Note that the `out` port definition does not need to specify an interface, as this is inferred from the parts and connections.

Figure 3.13 shows the `AudioDevice` placeholder and `DeviceController` leaf component definitions. The placeholder serves to describe the general shape of an audio device: each device should provide an `IDevice` interface for device control, and require an `IAudioChannel` interface for outputting its audio packets.
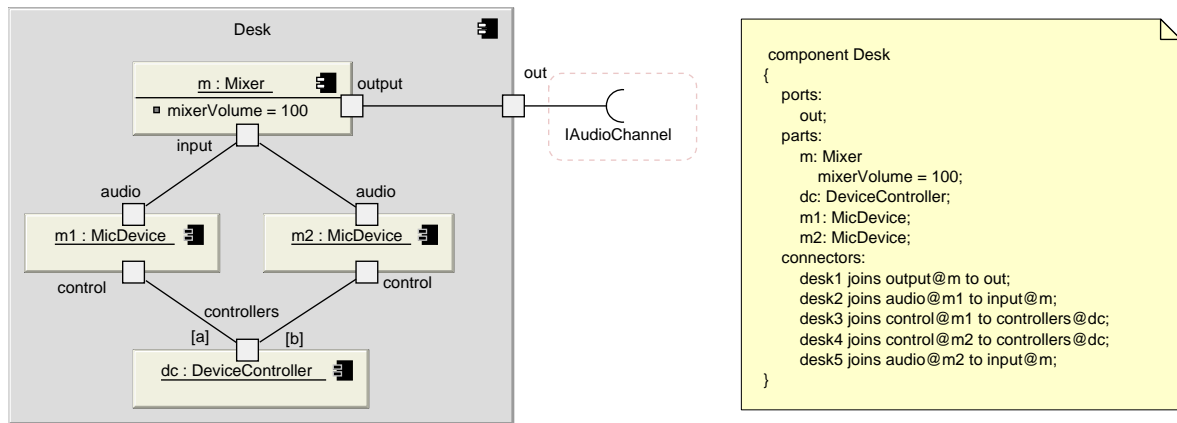
Figure 3.12: The Desk component

DeviceController requires an ordered collection of IDevice interfaces, representing the devices it controls via the play and stop operations.
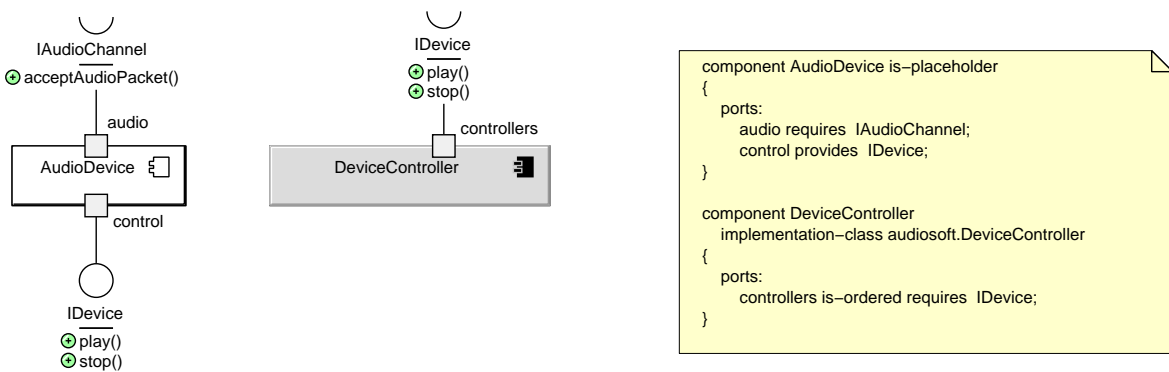
Figure 3.13: The AudioDevice and DeviceController components

The MicDevice leaf is defined using resemblance from AudioDevice, as shown in figure 3.14.

MicDevice is implemented by the audiosoft.MicDevice class. The implementation of this class must follow a convention, which the Backbone runtime engine expects. The Java class definition would start as follows[7].

```
package audiosoft;
public class MicDevice {
    public void setAudio_IAudioChannel(IAudioChannel channel) { ... }
    public IDevice getControl_IDevice() { ... }
```

The method names are formed by concatenating a get (for provided) or set (for required) with the port name and the interface name. The Backbone runtime will instantiate an instance of the class and connect to it using these methods during program execution.

---

[7]Currently, leaf and interface implementations are written in Java, although the approach is equally applicable to other object-oriented languages.
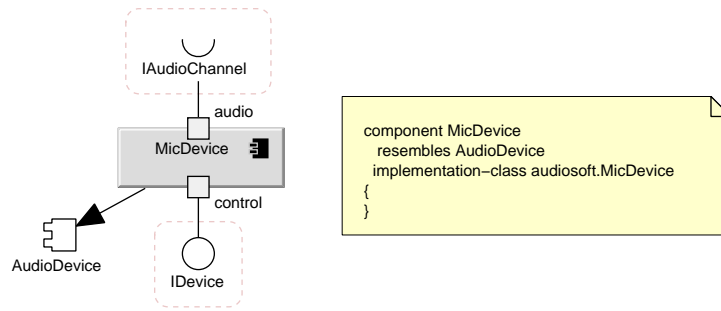
Figure 3.14: The MicDevice component

The leaf and interface implementations logically form a part of the stratum and must be distributed when sending the stratum to another party. In keeping with the NoSource requirement, the implementation source is not required when distributing. The compiled artifacts suffice, as no direct modification of the source code of the base is required when extending using the Backbone constructs.

The definition of the `Mixer` composite is shown in figure 3.15. It provides a number of audio channels for accepting multiple audio inputs and combining them into a single output (`Combiner`) and also has an frequency equaliser so that the signal can be adjusted (`Equaliser`). Note that the `Combiner` `volume` attribute has been aliased onto the `mixerVolume` attribute. This is a way of "pushing up" attributes, to avoid having them buried in the compositional hierarchy.
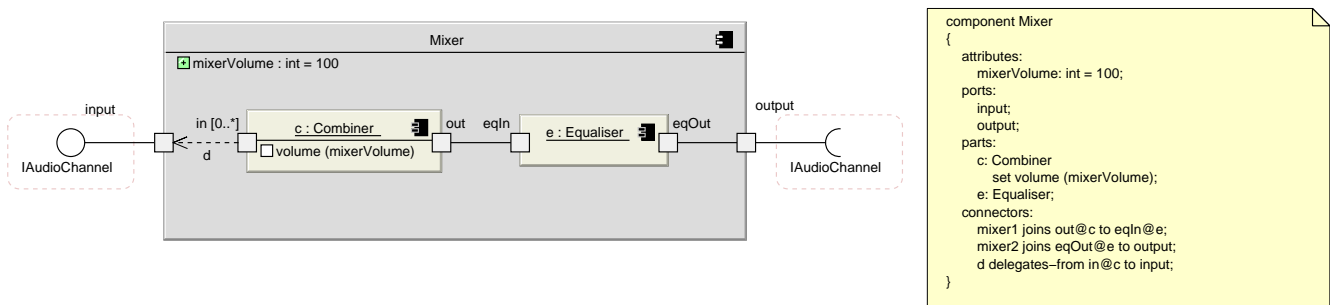
Figure 3.15: The Mixer component

The `Combiner` and `Equaliser` leaf definitions are shown in figure 3.16. Note that the `in` port of `Combiner` is not ordered, as the manner in which the audio signals are combined makes no difference to the output. `Equaliser` has an attribute with a default value. As such, it is not necessary to specify a slot for this in the `Equaliser` part of `Mixer`.

The final composition hierarchy of Desk is shown in the top of figure 3.2.
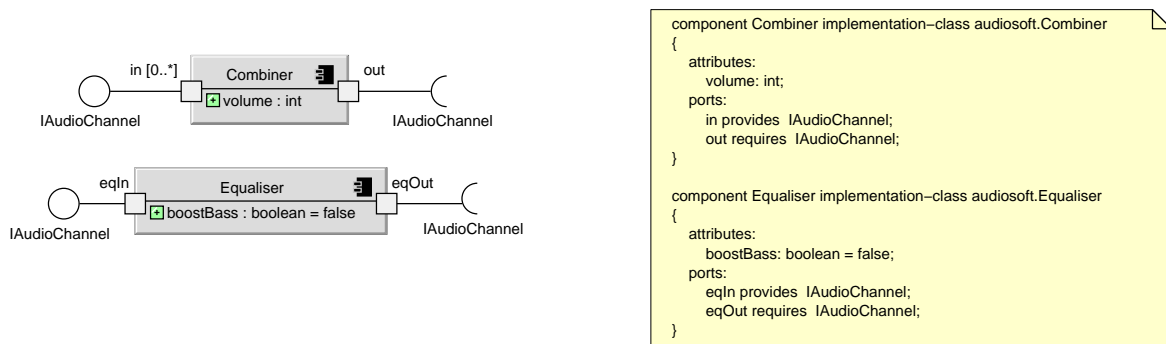
Figure 3.16: The Combiner and Equaliser components

3.3.4 The CD Player Extension

The `desk 1.0` stratum is distributed to developer Y who is tasked with adding an audio device to control a CD player. This is a type of planned extension as the application was specifically designed to allow new devices to be defined and connected into the desk.

Developer Y owns the `CD` stratum, and this is used to contain their component definitions and implementations.

`CDDevice` is defined in terms of resemblance from the `AudioDevice` placeholder, as shown in figure 3.17. Resemblance allows the component to reuse the shape of the placeholder, resulting in a terse definition.
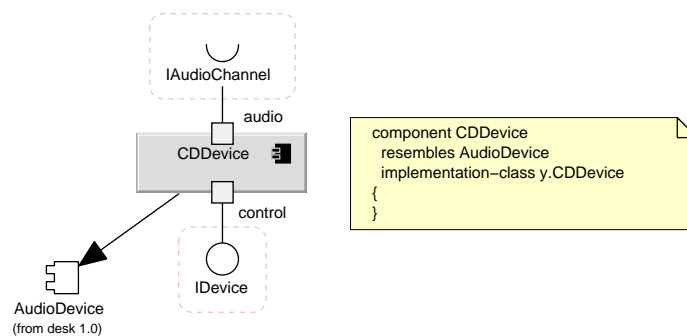
Figure 3.17: The CD device component

Y then uses an evolution of the `Desk` component to connect in a `CDDevice` part (figure 3.18), leaving all the other structures intact. Note that the textual form uses a combination of resemblance and replacement, in keeping with the definition of evolution as a secondary construct (table 3.3). Even if the graphical view does not explicitly show the evolution relation, the definition can be seen to be an evolution due to the prime after the name.

The addition of a CD device was accomplished using only the extensibility constructs, without directly editing the definitions in the base stratum. The `Desk'` and `CDDevice` definitions (and implementation
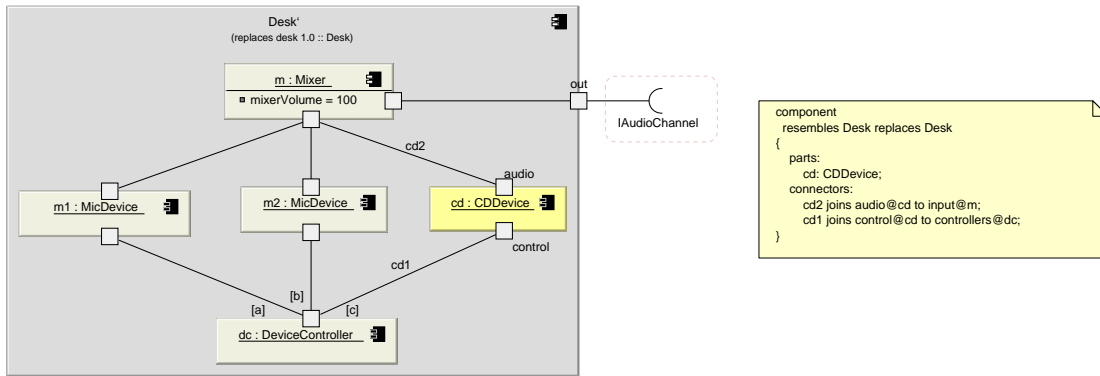
Figure 3.18: The evolved desk with CD support

artifacts) can now be packaged into the CD stratum and distributed to parties who wish to incorporate this extension into their system.

The composition hierarchy is the same as the hierarchy of the base, with the single addition of a CDDevice part.

3.3.5   The Turntable Extension

The desk 1.0 stratum is also distributed to developer X, who is tasked with adding an audio device to control a turntable. This device requires cuing facilities, where the audio can be sent to an off-air bus so that the start of a track can be located before broadcasting.

Unfortunately the base application does not provide cue support, so X must extend the application to add this. This is a type of unplanned extension as the base application did not foresee or cater for this type of change.

Developer X owns the turntable stratum, and this is used to contain their component definitions and implementations. The definition of TurntableDevice is shown in figure 3.19. Again, resemblance from the placeholder is used to inherit the shape, and the cue port is added.
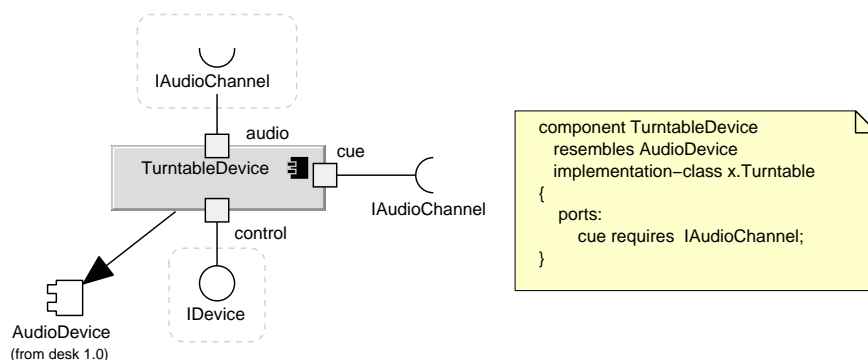
Figure 3.19: The turntable device component

The mixer of the base application does not have a way to handle cue inputs separately from on-air inputs. To add this facility, we first create a `CuingCombiner` component, which resembles the base `Combiner` component, but adds `cueIn` and `cueOut` ports (figure 3.20). Note that this is a leaf, and to add cuing support X must also create a new implementation class named `x.CuingCombiner`.
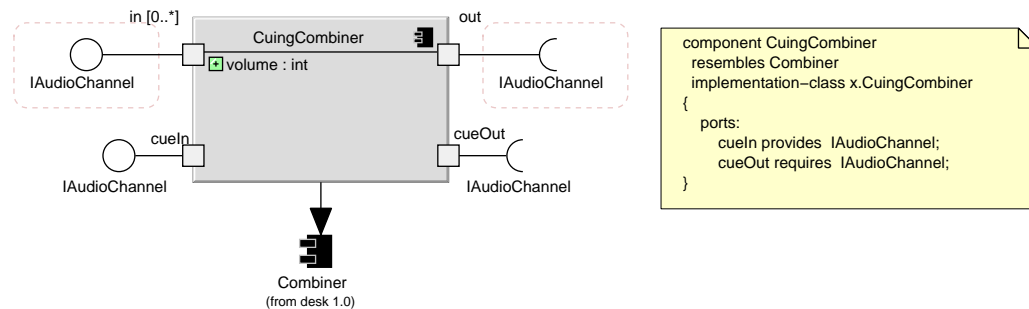
Figure 3.20: The combiner with cue support

The new implementation class can be phrased in a number of ways, as long as it respects that the source code of `audiosoft.Combiner` is not available. In this case, X makes `x.CuingCombiner` inherit from the `audiosoft.Combiner` class, using implementation inheritance to avoid recreating the entire implementation. This is not always possible leading to occasional reimplementation, as inheritance is limited to additions and selective overrides, whereas resemblance also allows deletions and replace also. As reimplementing a coarse-grained leaf can be costly, Backbone intentionally biases developers towards fine-grained leaf components. Backbone also allows for architectural hierarchy via composition, so this level of granularity does not pose a management or abstraction problem.

X now needs to incorporate the new combiner into an updated mixer. `CuingMixer` is defined via resemblance from `Mixer`, as shown in figure 3.21. X has replaced the inherited `Combiner` part with a `CuingCombiner` part. Note that as part of the replace, the name was altered from c to cc. As described in section 3.2, most names in Backbone are purely descriptive[8] and all artifacts also have a UUID. Because the UUIDs of parts and other constituents remain constant even when being replaced, a name change does not cause problems for inherited connectors. Note that `CuingCombiner` also inherits the ports of `Combiner`, which means that the inherited connectors still connect to the correct ports.

---

[8]Except where the names are used to denote leaf component ports and attributes which correspond to implementation names.
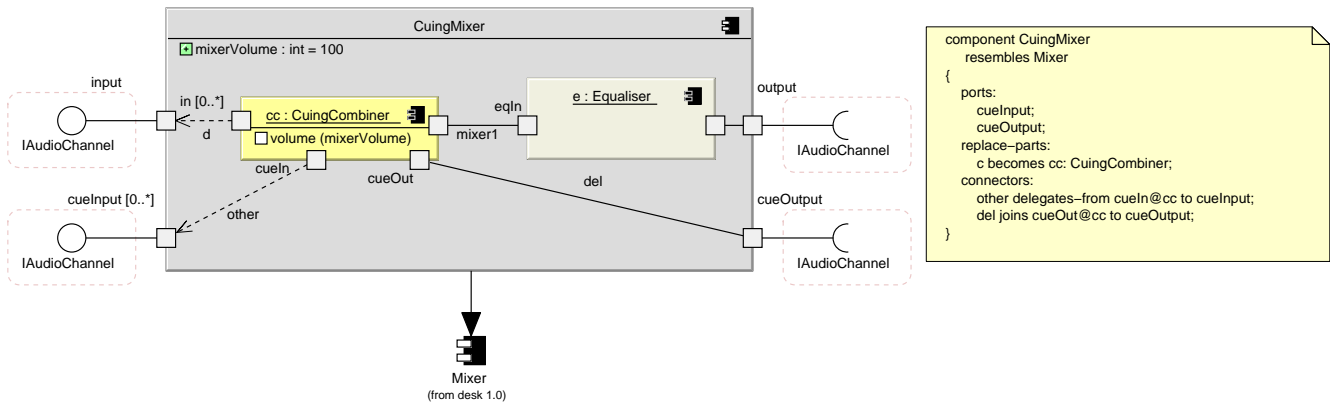
Figure 3.21: Defining a mixer with cue support

Finally, figure 3.22 shows how Y has evolved `Desk` to replace the mixer with a `CuingMixer` part, and that one of the microphones has been replaced with a `TurntableDevice` part. As `CuingMixer` and `TurntableDevice` both resemble elements in the base and hence inherit their constituents and UUIDs, we can replace the parts in our evolution of `Desk` without having to replace all the existing connectors.
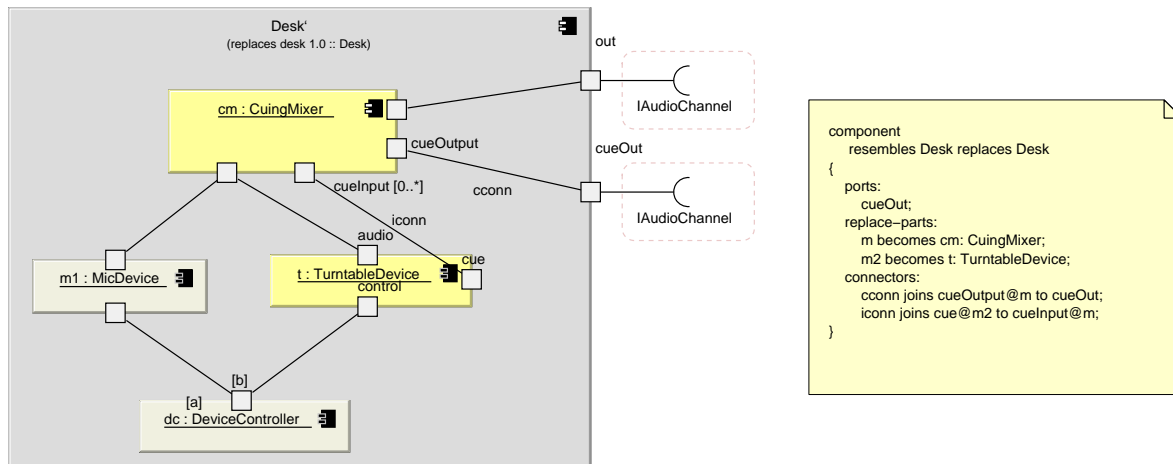
Figure 3.22: Evolving the desk to provide cue support

Adding the turntable and cuing has involved making a number of changes to the composition hierarchy of the desk application. As shown in figure (3.23), the hierarchy has been changed at two different levels.
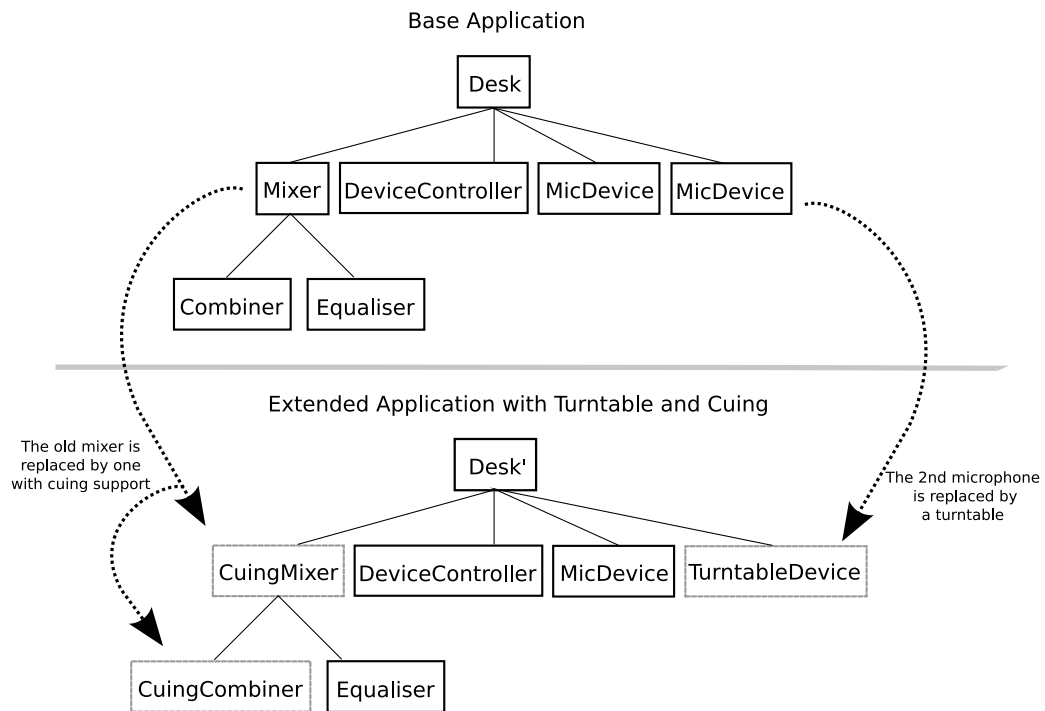
Figure 3.23: Changing the composition hierarchy for cuing and turntable support

The structural alterations are accommodated by the resemblance and replacement constructs, which allow deep change by permitting components at successive abstraction levels to be restructured. These constructs allows the hierarchy to be remade, whilst still permitting a convenient and intuitive graphical component representation in the spirit of UML2 composite structure diagrams.

Further, we have been able to make these changes even though the type of extension was unanticipated by the base application.

3.3.6 The Upgraded Desk

Audiosoft decides to upgrade the mixer component of the original desk, and package it in the `upgrade` stratum. They have recently purchased a superior mixer implementation with integrated combiner and equalisation capabilities. As they do not have the source code, they must evolve the mixer from a composite into a leaf component. It is not uncommon for an extension to either expand (due to further decomposition) or occasionally compress the compositional hierarchy of an application when making alterations.

Audiosoft also wish to change the name of the evolved mixer to `IntegratedMixer`. This is accomplished by the usual dual application of resemblance and replacement, as shown in figure 3.24. The name change is expressed by a delta alteration to the name constituent, and does not affect the logical identity of the component.
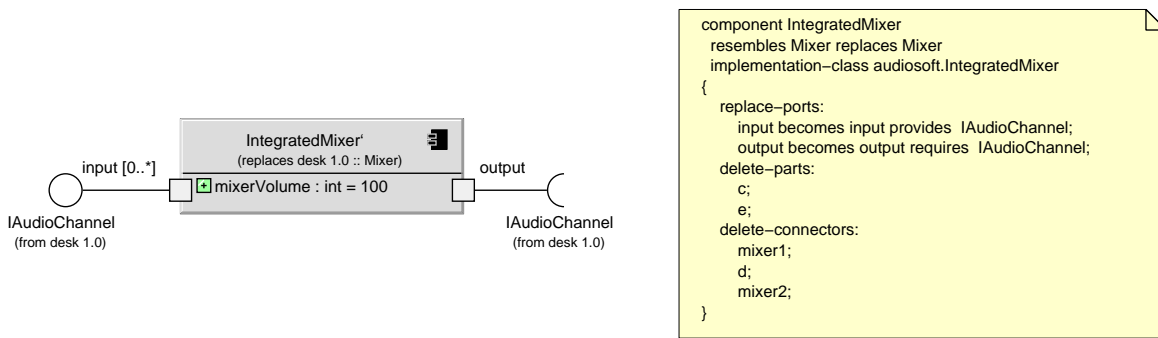
Figure 3.24: Evolving the mixer component from a composite to a leaf

The upgrade also features an evolved `DeviceController`, as shown in figure (3.25). Previously the resolution of the volume controls was found to be too coarse, and the `increaseResolution` boolean attribute was added to indicate to the new implementation that the resolution should be increased. By default, this is set to `true` so that existing parts of this type will automatically pick this up, although an extension can turn this off if necessary by using a slot to set the value to `false`.
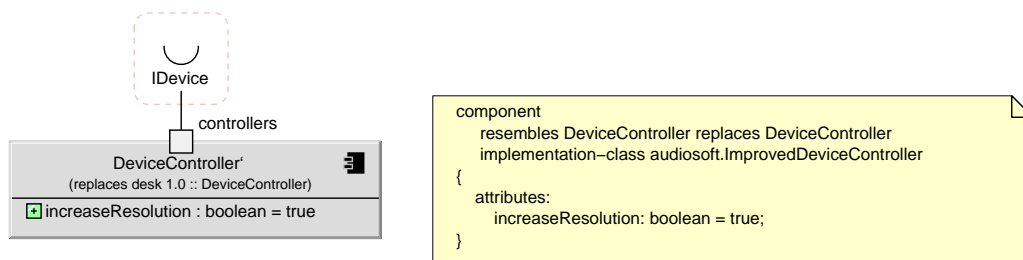
Figure 3.25: Evolving the device controller to specify a new implementation

The evolution of the controller specifies a different implementation class to the previous incarnation. However, it is not always the case that a new class must be used. If no implementation class is specified in the evolution, then the inherited one will be reused. This is only useful if the implementation source is available to the party defining the upgrade, as it implies that the source has been evolved also.

As we are now using an integrated mixer and no longer require the `Combiner` or `Equaliser` components, these can be retired. After this point, it is an error to continue to refer to these components. Figure 3.26 shows that retirement is expressed in terms of resemblance and replacement, in keeping with its status as a secondary construct.
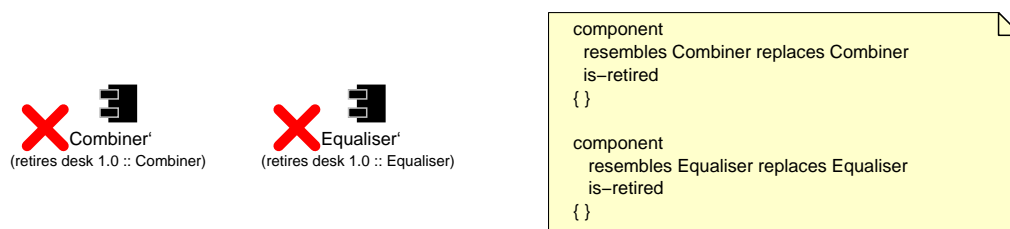
Figure 3.26: Obsolete components can be retired

3.3.7 Combining All Extensions to Form a Coherent Application

Radio station R now wishes to combine the upgrade, the CD player extension, and the turntable extension into a single coherent application. They retain the services of an extension developer to achieve this task for them.

According to the relationships between the parties, as shown in figure (3.10), each extension (including the upgrade) has been developed in isolation, with no view of the others. Therefore there is a possibility that the alterations made to the common base by the extensions may conflict when combined. This is not particularly surprising – by allowing extensions the freedom to radically alter a base application, we also provide the potential for them to make changes that conflict with each other.

In practice, Backbone mitigates against the possibility of conflict by supporting an architectural hierarchy with fine-grained leaf components. This allows an alteration to be specified at the correct level of abstraction (APPROPRIATELEVEL). Further, fine-grained decomposition means that an alteration can be targeted to a precise area in order to achieve the required architectural effect. These factors decrease the footprint of alterations, and hence reduce the chance of structural overlap and conflict.

Stratum Perspective

Stratum perspective is the view of the architecture from a particular stratum, including all of the changes made in that stratum and the strata it transitively depends upon.

Consider the view of the `Desk` component from the perspective of R's `combined` stratum. The dependencies of this stratum causes all extensions and upgrades to be combined, which automatically produces the consolidated component shown in figure 3.27. This component has four erroneous connectors, which are flagged up by the Backbone error checker: the connectors either do not propagate any interface types (connector `desk1`), or the provided and required interfaces of one of the connector ends cannot be made to match the other end.

The connectors errors are not the root cause of the conflicts – they are just a symptom of a deeper structural problem which we will now examine.

Combining Extensions

Consider how the consolidated `Desk` component is formed from the perspective of `combined`. Strata dependencies are used to order the definitions and evolutions from the various extensions, so as to form a consolidated (or expanded) resemblance graph, as shown in figure 3.28. The evolutions from the `turntable` and `CD` strata are in parallel, because the two strata are independent of each other. To form a final view of the component, we apply the deltas starting from the bottom of the graph, working up to the top. If separate branches of the graph conflict, for instance by replacing a constituent with one that the other branch finds objectionable or by deleting a constituent that the other side still depends upon, then we have a structural error which will be picked up by well-formedness rules.
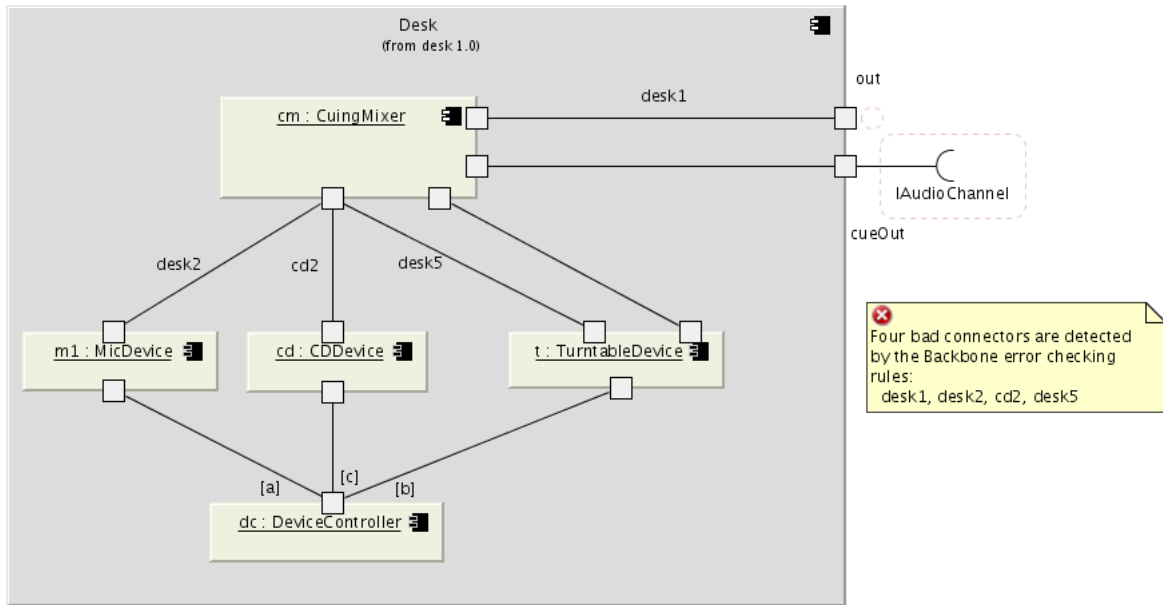
Figure 3.27: The consolidated Desk component

As a result of combining deltas, the consolidated `Desk` contains the `CuingMixer` and `Turntable` part replacements from the `turntable` stratum, and the added `CDDevice` part from the `CD` stratum. However, this has not resulted in a structural conflict for this component. The bad connectors are caused by a structural problem deeper down in the composition hierarchy at the mixer level.
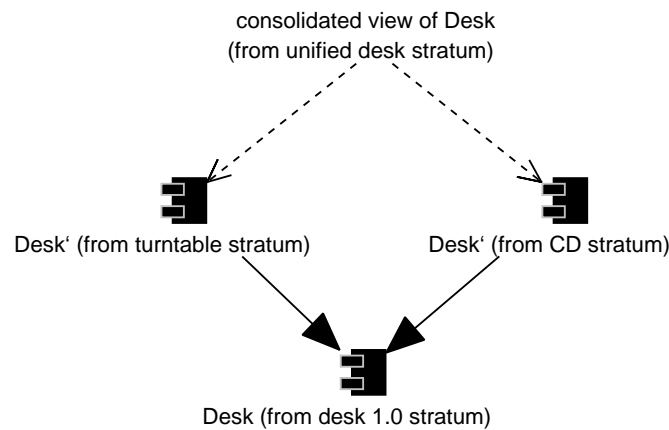
Figure 3.28: The expanded resemblance graph for Desk

Figure 3.29 shows this expanded resemblance graph for `CuingMixer`. Note that in addition to using strata dependencies, any evolutions are inserted before standard resemblance, often leading to a linear graph. The intuition behind this is that `CuingMixer` builds on top of the `Mixer` definition, but the `Mixer` component was subsequently evolved to `IntegratedMixer'`. As such, `CuingMixer` resembles `IntegratedMixer'` in a consolidated view.

These use of strata dependencies for ordering expanded resemblance graphs tends to linearise them in an intuitive manner. Parallel branches form when have multiple parallel evolutions of the same
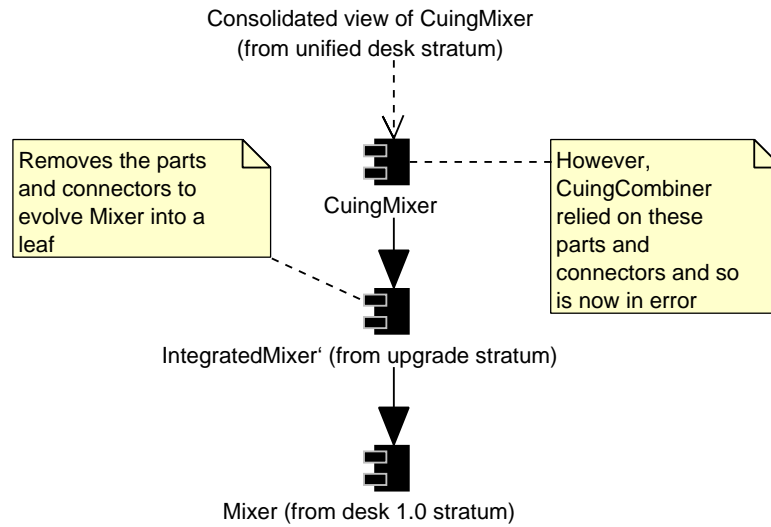
Figure 3.29: The expanded resemblance graph for CuingMixer

component in independent strata, as per figure 3.28.

Upon examining the consolidated `CuingMixer` resemblance hierarchy, the problem becomes clear: the evolution of `Mixer` into the `IntegratedMixer'` leaf has removed the `Combiner` and `Equaliser` parts that `CuingMixer` previously relied on. As such, the consolidated structure of `CuingMixer` (figure 3.30) has a number of structural errors such as missing connectors and a missing `Equaliser` part. A further problem is that the upgrade stratum retired the `Combiner` component: `CuingCombiner` resembles this, and inherits the retirement also.
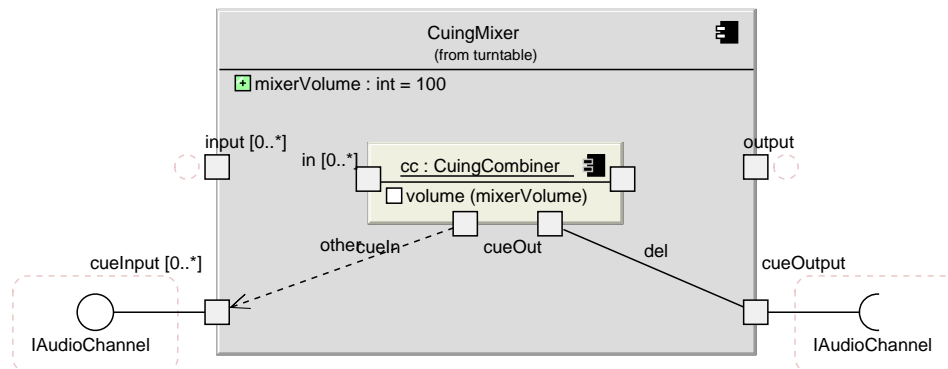
Figure 3.30: The consolidated CuingMixer component has errors

Conflict Resolution

To resolve these issues, R needs to restructure either the `CuingMixer` to cope with the newly evolved `IntegratedMixer'`, or needs to find another way to add cue support. As the evolved mixer has superior performance to the previous incarnation, R decides to retire `CuingMixer` and restructure the `Desk` to include another mixer to act as the cue audio bus. The evolution of `Desk` that R uses

to correct the conflicts is shown in figure 3.31. This involves adding the extra `IntegratedMixer` part, replacing the existing `CuingMixer` part with another `IntegratedMixer`, and redirecting two connectors. Although not shown, we are now also free to retire `CuingMixer`.

The final compositional hierarchy of the consolidated application is shown in the lower part of figure 3.2. The extension developers have together succeeded in transforming the base application to accommodate new and modified features, even though many of the changes were not anticipated by the base architecture.
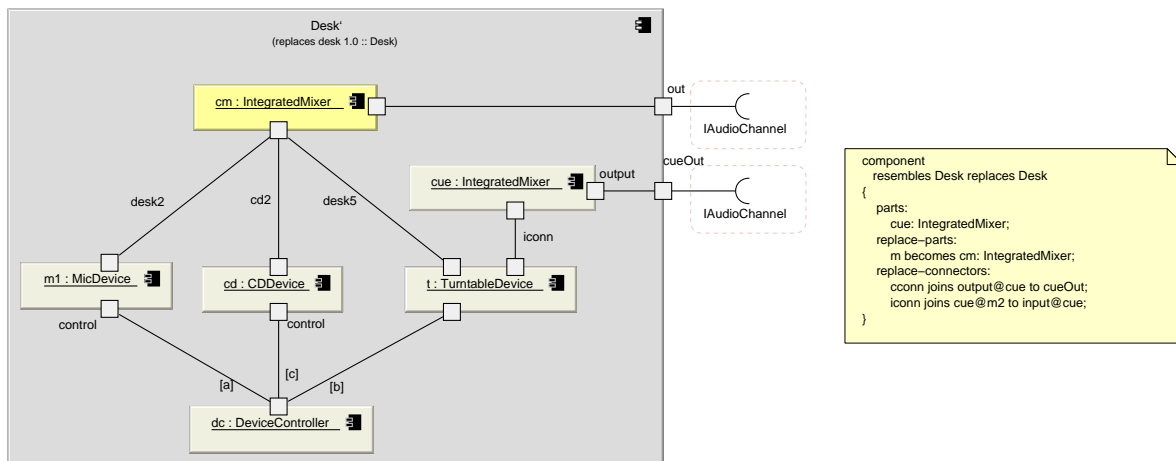
Figure 3.31: Evolving the desk to correct extension conflicts

3.4   Summary

The Backbone ADL consists of a conventional, UML2-based ADL, supplemented with three extensibility concepts: resemblance, replacement and stratum. Resemblance allows an element to be defined in terms of deltas from existing elements thereby facilitating reuse. Replacement allows a newly defined element to be globally substituted for an existing one. Combined together, resemblance and replacement enable an extension to make incremental alterations to the compositional hierarchy of a base application.

A stratum is a nested module, which allows definitions for an extension to be grouped and coarse-grained dependencies to be expressed. Strata form the unit of ownership and architectural control in Backbone, and strata combinations can be used to represent variants of the application.

The audio desk scenario demonstrates how these concepts enable complex extensions, regardless of whether or not the types of extensions were foreseen and therefore factored into the underlying base application. This characteristic allows the base architecture to be designed without the pollution of speculative extension points. In the natural process of architectural elaboration and decomposition into more granular components over time, extensibility is built into an architecture. As such, Backbone copes well with unplanned extension.

By allowing any part of the base to be restructured, Backbone meets the ALTER requirement. It satisfies the NoImpact requirement by keeping evolutions and replacements in separate strata from the components that they operate on. If one party wishes to see the change, then they can choose both the extension and base strata. If another party wishes to see only the base, then they can simply ignore the extension strata.

Backbone satisfies the NoSource requirement, as it enables full extensibility without requiring the implementation source code of components or interfaces. The support of an architectural hierarchy encourages fine-grained component decomposition, which reduces the reimplementation cost if it is necessary to fully replace a leaf in an extension when the source code is not available. Backbone also supports the evolution of leaf implementations via direct source code modification if required.

To form a consolidated component out of multiple independent evolutions, Backbone forms an expanded resemblance graph, where replacements are inserted into the original resemblance graph using strata dependency order. This allows the replacements from various extensions to be combined, meeting the Combine requirement. An upgrade can also be phrased as an extension and combined with other extensions and upgrades, and hence Backbone also meets the Upgrade requirement.

Backbone defines a number of well-formedness checks (section A.4) which can detect if a combination of extensions has resulted in a structural error. Errors can occur when combining independently developed extensions which both alter a common base, and the same extensibility constructs can be used to evolve, replace and retire elements to form a structurally correct system. This meets many facets of the DetectAndCorrect requirement. Behavioural checks are not currently considered and are discussed in future work.

By allowing an architectural hierarchy, Backbone permits extension to occur at the appropriate level of abstraction. It therefore satisfies the AppropriateLevel requirement. The use of a hierarchy also tends to diminish the likelihood of conflict, as extensions can target and minimise the changes they need to make to the base architecture.

In contrast to other approaches such as MAE [RVDHMRM04] which integrate general architectural concepts into a CM system, Backbone introduces extensibility constructs directly into an ADL. This increased modelling support provides powerful facilities for extension, but the constructs are also useful for component evolution and reuse [MMK06]. Backbone is agnostic as to the use of CM approaches and can work with existing CM systems if required.

Requirement	How the extensibility concepts meet the requirement
ALTER	Resemblance and replacement allow any part of a base application's architecture to be restructured for an extension.
NOSOURCE	The implementation source code of components or interfaces is not required to effect an extension. The extension is expressed as structural deltas against the architectural description of the base.
NOIMPACT	An extension is packaged into a different stratum from the base. Only parties who wish to see the extension need reference the extension stratum. Other parties are unaffected.
COMBINE	Replacements from various extensions can be combined, by forming expanded resemblance graphs based on the strata dependency graph. This gives a precise order of delta application.
UPGRADE	An upgrade can be phrased as an extension.
DETECTANDCORRECT	Well-formedness checks are used to detect if a combination of extensions has resulted in a structural error. Any errors can be corrected by another stratum.
APPROPRIATELEVEL	Components form an architectural hierarchy. Resemblance and replacement allow the appropriate abstraction level in this hierarchy to be adjusted by an extension.

Table 3.5: Checking the extensibility concepts against the requirements

Chapter 4

An Outline of the Backbone Formal Specification

This chapter presents an outline of the Backbone formal specification (appendices B and C), focusing on how an extension is able to use replacement and resemblance to alter the structure of an architecture. It also describes how structural conflict occurs between extensions and how this can be resolved in a further extension.

It is worth making the point that the specification does not have to be fully understood (or even considered) in order to make use of the Backbone approach and supporting tools. An understanding of the formal model is helpful, however, for advanced use and tool development.

The need for a formal specification first became clear after an early implementation of the Backbone toolset exhibited a number of unanticipated corner cases and inconsistencies. To resolve these issues, and allow the checking of expected properties, a rigorous specification was constructed in the Alloy logic language. The specification describes the Backbone component model at a manageable level of abstraction, and indicates precisely how resemblance and replacement interact to allow alterations to a base architecture.

The specification also revealed that the resemblance and replacement constructs, which were previously used only for components, were more generally applicable than previously thought. Resemblance and replacement have been subsequently used to express the substitutability and evolution of interfaces. Whereas a component has obvious structure which can be modelled as deltas, an interface is structured as implementation and operation deltas.

A key reason for using Alloy for the specification was its ability to generate counterexamples and witnesses automatically by checking assertions and predicates on a logical model. This encouraged an iterative approach to the construction of the specification, revealing the generality of the constructs.

The Backbone modelling tool and runtime implementations have been completely rewritten using the specification, as described in chapter 5. This has been a productive endeavour resulting in a system

67

with the expected properties. The corner cases that proved so problematic in the previous incarnations have either been eliminated or have been found to be handled correctly by the new implementation.

4.1   Organisation of the Formal Specification

The specification is divided into base and ADL layers, with each layer consisting of multiple Alloy modules as shown in figure 4.1. The dotted lines between the boxes show module dependencies. The base layer specification is listed in appendix B, and appendix C contains the ADL layer listings. Appendix A explains important parts of the specification in more detail, and describes the structural rules and well-formedness checks.
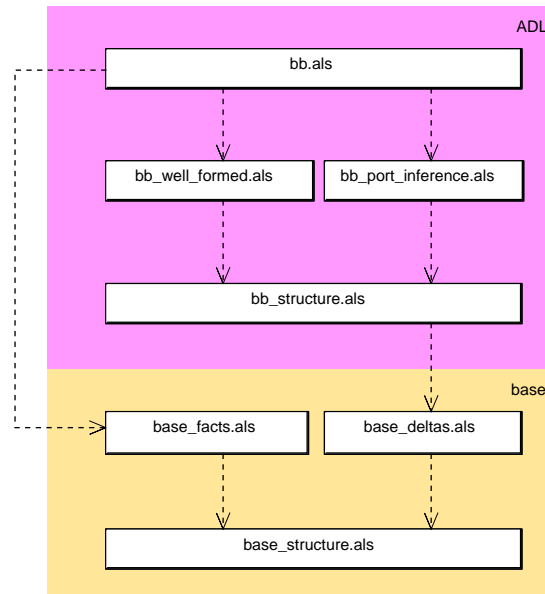
Figure 4.1: The module structure of the Backbone formal specification

The base layer describes the `Stratum`, `Element` and `Deltas` signatures, and introduces the resemblance and replacement relationships. These are general concepts which can be used to describe alterations to the compositional structure of elements, resulting from extensions.

The ADL layer builds on this foundation to describe the Backbone component model. `Component`, `Interface` and `PrimitiveType` are defined as subsignatures of `Element`. The structural rules which describe a well-formed model are contained in `bb_well_formed.als`. Port type inference is described in `bb_port_inference.als`, and general facts about the model are described in `bb.als`.

This chapter focuses primarily on imparting an intuitive understanding of the concepts in the (much smaller) base layer, also describing how the ADL layer is specified using the base concepts.

4.2 Key Concepts of the Base Layer

An *element* is a structural entity composed of *constituents.* It can *resemble* other elements of the same type, thereby inheriting their structure. It can have a possible *replacement* relationship to another element, globally replacing it in the architecture.

An element can contain a number of *delta* alterations, each of which can *add*, *delete* or *replace* a constituent in order to modify the inherited structure.

A *stratum* is a module construct that can be used to group the definitions of an extension or base application. Strata can have *dependencies* on other strata. Each element is owned by a single *stratum* which is known as its *home*. The dependencies of an element's home stratum determine what that element has visibility of in the system. Strata dependencies further govern the order in which replacements are applied to form a complete architecture.

Elements are described by the `Element` signature, strata by the `Stratum` signature, and deltas by the `Deltas` parametrised signature. The form of constituents is not prescribed by an Alloy signature – instead a constituent type is represented by a type parameter to `Deltas`. The resemblance and replacement relationships are represented as fields in the `Element` signature.

An example showing how replacement relationships and strata dependencies affect resemblance graphs is shown in the desk scenario of section 3.3.7.

4.2.1 Stratum and Dependency Rules

A stratum is defined by the partial signature below. It can contain nested strata (`nestedStrata`) which are collectively known as its children, and therefore each stratum has a single possible parent (`parent`). A stratum must explicitly declare any dependencies on any needed non-nested strata (`dependsOn`). It can access any definitions in nested strata without requiring explicit dependencies.

The strata dependency rules control what a stratum's elements can see for their resemblance and replacement relationships and constituents. Informally, an element can access definitions in its home stratum, any child strata of the home, and also any strata that the home or its parents depend on.

```
sig Stratum
{
  parent: lone Stratum,
  dependsOn: set Stratum,
  dependsOnNested: set Stratum,
  nestedStrata: set Stratum,
  isRelaxed: Bool,
  ownedElements: set Element,
    -- derived state
  exportsStrata: set Stratum,
  canSee: set Stratum,
```

```
  simpleDependsOn: set Stratum,
  isTop: Bool,
  transitive: set Stratum,
  replacing: set Element,
...
}
```

Listing 4.1: base_structure.als, lines 14-49

The Formal Dependency Rules

An element can access definitions in its home stratum, or any strata that its home stratum can see (canSee) by its dependencies. A strata can see any nested strata, any strata exported to it by those it depends upon, and any strata that its parents depend upon. The full set of reachable strata is held in transitive.

```
let fullDependsOn = dependsOn + nestedStrata + ^parent.dependsOn {
  ...
  s.canSee = s.fullDependsOn.exportsStrata
  s.transitive = s.^fullDependsOn
```

Listing 4.2: base_facts.als, lines 11-40

A stratum exports its dependencies (exportsStrata) to strata that depend on it. This allows a stratum to selectively export nested strata that form part of its public interface. A relaxed stratum (isRelaxed) exports itself, all dependencies exported to it, and any explicitly indicated nested strata (dependsOnNested). A strict stratum exports only itself and any explicitly indicated nested strata. The dependsOnNested field is only needed for exporting nested child dependencies, as parents implicitly depend on their children and therefore do not have to express dependence otherwise.

```
isTrue[s.isRelaxed] =>
  s.exportsStrata = s + s.(dependsOn + dependsOnNested).exportsStrata
else
  s.exportsStrata = s + s.dependsOnNested.exportsStrata
```

Listing 4.3: base_facts.als, lines 20-23

We remove any redundancy from the dependencies and hold this in simpleDependsOn. We further rule out cycles, forcing the dependency structure to be a graph.

```
s.simpleDependsOn = s.fullDependsOn - s.fullDependsOn.transitive
-- no cycles allowed
s not in s.transitive
```

Listing 4.4: base_facts.als, lines 27-31

Although seemingly complex, the rules are intuitive when considered in the context of an example (section 4.2.2). The rules were distilled from architectural principles and practices where implementation strata must be hidden, and strata representing public definitions selectively exported. These facilities are useful when organising the architecture of a large system.

The rules also prevent a nested stratum from accessing the definitions in its transitive parents. This allows us to extract an extension stratum from one model and import it into another, even if that model does not contain the parents. This allows extensions to be shared between parties in an extension setting.

Independently Developed Strata

Two strata are mutually independent, and hence cannot access each others' definitions, if neither can reach the other through their full transitive dependencies. This is described by the predicate `independent` below.

Independent strata share a common base if their transitive dependencies overlap in some way. This is described by the `independentOnCommonBase` predicate. This allows us to model the situation as shown in section 3.3.1, where two extensions are *independently developed* by unrelated parties (X and Y) on top of a common base application. X and Y each cannot access the definitions of the other, but can cause conflict by altering the base in a way which results in problems for the other party.

```
pred independent[a, b: Stratum] {
  a not in b + b.transitive and b not in a + a.transitive
}
pred independentOnCommonBase[a, b: Stratum] {
  independent[a, b] and some a.transitive & b.transitive
}
```

Listing 4.5: base_facts.als, lines 54-59

The *perspective* of stratum `s` consists of the system after we have applied all of the definitions (including replacements) contained within `s + s.transitive`, starting with strata lower down in the graph structure first.

A stratum has `isTop` set to true if no other strata depend on it. This is usually the *top* stratum in a system as per `combined` in figure 3.11.

4.2.2 A Strata Dependency Example

Figure 4.2 shows two extension strata (`extension1`, `extension2`) building on the common `base` stratum. The `extension2` stratum contains two nested strata: `api` to hold the public definitions and `implementation` to hold hidden definitions. The `unified` stratum is the top stratum of the system, and combines the extensions.

Consider the exports (`exportsStrata`) of each stratum:

- `extension1` exports `[extension1, base]`

- `extension2` exports `[extension2, api, base]`

- `api` is strict and exports only `[api]`, although it can see `implementation`

The transitive dependencies of `unified` are `[extension1, extension2, api, implementation, base]`. However, `unified` cannot see any definitions in `implementation`, as this is not visible to it. Strata `direct` shows that it is possible to access a buried stratum by directly depending on it. This type of deep access is often required in extensible systems, where otherwise hidden implementations need to be altered for an extension.
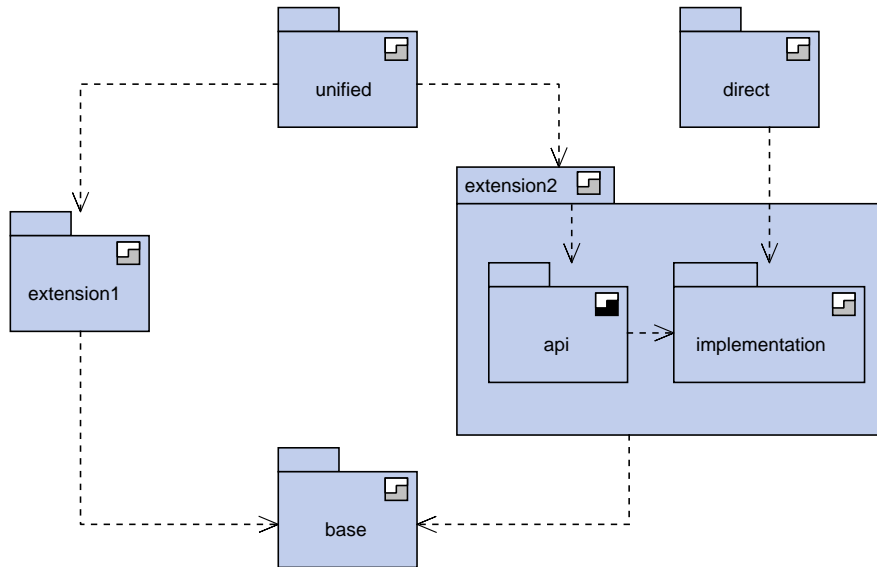
Figure 4.2: Strata dependencies

The dependency rules describe how to form an equivalent non-nested arrangement as shown in figure 4.3. This shows that `extension2`, `api`, `implementation` and `direct` are mutually independent of `extension1`. As such, to form the `unified` perspective, we must apply any replacements in the following strata order. Note that `direct` does not feature in `unified`'s stratum perspective.

1. `base` first

2. `extension1` in parallel with `[implementation` followed by `api`, followed by `extension2]`

3. `unified` last

There is a possibility of structural conflict when combining independent extensions that both alter the common base. Consider that `extension1` and `extension2`, each developed independently, are error free when combined on their own with `base`. When combining both extensions, we can get
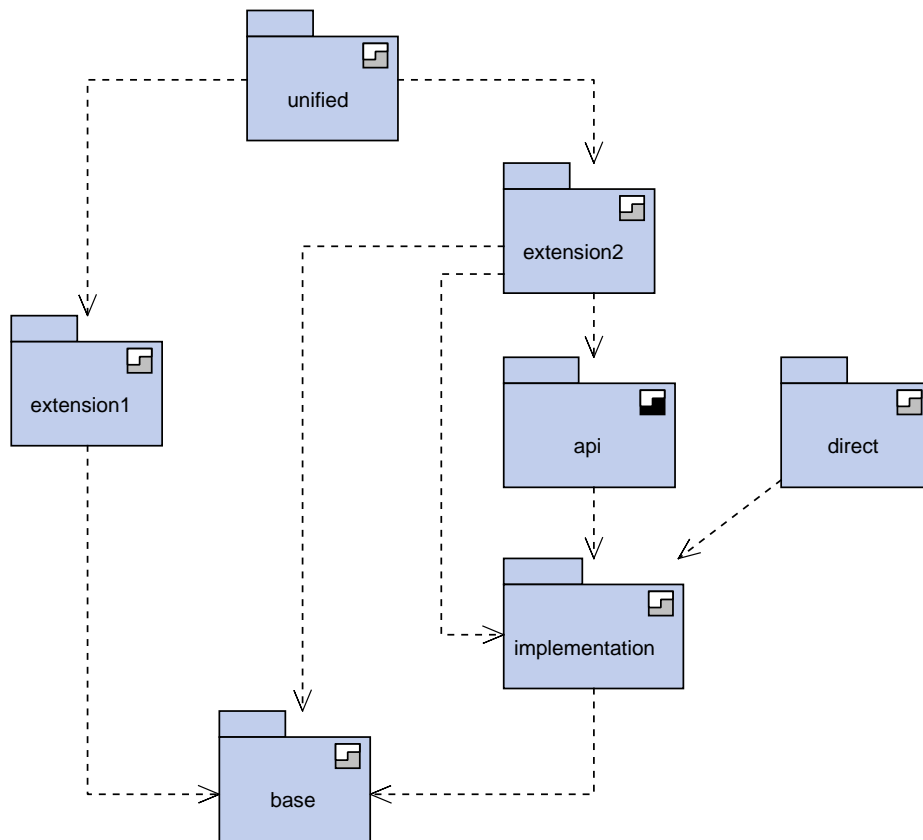
Figure 4.3: Nested strata can be flattened

conflicts if replacements from `extension1` change the base in a way which causes structural errors for `extension2` and vice versa. The structural rules that detect these errors are described in section A.4.

4.2.3 Elements

An `Element` is a structural entity that can participate in resemblance and replacement relationships. It is the common base signature for components, interfaces and primitive types in the ADL layer.

An element is composed of constituents, which are added, deleted or replaced by deltas (section 4.2.4). An element can resemble one or more elements of the same type (`resembles`), which means that it inherits their structure, which is then altered by any deltas. Each element may further use replacement to globally replace at most one other element of the same type (`replaces`) which cannot be defined in the same stratum. Each element is owned by a single stratum, called its home (`home`).

Element replacement, and using a delta to replace an inherited constituent are two different concepts. When we need to differentiate between these two types of replace, we use the term replacement which refers to the former.

---

abstract sig Element
{

```
  home: Stratum,
  replaces: lone Element,
  resembles: set Element,
-- derived state
  resembles_e: Element -> Stratum,
  actsAs_e: Element -> Stratum,
  isInvalid_e: set Stratum
}
```

Listing 4.6: base_structure.als, lines 61-73

The Expanded Resemblance Graph

The resemblance graph of an element can change with each stratum perspective, as replacements are applied using the strata dependency order graph as a guide. For each element, we form an *expanded resemblance* graph per stratum by inserting any replaced definitions into the existing resemblance graph. The expanded graph for each perspective is held in `resembles_e`.

Consider how this works in figure 4.4. Component `A` resembles `B` in the `base` stratum as shown, and the expanded graph for this perspective is (`base::A` resembles `base::B`). The evolution of `B` introduced by `extension1` causes the expanded graph from perspective `extension1` to become (`base::A` resembles `extension1::B'` resembles `base::B`). From the perspective of `extension2`, the expanded graph is (`base::A` resembles `extension2::B'` resembles `base::B`).
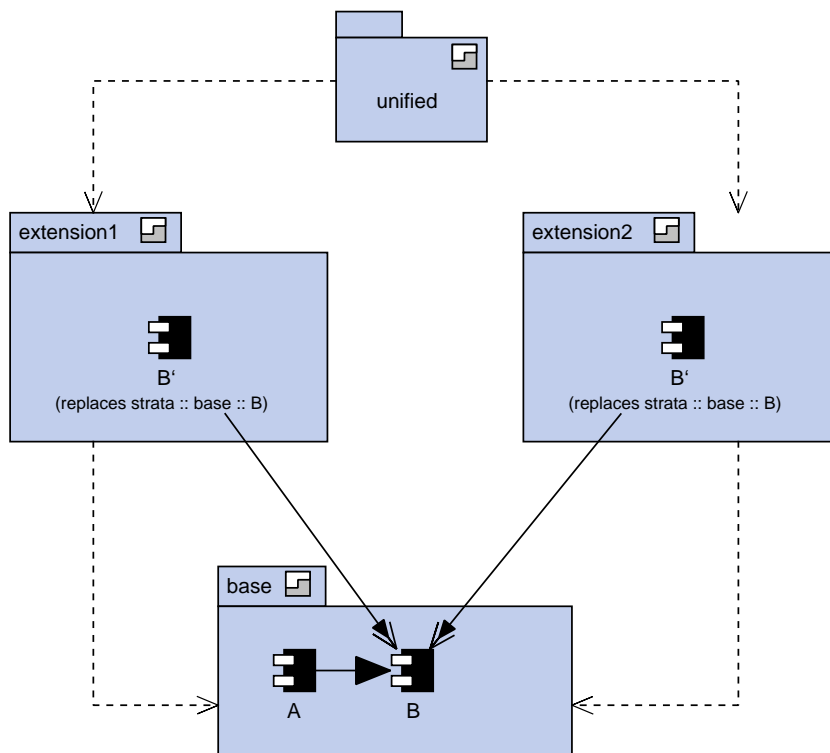
Figure 4.4: Replacements adjust the resemblance graph for each stratum perspective

From the `unified` perspective, we get the diamond-shaped graph as shown in figure 4.5:

(`base::A` resembles (`extension1::B'`, `extension2::B'`) resembles `base::B`)
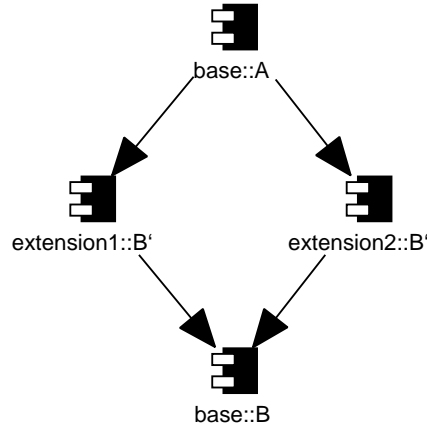
Figure 4.5: The expanded resemblance graph of A from the unified perspective

Consider that `extension1::B'` may change the inherited structure in ways which `extension2::B'` finds unacceptable. For example, if `base::B` has a part `X` and connectors in `extension2::B'` bind to its ports, and if `extension1::B'` replaces `X` with part `Y` which does not offer the same ports, then we will have structural errors when combining the extensions to form `A`. We call this a *structural conflict*.

Simple plugin systems [Vol99, MMS03] avoid the possibility of structural conflict by disallowing the equivalent of replacement. This violates the ALTER requirement, however, as the underlying base cannot be remade unless replacement is allowed. This reveals a trade-off between preventing conflict and allowing possibly unplanned alterations.

4.2.4 Deltas

Each instance of `Deltas` hold a collection of adds, deletes and replaces for constituents of an element, allowing alterations to the inherited structure to be expressed. Each component definition contains separate instances of `Deltas` for expressing port, part, connector, attribute and port link alterations or additions. Components which have no inherited structure are made up of added constituents via deltas to establish the structure.

For each stratum perspective, an element can be fully expanded by applying all the deltas in the expanded resemblance graph to determine the final structure.

```
sig Deltas
{
  newObjects:           set Object,
  addedObjects:         set newObjects,
```

```
  replacedObjects:          set Object,
  newIDs:                   set ID,
-- the adds, deletes and replaces
  addObjects:               newIDs one -> one addedObjects,
  deleteObjects:            set ID,
  replaceObjects:           ID one -> lone replacedObjects,
-- the expanded constituents, for each stratum perspective
  objects_e: Stratum -> ID -> Object,
...
}
{
  no dom[replaceObjects] & deleteObjects
  replacedObjects = newObjects - addedObjects
}
```

Listing 4.7: base_deltas.als, lines 5-41

The key fields of `Deltas` are `addObjects`, `deleteObjects` and `replaceObjects`: `addObjects`
are the constituents that the definition adds, `deleteObjects` identifies any inherited constituents
that are to be deleted, and `replaceObjects` holds new constituents to replace existing inherited
ones. In all cases, inherited constituents to be deleted or replaced are identified by their universally
unique identifier. `ID` represents this identifier, and models UUIDs in the formal specification.

`Deltas` are defined by an Alloy module, parametrised by constituent type (`Object`) and UUID type
(`ID`). For instance, the `Deltas` used to represent component parts is parametrised by substituting
`Part` for `Object` and `PartID` for `ID`.

Consider now how to apply the deltas to form an expanded component, by building on the previous
example as shown in figure 4.6. The `base` definition of `B` adds `part1` and `part2`. A resembles `B`
and adds `part3` and replaces the inherited `part1` with `part4`. The evolution of `B` in `extension1`
replaces `part1` with `part5`, and `part2` with `part6`. The evolution of `B` in `extension2` deletes
`part1` and replaces `part2` with `part7`.

The expanded resemblance graph of `A`, from the perspective of `unified`, is shown in figure 4.5.
The deltas are applied in the following order, reflecting the strata dependency order and the original
resemblance graphs.

- `part1` and `part2` are added (base)

- `part1` is replaced with `part5` and `part2` is replaced with `part6` (extension1::B')
  -in parallel with–
  `part1` being deleted and `part2` being replaced by `part7` (extension2::B')

- `part3` is added and `part1` is replaced with `part4` (base::A)

The second step involves both a replace and a delete of `part1`. When this occurs, Backbone ignores
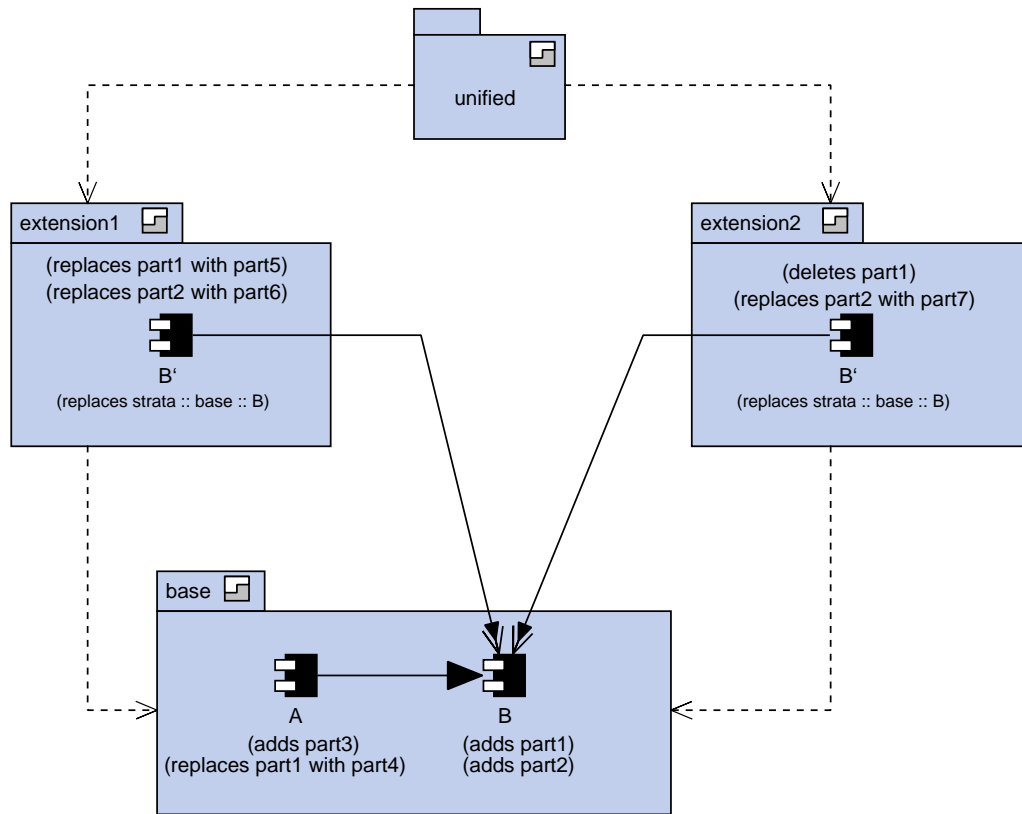
Figure 4.6: Deltas are applied to form a fully expanded element

the deletion. We also see that `A`'s replacement of `part1` with `part4` supersedes the replacement of `part1` with `part5` in `extension1`.

The final component `A` from the `unified` perspective has the following mapping of `UUID` to constituent.

- `UUID (part1)` → `part4`

- `UUID(part2)` → `part6, part7` (conflict)

- `UUID(part3)` → `part3`

Note that we have two constituents for `UUID(part2)`, reflecting that both `extension1::B'` and `extension2::B'` replace the same constituent. This is a structural conflict[1]. To resolve this situation, a definitive replace is required in another evolution higher up in the resemblance graph. For example, we could create `unified::B'` and replace `part2` with `part8`, which would supersede the conflicting constituent replaces, resolving the problem.

Even when replacing another, a constituent retains the `UUID` of the original constituent, as shown for `part4` which retains `UUID(part1)`. This allows us to combine independent strata, where each only knows about the UUIDs of the base elements and constituents.

---

[1]This conflict is detected by rule WF_COMPONENT_PART_PER_UUID described in section A.4.

The final expanded component `B` from the unified perspective, has the following makeup.

- `UUID(part1)` → `part5`

- `UUID(part2)` → `part6, part7`

The expanded deltas are pushed back into `B`'s `object_e` field for the `unified` perspective. This means that existing references to `B` from other components do not have to be redirected to an evolution when replacement occurs, and we can instead pick up the full structure from the definition of `B` itself. An earlier version of the specification used redirection, where a replacement redirected existing component references to itself, but this led to a cumbersome specification which further introduced an overhead into the graphical modelling tool. The current approach relies on never directly referencing an evolved component, but instead always referencing the original element, which then holds the full structure from each perspective.

Although small in terms of number of Alloy lines, the `Deltas` logic is reasonably involved. This primarily a consequence of having to rewrite the expanded resemblance graph for each element from each perspective. This logic is explained in detail in appendix A. Although our example focused on part deltas, the same logic applies for any constituent type.

4.3   Key Concepts of the ADL Layer

The ADL layer builds on the base layer to form a full component model, where components, interfaces and primitive types are modelled as subsignatures of `Element`.

A class diagram of the relationships between the main signatures is shown in figure 4.7. This diagram will be further explained in this section.

4.3.1   Components

Components are represented by the `Component` signature, which inherits from the `Element` signature. `Component` has a separate `Deltas` instance for each constituent: parts, attributes, ports, connectors, and port links. As discussed earlier, full component expansion uses the strata dependency order to insert replacements (`replaces`) into the resemblance (`resembles`) graph, and then the alterations from the deltas can be applied to form a complete structure.

The implementation class name (`myCImplementation`) of a component is also phrased as a constituent in order to allow the implementation class to be added, replaced or deleted by a delta alteration. A leaf component must have a single implementation class name and no parts. To turn a leaf component into a composite via evolution, parts must be added and the implementation class name deleted via a delta. Multiple conflicting implementation class names can result when independent evolutions of a leaf are combined, and to resolve the situation a further evolution is required which replaces both names with a new, definitive one.
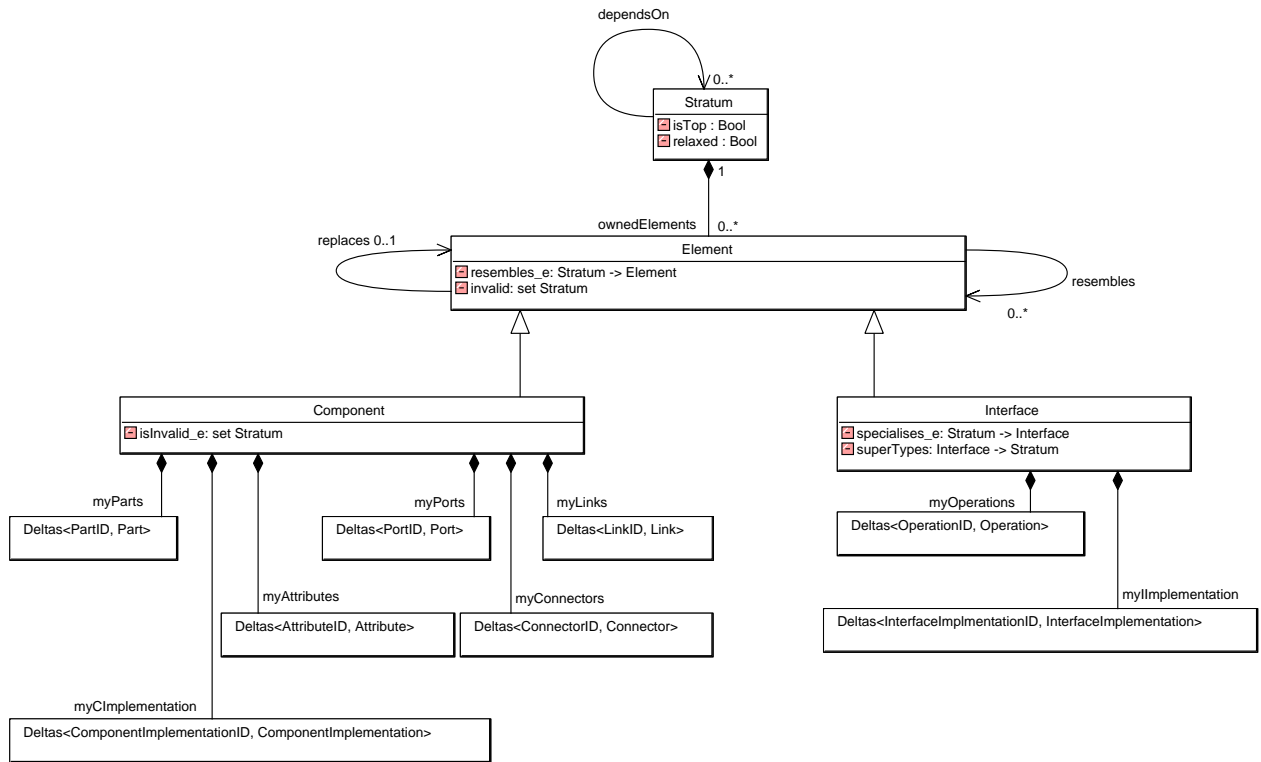
Figure 4.7: The main signatures in the specification

4.3.2 Interfaces

Interfaces are represented by the `Interface` signature, which also inherits from `Element`. The constituents of an interface are operations and implementation interface name.

If an interface resembles another and does not delete or replace any of the inherited operations, then the resembled interface is one of its supertypes. This is described by the following logic.

```
i.superTypes.s =
  { super: i.resembles |
    super.myOperations.objects_e[s] in
      i.myOperations.objects_e[s] }
```

Listing 4.8: bb.als, lines 43-46

Interface Compatibility for Connectors

A provided interface is compatible, for connection purposes, with the same required interface or any of its supertypes. In essence, the rule is "provide at least as much as is required". For a set of provided interfaces and a set of required interfaces, this notion is expressed by the following logic.

```
pred providesEnough(s: Stratum,
    provided: set Interface, required: set Interface)
```

```
{
  all prov: provided |
    one req: required |
       req in prov.*(superTypes.s)
  all req: required |
    one prov: provided |
       req in prov.*(superTypes.s)
}
```

Listing 4.9: bb_inference_help.als, lines 6-16

For two ports to be connected together, there must be a one-to-one, unambiguous mapping between each provided interface of one port and a required interface of the other port (and vice versa). This is captured by the following logic.

```
s -> c not in end.linkError <=>
{
  no end.partInternal =>
    providesEnough[s, infProv, matchingTerminalProvides]
  oneToOneRequiredMappingExists[s, c, infProv, allEnds]
  oneToOneProvidedMappingExists[s, c, infReq, allEnds]
}
```

Listing 4.10: bb_port_inference.als, lines 146-153

4.3.3   Human Readable Names and Renaming

The formal specification does not model human readable names, and instead uses unique identifiers which never change once assigned to an element or constituent. The implementation does need to deal with names (particularly to provide support for the modelling tool) and does so by treating them as another constituent type which can be inherited, deleted or replaced. This neatly handles the renaming of an element in an extension, ensuring that this cannot affect an element's logical identity.

Unlike other constituent types however, no error results if two separate names are introduced for the same element by independent extensions using element replacement. Instead, both names will be displayed against the graphical view of the element, and a further replacement can be used to return back to a single name. This somewhat relaxed approach towards naming conflicts is in keeping with the diminished importance of human readable names in Backbone, given that they do not confer logical identity.

Similarly, constituent replace can be used to evolve the human readable name of a constituent such as an attribute or an operation.

4.4   Areas Not Covered By the Specification

The component model of Backbone is specified at a level of abstraction that allows the implementation concepts to map closely onto their formal counterparts. The implementation does, however, cover several concepts which are not described by the specification. In this section we briefly discuss these areas and indicate how they are handled by the implementation.

Firstly, it is important to reiterate that the primary aim of the specification is to describe how deltas alter component architectures in the presence of replacements from extensions. The specification does not aim to deal with architectural flattening, in either a local or distributed sense. This area has been covered for hierarchical systems in [KME95, MDEK95]. As such, the Backbone formal specification does not cover hyperports (see section 6.2), which are handled in the implementation by adding connectors to the flattened representation based on the previously discussed interface compatibility rules for connectors between ports.

The specification does not deal with the dynamic instantiation of parts via factories (see section 6.3.2). In the implementation, factories are treated like normal components, except that the connections and parts are instantiated on demand rather than at program startup.

Although the specification describes connector indices, it does not mention delegate connectors. These connectors can be used to connect two ports, potentially avoiding the need for many indexed connectors. In the specification a delegate can be modelled by a set of explicit connectors each with explicit indices. In the implementation a delegate connector establishes an alias between two ports, and this notion may be incorporated back into the specification at a later time.

Finally, retirement is not covered. This concept could be easily incorporated into the specification in the future. In the implementation, retirement of an element is represented by a boolean delta. In other words, retirement is handled by evolving an element and setting its retirement flag constituent to true.

4.5   Summary

The formal specification of Backbone is divided into two layers. The base layer describes the general extensibility concepts of stratum, element, deltas, resemblance and replacement. Essentially, this explains how delta changes to the compositional structure of elements can be applied, in the presence of replacements from extensions, to form fully expanded elements. It also describes how structural conflict occurs and how it can be remedied by a further extension. A key point is that replacements are applied in an order governed by strata dependencies.

The ADL layer builds a component model on top of the base layer. The component, interface and primitive type concepts are each modelled as elements, and can therefore participate in resemblance and replacement relationships. The component model is augmented by rules describing a structurally correct architecture. These rules are outlined in appendix A.

The implementation class name (or interface name) of a leaf component (or ADL interface) is also represented as a constituent. This neatly expresses the mapping between architecture and implementation, allowing an extension to adjust this for leaves or interfaces. Although human readable names are not described in the formal specification, these too are implemented as delta constituents. This allows the renaming of elements and constituents by an extension. Names have a diminished status in Backbone as they do not establish or affect logical identity.

The next chapter discusses tool support for Backbone. The specification was initially written with the aim of reducing corner cases and inconsistencies discovered in an earlier implementation of the toolset. We have now fully reimplemented the modelling tool and runtime platform using the formal specification, and this has eliminated the earlier issues. The writing of the specification also clarified many of the extensibility ideas and led to the insight that resemblance and replacement were applicable to more than just component structures.

Chapter 5

Tool Support for Backbone

Backbone is supported by a graphical modelling tool for creating both base applications and extensions. A runtime environment is also provided, which can instantiate and connect up components from a Backbone description. This allows us to execute a Backbone model, assuming we have implemented the leaf components and interfaces described therein.

The tool and runtime implementations are both based on the formal specification. A previous incarnation of the toolset used an informal description of the resemblance and replacement constructs, but this was found to contain many inconsistencies. The reimplementation does not have the errors and corner cases of the previous version, and this experience has demonstrated the value of a precise specification to our modelling approach. See chapter 4 for further details on the motivation behind the formal specification.

In addition to respecting the specification, the modelling tool must also deal gracefully with inconsistent and possibly erroneous models. It is natural when elaborating an architecture in Backbone to proceed through a set of incomplete representations before arriving at the final system. The tools must deal with these transitional models robustly and report on any issues sensibly, whilst still retaining the full benefits of the approach.

Appendix E contains instructions for downloading the Evolve modelling tool, Backbone runtime and the example models used in this thesis.

5.1   The DeltaEngine: Implementing the Specification

The DeltaEngine is the Java library which implements the logic described by the formal specification. It is able to apply deltas to form an expanded view of an element from a given stratum perspective, tolerating possible errors in the model. It also implements port type inference and the well-formedness rules described in appendix A.

5.1.1   Organisation of the DeltaEngine Library

A key requirement is that the same library should be usable for both the modelling tool and the runtime environment. The modelling tool requires a full underlying UML2 repository, which is large and complex. As we do not wish to have this "baggage" also in the runtime, the DeltaEngine library supports various interfaces to allow the storage and manipulation of the definitions to be handled in different ways.

In the modelling tool the component and interface definitions are stored in either a UML2 XMI [Obj09d] fileset or an object database. In the runtime environment, the definitions are stored in XML files conforming to a lightweight Backbone-specific schema. As such, the runtime remains unencumbered by the complex UML2 model and infrastructure, but is still able to use the DeltaEngine library. This organisation is shown by the layer diagram in figure 5.1.
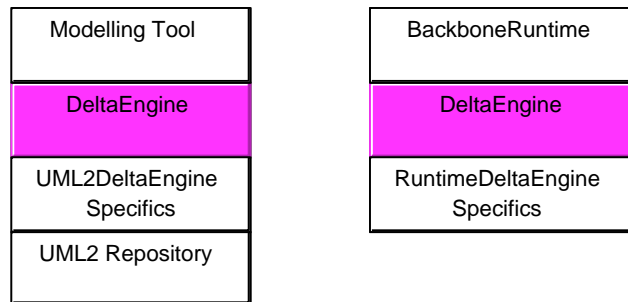
Figure 5.1: The DeltaEngine library is used in both the modelling tool and runtime environment

5.1.2   Handling Incomplete and Incorrect Models

When creating early iterations of a system, it is common to transition through a number of incomplete and inconsistent architectural models. The toolset must cope with these transitional models and report accurately on any errors.

Consider that the Backbone approach relies on applying the deltas of an element using the expanded resemblance graph, as described in section 4.2.3. Any circularity in the strata dependencies or resemblance relationships will cause circularity in these graphs. If we do not deal with this in a robust way, the implementation can go into an infinite loop when expanding the deltas for certain types of models. On the other hand, we wish to show as much as possible of the structure of elements with circularity.

We deal with this by temporarily removing any resemblance to elements which are themselves circular, in the expanded graph. Consider how we might apply this to the resemblance graph of A in figure 5.2(a). In this case, B has circular resemblance, so we temporarily remove the relation from A to B, giving the graph shown in figure 5.2(b). If the situation is remedied at a later point by removing the resemblance from B to A, then the resemblance from A to B can be re-included in the graph for delta application.

(a) with circularity

(b) removing circularity for A
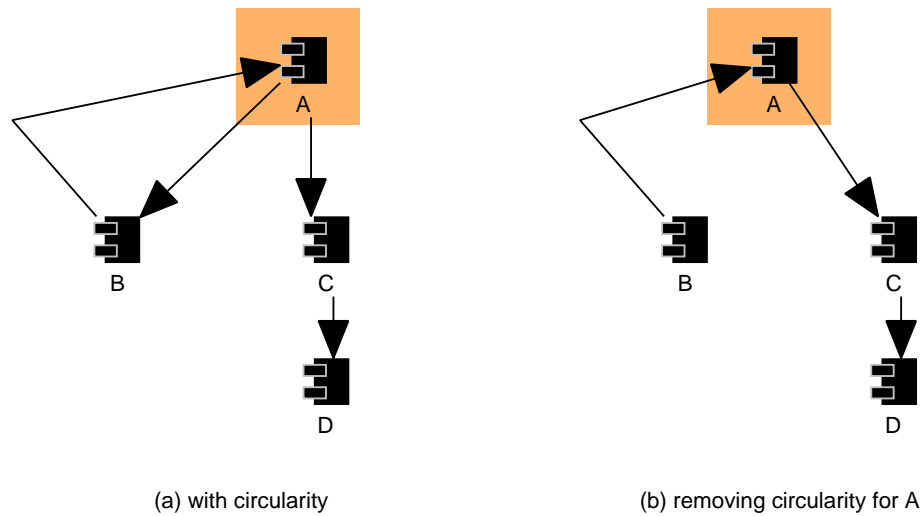
Figure 5.2: Fixing a circular resemblance graph

Inconsistencies Through Direct Editing

Inconsistencies can result when a stratum is directly modified after another stratum has built on top of its definitions. Figure 5.3 shows how `B` from stratum `Y` resembles `A` from stratum `X`, adding a single connector `C`. Consider if we started with this model, and then the owner of `X` subsequently decided to directly edit `A` and turn it into a leaf by physically deleting the part and connector. The connector `C` would now be in error in this scenario because the part it previously connected to has disappeared. The modelling tool must handle such situations gracefully, as there is no guarantee that a stratum's definitions will remain static, or that they will only be altered using the Backbone constructs.
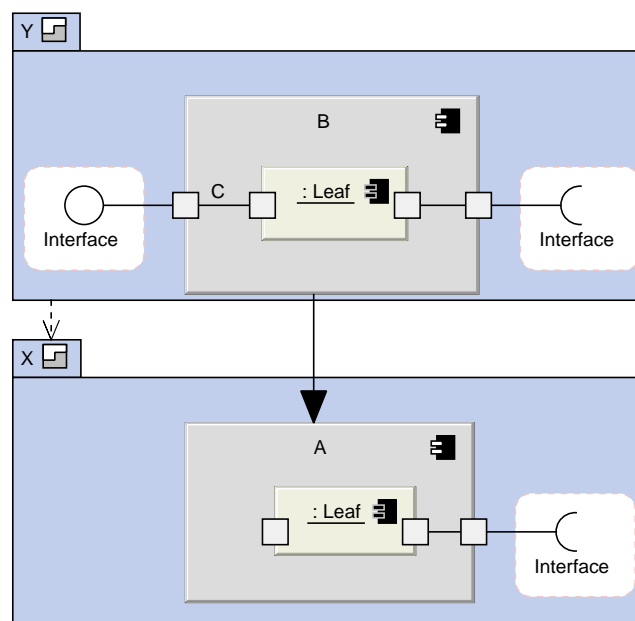
Figure 5.3: Destructive editing can cause inconsistencies

5.1.3   Error Checking

The DeltaEngine includes a set of rules for checking the structural correctness of the definitions in a model. For instance, a component is checked to see that it has some ports, and that if it has parts (and is therefore a composite) that it doesn't specify an implementation class. There are approximately forty such rules, which are described in more detail in appendix A.

Error checking the definitions in a single stratum is straight forward. However, if a stratum contains replacement then it can alter existing definitions in other strata: we need to check the stratum and also any other stratum it transitively depends upon after the replacements have been applied. Checking an entire model in this way implies a combinatorial explosion. We use the absence of replacement within a stratum (i.e. a non-destructive stratum) to prune this graph.

Consider how this works in the model of figure 5.4 where only stratum c contains replacements. To comprehensively check the entire model, we initially check stratum d, then b. Next, we check c, but because it contains replacements, we must also recheck d with the replacements from c applied. Finally we check a, but as it does not contain replacements we can avoid rechecking the other strata. This configuration results in 5 stratum checks for 4 strata. If a and b also contained replacements, we would have to perform 9 stratum checks.
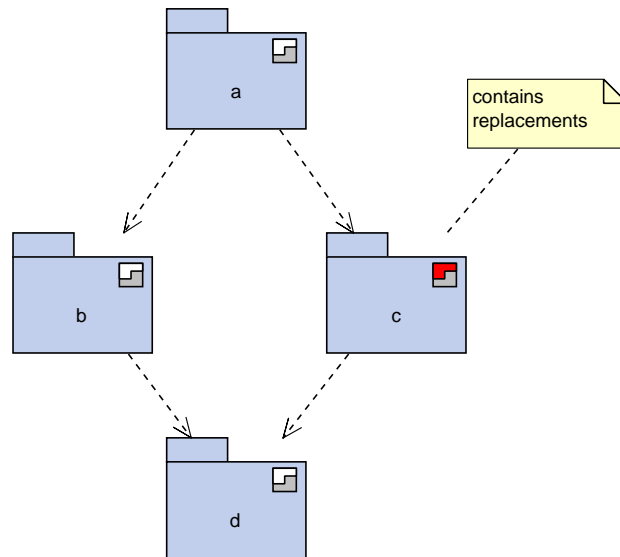
Figure 5.4: Checking strata with replacements

5.1.4   Performance Considerations

Certain operations of the DeltaEngine need to be fast enough to support the interactive graphical manipulations required by the modelling tool. Delta expansion fits into this category, as components are always shown graphically with fully expanded structure. Although this expansion is expensive, particularly as the model and resemblance graphs become large, caching of the results ensure that information is not recomputed unnecessarily. Coupled with the relatively good performance of the

Java runtime, real-time graphical interaction presents no problems even in relatively large models with more than 500 components featuring complex resemblance relationships, where many components have up to 100 or more attributes and ports.

Another operation that needs to be performed quickly to support graphical interaction is port type inference. This represents a potential problem, as the algorithm must consider the full depth of the composition hierarchy when determining the port interface types in the worst case scenario. In practice, two factors mitigate against this being a problem for interactive use. Firstly, the algorithm has been optimised such that even large hierarchies can be used. Secondly, the algorithm does not always reach the bottom of the hierarchy, as the connectors and structure for composite ports may not push it down this far. In practice, the performance of the port type inference has been acceptable for use in a graphical manipulation environment.

5.2   Evolve: The Backbone Graphical Modelling Tool

Evolve is the Backbone modelling tool. It was designed from the ground up to support the extensibility approach in a deep way, and is not a modification of an existing tool. Evolve facilitates the creation of a base application architecture, and the development of any extensions on top of this. It also allows strata to be shared and distributed amongst developers in an extension setting.

Evolve uses UML2 [Obj09e] composite structure diagrams for depicting Backbone models. These diagrams can show a fully expanded structural view of class-like elements, and are commonly used by industrial practitioners to show the internal composition of classes [Sel03]. Composite structure diagrams also integrate well with extended state charts and port sequence diagrams, allowing the behavioural side of a system to be described within the same structural context. Extending Backbone to cover the behavioural aspects of a system is considered in section 8.3.

UML2 has been criticised for the excessive size of its metamodel, the overlap between different diagrams[1] and the lack of a precise semantics [EK99]. To sidestep these issues, Backbone instead takes a lightweight approach to integration by mapping Backbone concepts onto a subset of UML2. We then use the stereotype extension mechanism to build the extra rules into UML2 model. By using this approach, Backbone leverages a familiar and well-accepted industrial component model, and builds on this foundation in a principled way without incurring the full complexity of the UML2 model. Further, the formal specification of Backbone is dealt with separately, avoiding the complex and uncertain area of UML2 semantics.

5.2.1   Navigating the Model

Evolve allows a single diagram to be associated with every stratum. Consequently, nested strata allow for nested diagrams.

To navigate into a stratum and view its diagram, the user double clicks on the stratum symbol. To navigate to the parent stratum, the user double clicks the diagram background. This allows for fast navigation around a model. Via the tabbed interface, it is possible to have any number of diagrams visible on the screen at any one time. Multiple views of a diagram are supported, as well as multiple top level windows for a single model.

A strata diagram can display any of the components and interfaces defined within the stratum, and also any definitions that are visible to that stratum.

5.2.2   Recording and Visualising Deltas

UML2 composite structure diagrams show components which have been fully expanded structurally – there is no provided notation for structural deltas. Early attempts at the depiction of Backbone deltas

---

[1]A UML2 consortium member confided that the overlap between composite structure and component diagram types was well known at the time the specification was written, but both were included for political reasons.

showed that visualising the fully expanded form allowed system creation and extension to be handled uniformly and intuitively. This is in keeping with our philosophy that extension and alteration should be as easy as initial creation.

As such, Evolve always shows the fully expanded components resulting from resemblance, but when edits are made to the model it records these in the underlying repository as deltas. At any point a designer can toggle delta indicators, showing which parts of the structure have been added, deleted or replaced. The indicator symbols are shown in table 5.1.

Icon	Delta Indicator
	Added constituent
	Deleted constituent
	Replaced constituent

Table 5.1: The delta indicator symbols

Figure 5.5 below shows the consolidated desk model from figure 3.31 inside the modelling tool. The designer has hovered over the replacing part and a popup indicates which part it has replaced. Constituents which have been inherited do not have a delta indicator.
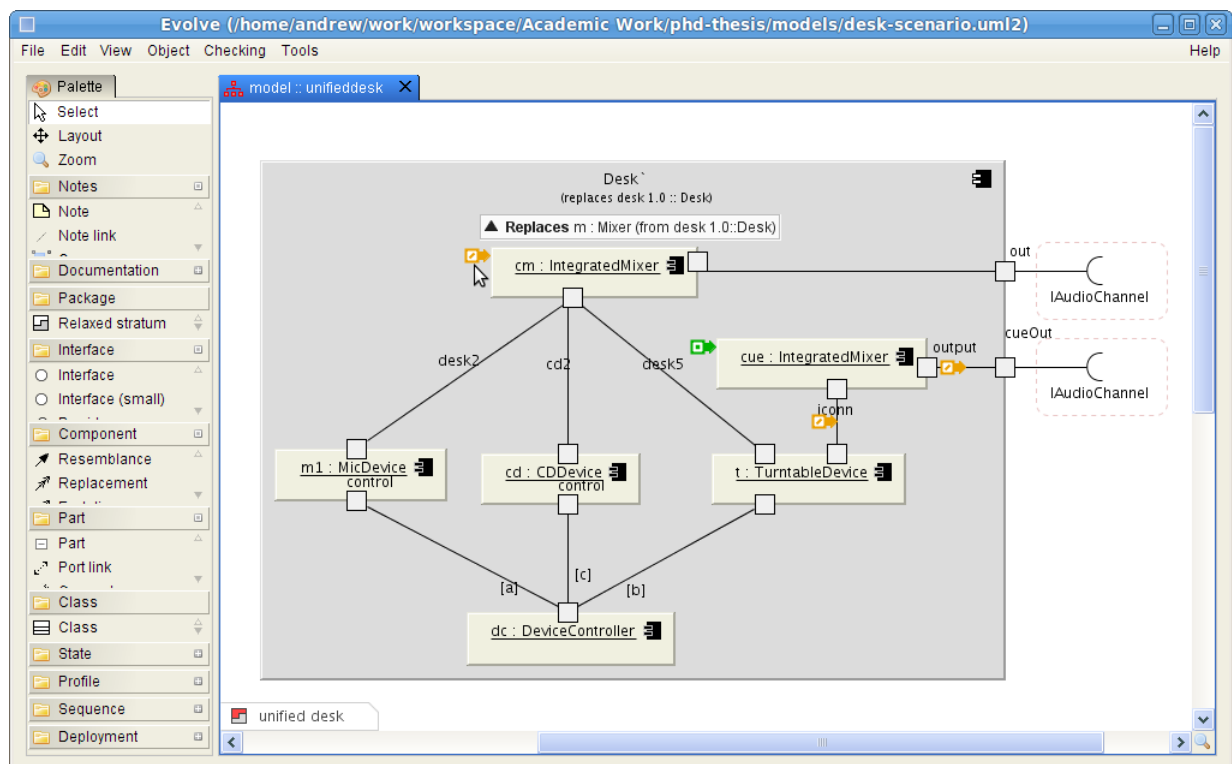
Figure 5.5: Evolve showing delta indicators

Evolve includes a browser which shows the underlying component structures and deltas from the repository. Figure 5.6 shows the repository representation for the expanded Desk' component of figure 5.5. The Desk' component is highlighted, and we can see that it evolves Desk, adds the cue part, replaces the m part and replaces the iconn and cconn connectors.
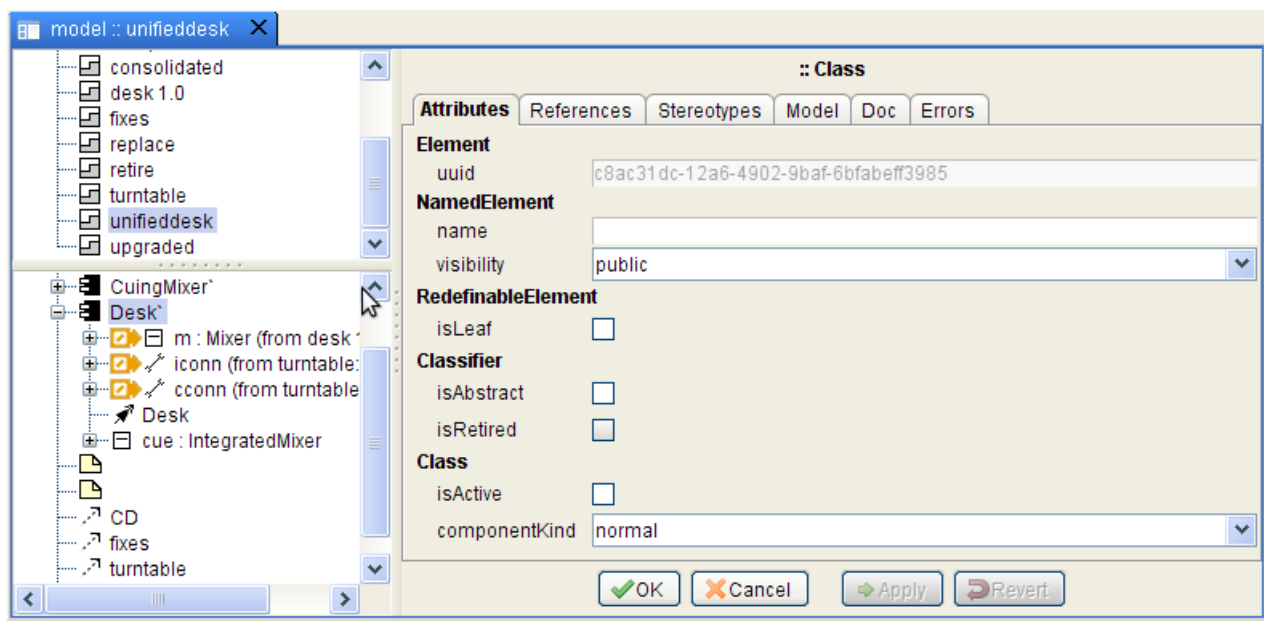
Figure 5.6: Viewing deltas in the browser

The dual approach of showing expanded structures in the diagrams (with optional delta indicators) and manipulating deltas in the browser works well in practice. The developer can work with the full structures in the diagrams without the need to continually refer back to the resembled elements to understand the full structure. Showing and editing deltas in the browser also resolves the problem of how to delete a deletion, or delete the replacement of a constituent. Both can be achieved by simply deleting the appropriate delta via the browser.

5.2.3   Resemblance as a Visual Concept

The concept of resemblance carries over neatly into the visual dimension also. The layout of parts, ports and connectors of an element can be inherited from elements being resembled. Consider figure 5.7(a) which shows component B without a resemblance relationship to component A. Figure 5.7(b) shows that as soon as we make B resemble A, the structure and layout are both inherited, giving a similar visual form.

Subsequent additions to A will also result in the layout of these constituents being copied into B also. However, the visual layout is just a copy taken at a point in time, and the layouts of A and B can diverge at a later point. Unlike the visual link, the structural link between the components is always in force as long as the resemblance relation is present: any structural changes made to A will also cause changes to the expanded structure of B.

Existing deltas can be made invalid by removing resemblance relationships. Consider if we add a delta to B to delete the connector in figure 5.7(b), and then we subsequently delete the resemblance relationship. The deletion of the connector will now be invalid because B no longer contains an inherited connector to delete. To rectify this, we can either re-establish the resemblance, or delete the delta via

(a) before resemblance   (b) after resemblance, showing inheritance of layout
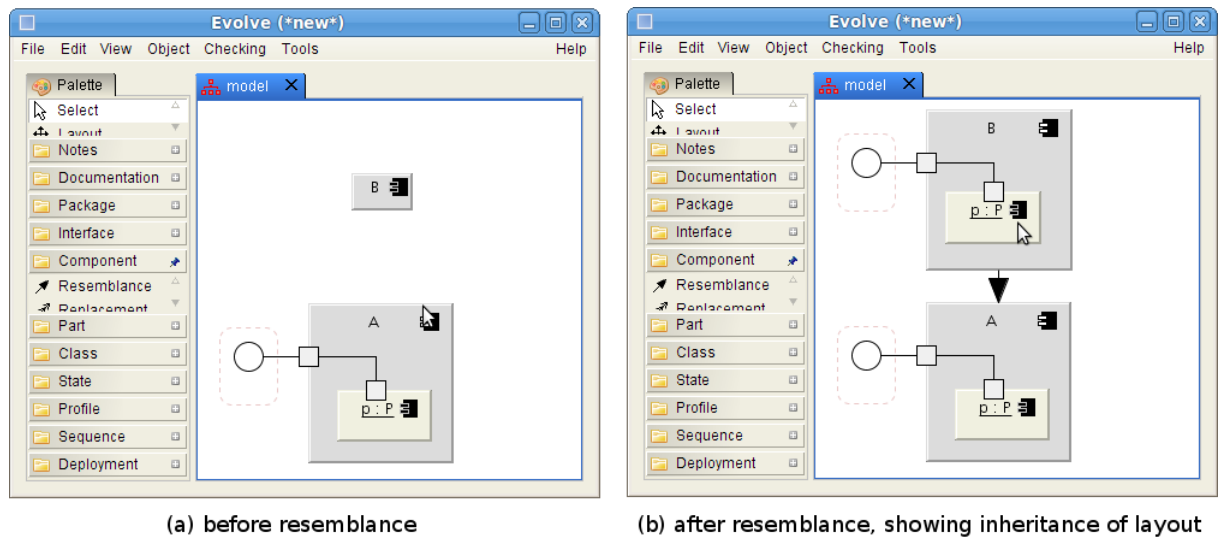
Figure 5.7: Resemblance operating at a visual level

the browser.

5.2.4   Renaming and UUIDs

Each element or constituent has a UUID, which remains constant regardless of any renaming. The modelling tool hides these from the developer, although they are used internally to track the various parts of the model in a robust way. In effect, names are just human readable tags which help the developer understand the model. The DeltaEngine treats these names as just another constituent which can be replaced.

Names can be omitted from the model in most cases. Although Evolve insists on names for strata, components and interfaces, this is for stylistic reasons. Names are only truly mandatory when constituents have a direct link to implementation artifacts, such as for port and attribute names in leaf components.

Figure 5.6 shows that the UUID of the `Desk'` component is `c8ac31dc-12a6-4902-9baf-6bfabeff3985`. The name is inherited from the `Desk` component and a prime is added to show that it is an evolution. However, we can give this component another name, in which case this will replace the name constituent of the component: in effect we are able to evolve the name. This demonstrates how deltas are applied uniformly to all aspects of the structure and composition of a component. Names are regarded as another part of an element's structure.

The use of UUIDs means that elements can be moved between strata without losing their logical identity. It is quite common when elaborating a model to move an element from a base stratum to an extension stratum and vice versa.

5.2.5   Sharing Strata and Handling Evolution

Strata are the unit of import and export in the Evolve modelling tool, allowing extensions to be distributed and shared between parties in an extension setting. A set of stratum (including their nested children) can be exported as a file, and transferred to another developer to be imported into their model. This facility does not prevent a group of developers from also using a CM system to manage parts of a model, or indeed an entire model – it simply provides a lightweight sharing facility which can be combined with other sharing approaches. In an extension setting where no common CM infrastructure exists between parties, the exchange of files ensures that models can always be shared. To facilitate the tracking of versions when exchanging strata files, each stratum contains a set of fields that can be used to store possible version details.

The use of UUIDs allows us to retain the concept of logical identity for elements even in the presence of disconnected files. Importing a stratum raises a number of interesting possibilities: an older copy of that stratum may already exist in the model, the new stratum may contain an element which is already in a different stratum in the existing model, or the new stratum may delete existing elements. In all cases, the imported stratum is treated as newer and authoritative, and UUIDs are used to determine which stratum and elements are being replaced. Any existing references are updated to the newer elements.

Figure 5.8(a) shows how the modelling tool allows examination of a strata file before importing. In this case, we are importing an edited version of stratum CD from the example in section 3.3.4. The modified stratum has deleted the CDDevice component. Before we consent to import this stratum, the tool warns us that the new stratum will cause problems for several references, as shown in figure 5.8(b). The warnings do not prevent the import, but indicate that the final model will contain errors where existing elements and diagrams previously referenced the deleted component. The errors can be fixed by following the import warnings and correcting any incorrect references.

(a) examining a file for import   (b) warnings due to deleted elements
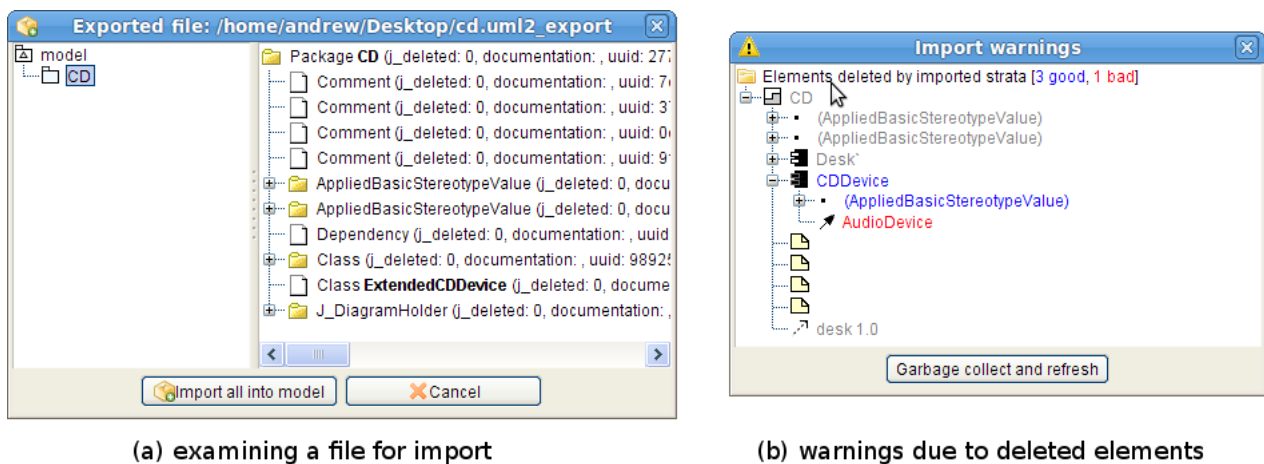
Figure 5.8: Examining and importing a stratum

The Backbone rules ensure that a stratum cannot access the definitions in its parent stratum. This allows an extension stratum to be isolated from a model, fitting well into an extensibility setting –

we can export an extension without needing to export its hierarchical parents. This is the opposite of the rules for UML2 packages. As explained in section 5.2.11, we use stereotypes to model strata as packages with extra constraints to prevent a stratum from accessing definitions in its parent.

5.2.6 Error Checking

Evolve provides a graphical interface to the DeltaEngine error checking facilities. It is possible to check the entire model from every perspective, which will check each stratum in turn, along with any transitively depended on strata if replacement is present. In the model from section 3.3.7, this results in 78 different strata checks from the required perspectives. Another option is to check the model from the perspective of a single stratum: this results in 13 different strata checks from the single `combined` perspective. The different checks in this case correspond directly to the unique strata in the model. This removes any combinatorial issues at the cost of not checking the model from every possible angle.

If any errors are found, they are visible in both the diagram and the browser. Figure 5.9 shows a simple model which contains errors recorded against the `AdvancedLeaf` component in the `base` stratum. In the diagram, error icons are shown against the component itself: these are known as *direct errors* because the errors exist in the element's home stratum. Hovering above an icon explains the error in more detail. Some error checks, such as a leaf component port not having a name are only checked in the home stratum. Other error checks, known as well-formedness rules, are performed from each perspective (section A.4).
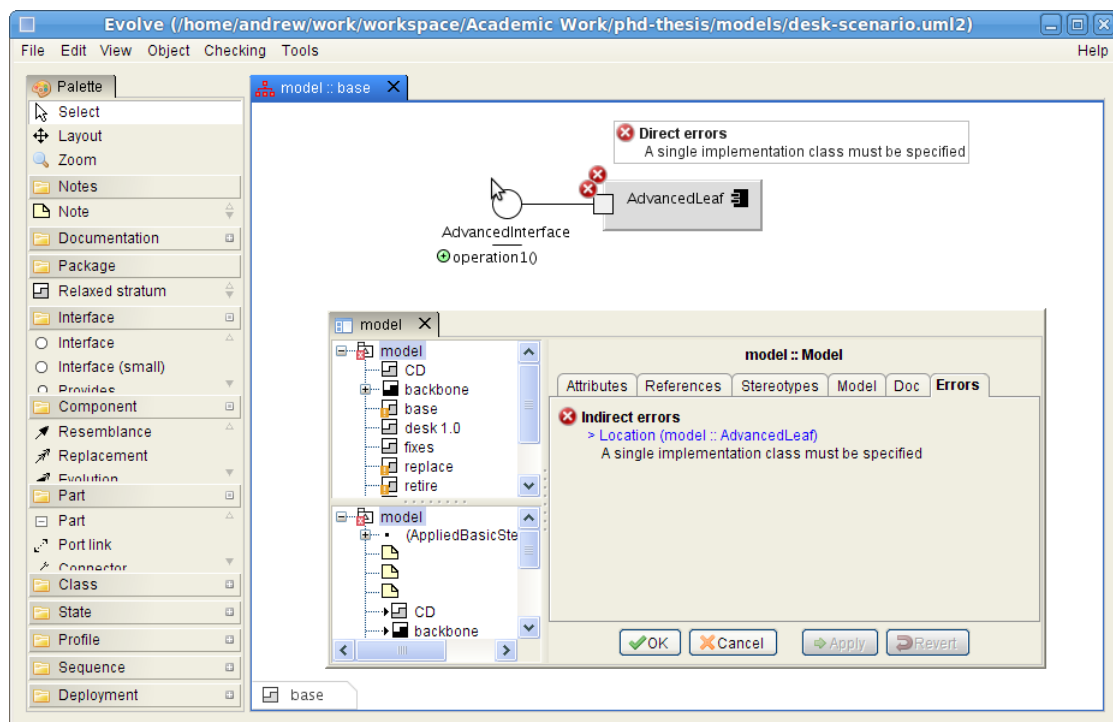
Figure 5.9: Viewing errors in the diagrams and browser

With replacement, it is possible for a subsequent stratum to correct (or introduce) errors in lower down strata. Errors against an element, when the perspective is not the element's home stratum, are known

as *indirect errors* because they result from a combination of definitions in one stratum interacting with the definitions in another. As the error is a combination of element and non-home stratum, there will not be a diagram element to list the errors against. Instead, these errors are shown in the browser listed against the stratum perspective that resulted in the errors. We can see this in figure 5.9 – the browser shows that from the perspective of `model` (which is the top hierarchical stratum containing `base`), the `AdvancedLeaf` component still does not have an implementation class specified.

5.2.7   Viewing the Model from a Fixed Stratum Perspective

As replacements from extension strata are applied to an existing base, the definitions change. Normally, however, each diagram is only shown from the perspective of the owning stratum. Consider, for example, the original definition of `Desk` from stratum `desk 1.0`, as shown in figure 3.12. The diagram shows all of the definitions of `desk 1.0` only, and does not show any of the replacements from `combined`.

Evolve allows any stratum to be set as a fixed perspective, so that existing diagrams can be viewed after extensions have been applied. Figure 5.10 shows what the same diagram looks like when `combined` is set as the fixed perspective. The diagram changes are made automatically, sometimes resulting in an imperfect layout, but it gives a view into what an existing diagram looks like from a non-home perspective. When using a fixed perspective, any diagrams from strata "below" that perspective are regarded as read-only.
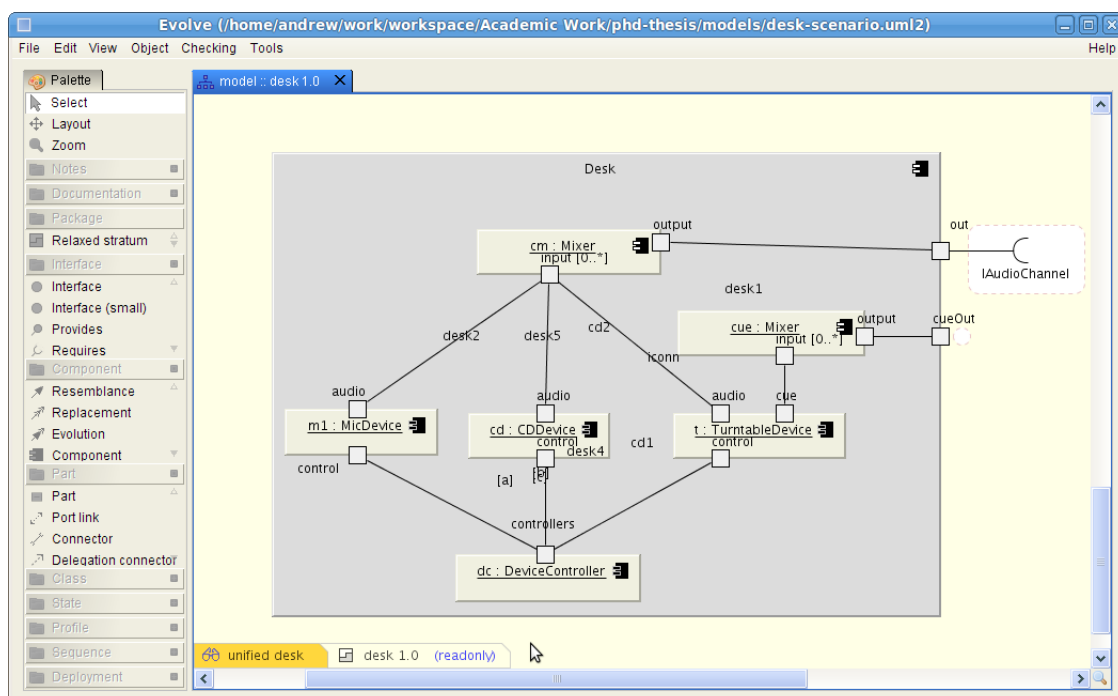
Figure 5.10: Viewing diagrams from a fixed perspective

5.2.8 Performance Considerations

In addition to the DeltaEngine performance considerations discussed in section 5.1.4, Evolve further ameliorates any performance issues by only expanding elements which appear on the currently edited set of diagrams. Furthermore, it lazily loads diagrams on demand, and removes unmodified (and currently unviewed) diagrams from the working set on a least-recently-used basis. This is transparent to the developer using the tool. The approach greatly limits the amount of work that must be performed when manipulating component structures in a diagram.

This type of lazy loading must be built deep into the graphical approach, as it affects many of the infrastructure facilities such as undo/redo and the repair of diagrams when an element is deleted.

5.2.9 Mapping JavaBeans onto Backbone

JavaBeans [SUN09b, O'N98] is a lightweight component model for Java, as described in section 2.3.2. Many Java libraries consist of a collection of beans. Being able to interoperate with this component model allows Backbone to be used with a large amount of existing software, whilst still retaining the full extensibility benefits of the approach.

This section outlines the mapping of the JavaBeans component model onto the Backbone component model and examines the limitations and restrictions of beans compared to Backbone components.

Essentially, a bean is just a plain Java class that conforms to a small set of lexical conventions for describing properties (analogous to Backbone attributes) and events. The JavaBeans component model can be mapped neatly onto a subset of the Backbone component model, with the advantage that beans can then be used freely in Backbone architectures.

The Mapping

Beans do not explicitly support provided interfaces. It is possible for a bean to implement an interface, but this is not standard practice apart from where a bean implements a listener interface so it can respond to events from another bean. As such, we model each bean as a Backbone leaf component which supports a single provided "synthetic" interface through a port called main. The interface is synthetic in the sense that it is just the same bean class masquerading as an interface. This sleight of hand is necessary to conform the JavaBeans model to the component and interface dichotomy of Backbone.

As shown in figure 5.11, the `JButton` bean (which represents a GUI button in the Swing toolkit) is translated into a Backbone leaf component called `JButton` where the implementation class is set to `javax.swing.JButton`. The port `main` provides a single synthetic interface modelled as `IJButton`, where the implementation class is also set to `javax.swing.JButton`. Any bean attributes are simply mapped onto Backbone attributes of the same name. No other provided ports or interfaces are possible.
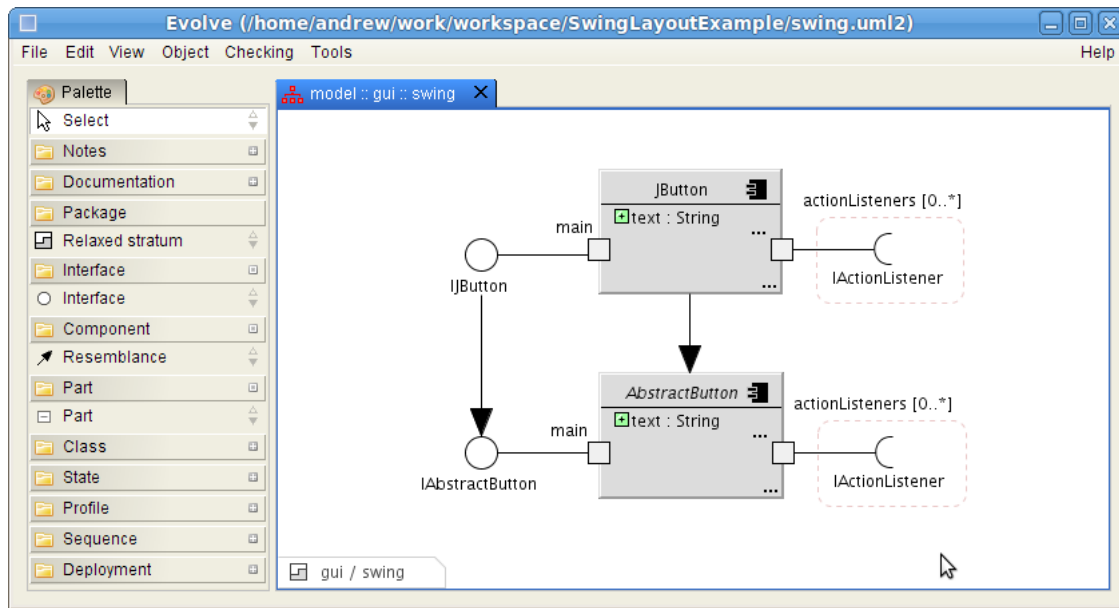
Figure 5.11: The JButton bean translated into Backbone

The JButton bean class inherits from AbstractButton. We translate this into Backbone resemblance between the two components and mirror this relationship on the interface side with resemblance between the two interfaces. We can see that the text attribute and the actionListeners port have been inherited. The main port of JButton replaces the inherited main port, and changes which interface is provided.

JavaBeans uses add methods to represent any required interfaces. We translate each add method into a single required interface on a port with [0..*] multiplicity, as shown for the actionListeners port. This port is used to register listeners with JButton in order to receive button events.

Our mapping also allows for a flexible interpretation of required interfaces. Although not explicitly mentioned in the JavaBeans specification, we choose to interpret a single get method which returns an interface as a port with a single required interface and a multiplicity of [0..1]. In some cases it is unclear as to whether a get method should translate into an attribute or a port, particularly in the presence of our synthesised bean interfaces. In these situations, the Evolve import tool guides the developer through the ambiguous choices.

The mapping exposes limitations in the JavaBeans component model. In particular, only one port with a provided interface is allowed, and every other port can only require a single interface each. Slots of bean parts cannot be aliased because there is no way to make the attributes of two separate bean instances refer to the same instance of a primitive type such as int. Further, JavaBeans provides no explicit support for composite structures, apart from simple object containment. The limitations on bean composite structures are further explored in [McV09].

The mapping is intentionally incomplete with respect to the different property types available in JavaBeans. Although the mapping does not support bound or constrained properties, we have not found this to be a limitation in practice as these are not frequently used.

Importing JavaBeans into Evolve

Evolve provides a tool for translating JavaBeans and importing them into an architecture. Figure 5.12 shows Evolve in the process of importing part of the Swing library. The top left tree shows the packages from the chosen Java library and the top right tree shows the beans found in the selected package. Beans can be selected and added to the import list, in the bottom left corner. The lower two trees show the beans after they have been translated into the Backbone component model. In this case, although we have only explicitly chosen `JButton`, we must also take a number of inherited classes and listener interfaces also. The bottom right panel contains the mapping of the attributes and interfaces of the bean into Backbone.
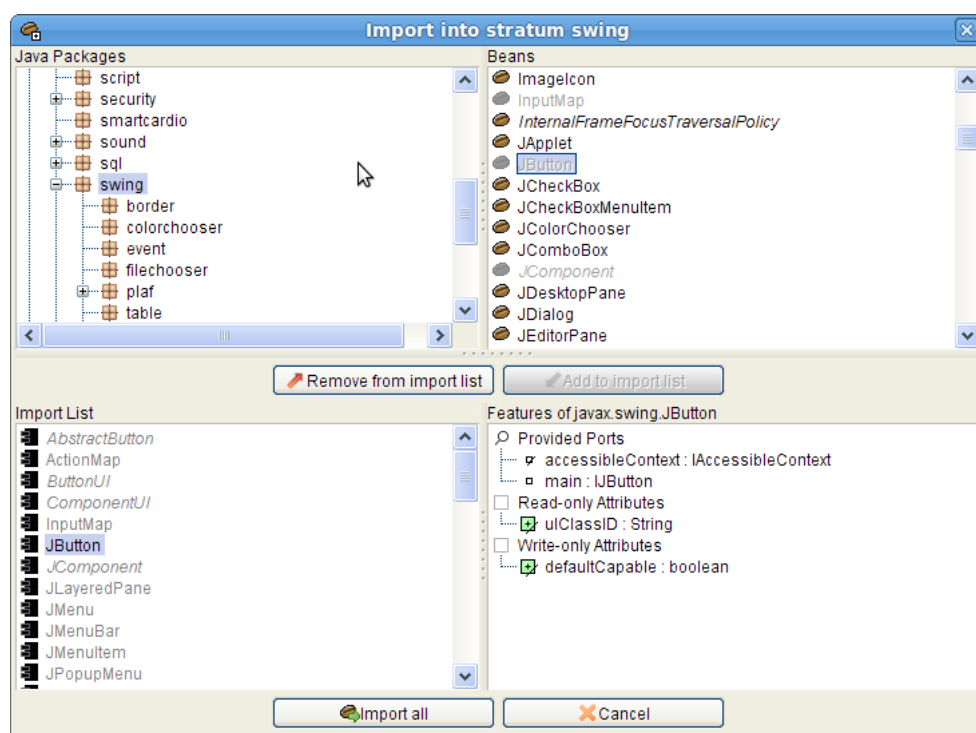
Figure 5.12: Importing JButton into an Evolve architecture

Graphical Management of Complex Components

When we developed the import tool and starting importing mature JavaBeans libraries, we soon realised that these libraries often involved many hundreds of components with complex inheritance relationships. Further, many of the components had a large number of ports and attributes. It was clear we needed to add extra mechanisms into Evolve to allow this type of complexity to be managed. To illustrate the problem, figure 5.13 shows the depiction of `JButton` with about half of its ports and attributes. The result is a complex and unwieldy graphical mess.

To address this, the import tool allows the developer to choose which constituents of a component are visible on diagrams by default. In practice, this is a relatively small number, as in the case where the primary attribute of `JButton` is the `text` attribute. The other constituents can be added at a later
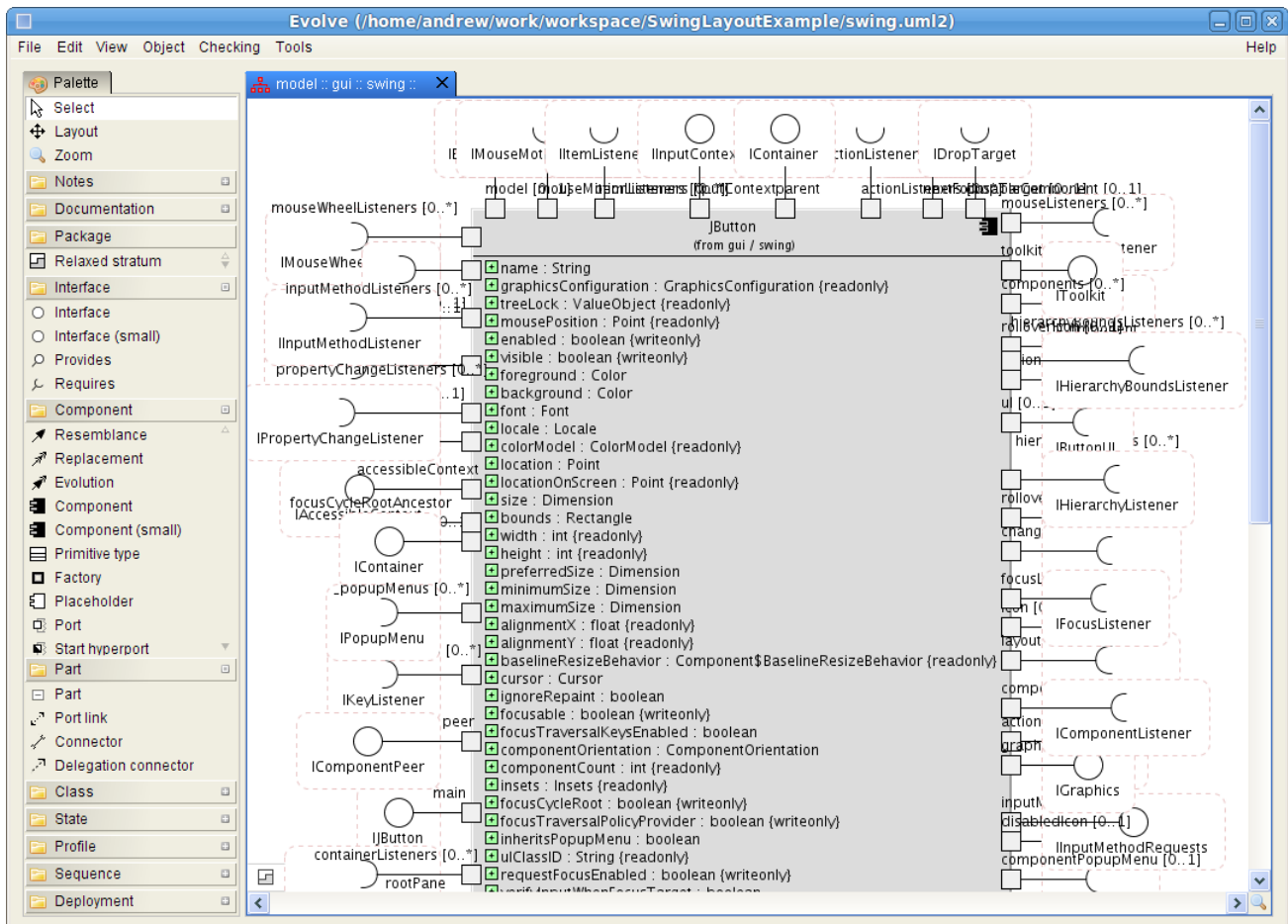
Figure 5.13: The fully expanded JButton component is unworkable

point, as shown in the menu of figure 5.14 where the developer is making a further port of `JButton`
visible. After experimenting with fish-eye menus [Bed00] and other graphical approaches, the multi-
column, alphabetically-sorted menus shown in the figure were found to be the most convenient.

A further approach to limiting the number of attributes and ports is to exclude them from the model
when translating them into Backbone. This is an acceptable option because, in practice, many of the
constituents of beans are never used. The import tool supports this.

Working with these large libraries prompted several of the performance optimisations mentioned in
section 5.2.8. The current version of the Evolve tool can comfortably cope with a model several times
larger than the full Swing library (which itself contains around 500 components) without any noticeable
graphical delay during manipulation. This enables the practical use of complex Java libraries with the
Backbone approach.

5.2.10   Generating Leaf Component Source Skeletons from Evolve

To minimise developer effort and design overhead, Evolve can generate a source code skeleton for
any leaf component implementation, upon request. Each skeleton contains Java fields to represent
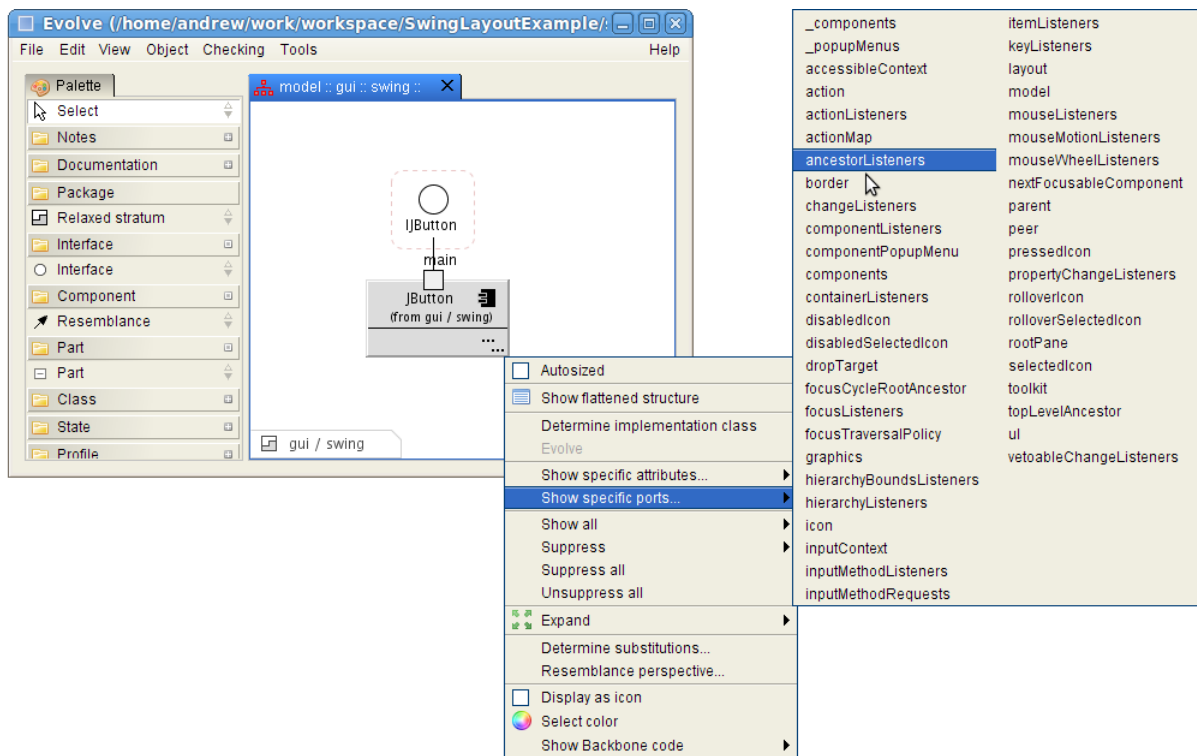
Figure 5.14: Coping with large numbers of constituents

attributes and ports, and also provides supporting methods and class definitions. Roundtripping, where further generation preserves code added by developers, is also supported.

To show the mapping into Java, consider the leaf component shown in figure 5.15.
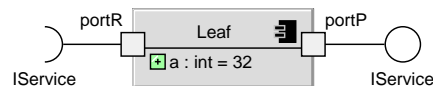
Figure 5.15: A simple leaf component

The generated skeleton for this is shown below.

```
public class Leaf
{
// start generated code (A)
// attributes
    private Attribute<java.lang.Integer> a =
        new Attribute<java.lang.Integer>(32);
// required ports
    private example.IService portR;
// provided ports
    private IServicePortPImpl portP_IServiceProvided =
        new IServicePortPImpl();
```

```
    // setters and getters
        public Attribute<java.lang.Integer> getA()
          { return a; }
        public void setA(Attribute<java.lang.Integer> a)
          { this.a = a;}
        public void setRawA(java.lang.Integer a)
          { this.a.set(a);}
        public void setPortR(example.IService portR)
          { this.portR = portR; }
        public example.IService getPortP_IService(Class<?> required)
          { return portP_IServiceProvided; }
    // end generated code (B)
        private class IServicePortPImpl implements example.IService
        {
            //@todo add interface methods
        }
    }
```

Lines (A) and (B) are the start and end markers. Any code within these comments will be regenerated upon request, overwriting any changes. The generated lines describe fields (and supporting methods) corresponding to any attributes or provided or required port interfaces. Attributes use the `Attribute` wrapper class, which supports aliasing even for primitive types such as integers. Each required interface port field is named after the port itself, unless there is a name clash. Provided interface port fields have the interface name appended as well.

`IServicePortPImpl` is an inner Java class that has been generated to hold the implementation of the provided port interface. This class is only added if the leaf source file does not exist, and subsequent changes to this by the programmer are not overwritten during roundtrip generation. Note that the generation of fields for individual attributes and ports can be suppressed if required. This allows custom logic to be manually placed inside the relevant methods.

Any changes made by a programmer, outside of the marker regions, is preserved when the skeleton is regenerated.

This approach to generation and roundtripping is a robust and flexible way of combining automatically generated code and manually entered code. Unlike other approaches that have previously been used by UML modelling tools, this scheme does not rely on the presence of fragile UUID markers in the code to track the identity of model artifacts.

5.2.11   Mapping Backbone onto UML2

This section describes the mapping from Backbone onto the UML2 model. As discussed in section 5.2, the approach of using a mapping rather than relying directly on the UML2 component model allows

us to avoid some of the excesses and redundancy of the complex UML2 approach.

UML2 includes the stereotype extension mechanism in order to support non-standard object models. This is further discussed in section 2.4.4. A stereotype is a lightweight form of metamodel extension where classes in a user model can virtually subtype the existing UML2 metamodel classes [HSGP06]. In practice this means that new attributes and constraints can be added to the existing UML2 object model in a lightweight way. A profile is a collection of stereotypes for a given purpose. Backbone qualifies as a profile of UML2, although it has not been explicitly phrased in this way.

Backbone is almost a proper subset of the UML2 composite structure model. As such, the description of the mapping is brief and simply involves pointing out the element that each concept maps to, and any constraints. A UML2 user, when presented with a Backbone diagram, should have little trouble interpreting its structural meaning. The full set of constraints are documented in section A, as a set of structural rules.

Backbone Components and Interfaces

Backbone components are mapped onto UML2 classes, rather than UML2 components. The latter do not provide any tangible benefits for our approach over the former, as UML2 classes are structured entities and can contain ports, parts and connectors. In any case, the UML2 `Component` metamodel class directly inherits from the `Class` metamodel class and adds little in the way of attributes. Our approach of using classes permits us to use composite structure diagrams rather than the more graphically verbose component diagrams. Backbone ports, parts, connectors (assembly and delegation) and attributes map directly onto their UML2 namesakes. Port links do not exist in UML2 and are represented as stereotyped dependencies.

Unlike UML2 structured classes [OL06], Backbone constrains composite components so that they cannot have an implementation. This forces composites to be purely structural entities which can always be flattened into a connected set of leaf instances, making it only necessary to specify implementation (and properties such as behaviour) at the leaf level. This facilitates analysis also, as we do not have to consider any implementation at the composite level. Backbone components are constrained to always feature at least a single port.

Parts in Backbone are constrained to have multiplicity of `[1]`. Parts of optional multiplicity are catered for by isomorphic factories which allow dynamic instantiation. This is described in section 6.3.2.

Backbone interfaces are mapped directly onto UML2 interfaces. To allow leaf components and interfaces to hold the name of a potential implementation class, we have added an extra attribute to the `Classifier` metamodel class using a stereotype. This is the common base class of the `Class` and `Interface` metamodel elements.

Deltas

A delta alteration can take the form of the addition, replacement or deletion of a constituent of an element. The addition of constituents (e.g. adding attributes) to a component or interface is already catered for in UML2.

To model delta replacement, we introduced the `DeltaReplacedConstituent` metamodel class and subclasses into UML2. This contains a reference to a replacing, and replaced constituent. By adding fields of type `DeltaReplacedConstituent` to the `Class` and `Interface` metamodel classes, we allow both classes and interfaces to contain constituent replacements.

Similarly, we introduced the `DeltaDeletedConstituent` metamodel class to model delta deletes.

Primitive Types

Primitive types are represented by UML2 classes, which can contain deltas for name and implementation class. Although not an area currently explored, there are no constraints that prevents primitive types from having their structure elaborated in terms of attributes, parts and connectors also.

Resemblance and Replacement

Resemblance is modelled as a stereotyped dependency between two elements of the same type. Consideration was given to introducing a `Resemblance` metamodel element, inheriting from the UML2 `Generalization` relation, but the direction of subtyping in this case conflicts with the notion that resemblance is more general than inheritance. Although inheritance is not present explicitly in Backbone, it can be regarded as a subset of resemblance with additional constraints prohibiting constituent deletions and also limiting the scope of constituent replacement.

Replacement is also modelled as a stereotyped dependency between two elements of the same type.

Stereotypes

Backbone models can also have stereotypes and profiles, and Backbone elements can be tagged with a stereotype. Not surprisingly, Backbone stereotypes and profiles map onto their namesakes in UML2.

Backbone differs from UML2 only in that we constrain each element to be tagged with at most one stereotype, rather than the multiple allowed in UML2. Element stereotype tagging in Backbone is handled via deltas, and a resembling element can replace or delete the stereotype tag it inherits.

Backbone stereotypes are used to expand an element by adding extra constituents to it automatically. This is described in section 6.1. The reason why we constrain each element to a single stereotype is that multiple stereotypes raised the confusing issue of order of expansion.

Strata

The Backbone stratum concept is mapped onto UML2 `Package`, with a stereotype to introduce new rules about visibility. A stratum cannot access any of its parent's definitions, which allows each stratum to be treated as a self-contained unit for import and export purposes. If a stratum could access definitions contained within the parent, then when we exported it we would potentially have to export some or all of the parents also, or always import it into a model which contained the parent.

UML2 packages have access to their parent's scope. This arrangement was inspired by the way that nested functions in a programming language have access to definitions within the scope of the parent function. As pointed out in [SW98], this is often a confusing arrangement which hampers the use of packages as a modular unit. Until UML1.3, the visibility rules were closer to those of Backbone, and after this point they were altered to their current definition.

Backbone contains rules about transitive visibility, documented in the formal specification, which resolve the confusing and inconsistent package visibility rules based around access permissions in UML2 [SW98]. The intention of the rules is to force a Backbone architecture to be explicit about layering and strata dependencies, thereby making the module structure of an architecture explicit.

Related but Unused Concepts in UML2

UML2 introduced the notion of *package merge* to allow the metamodel and specification to be organised into a number of layers, where each layer might add further attributes and properties to existing elements. Package merge allows two packages to be merged and any elements with the same names are coalesced into a consolidated definition. This is not dissimilar to the notion of viewpoints, as described in section 2.1.8. Unfortunately, package merge only allows for addition and some level of replacement and is considered too complex for conventional modelling [RJB04]. As pointed out in [ZA06], a modelling of package merge in Alloy has shown that it does not guarantee to preserve data model compatibility with the source packages: this is unfortunate as this compatibility was one of the key design goals. This has also revealed inconsistencies and unforeseen corner cases in the operation. UML2 does not formally specify this construct.

UML2 contains a facility known as the Object Constraint Language (OCL), which is a lightweight, formal language for specifying rules about a model. Rather than use OCL and attempt to integrate our formal specification into the large and sometimes inconsistent UML2 OCL specification [BGG04], we instead chose to specify Backbone using Alloy. We could have instead potentially used a combination of OCL and the USE toolkit [GBR07] for this purpose also. USE provides a workbench that can check an object model against an OCL specification. The snapshot generator, however, is less advanced that the Alloy Analyzer and involves specifying object configurations in detail using a language called ASSL [GBR03]. Counterexamples cannot be generated automatically which limits the effectiveness of the toolset in finding errors in a specification.

UML2 introduced the concept of *redefinition*, where a constituent nested inside a classifier can be selec-

tively replaced in the scope of an inheriting classifier. This facility clarifies and generalises the notion of when a method polymorphically overrides another in a base class. The rule is that the redefining constituent should be substitutable for the redefined one, and a set of rules exist for what constitutes substitutability for each constituent type. In the case of nested classifiers the vagueness of the rules have prompted questions and speculation even on the part of the specification's primary authors [RJB04]. Backbone resemblance is effectively a superset of UML2 redefinition (within the confines of the Backbone model), allowing also deletion and non-conformant replacement. Well-formedness rules flag any structural errors that may result.

UML2 also contains a substitution relation, which indicates when one class can be substituted for another. This was not used for the mapping because it implies a level of conformance to the base and also does not actually replace the base element. Instead it just indicates that a substitution is possible, leaving the mechanism for replacement unspecified.

5.2.12   Using Evolve in an Extension Scenario

Evolve is designed to support the workflow of developers in an extension setting. To examine this in detail, we use the modelling tool in the context of the desk extension scenario described in section 3.3.

Audiosoft starts the scenario by creating the `desk 1.0` stratum, associated definitions and leaf component implementations in their own environment. This stratum holds the base application. As we can see from figure 5.16, only this stratum is present in the model: Audiosoft has no need to see any of the extension strata. Upon completion, Audiosoft exports this stratum as a file and sends this and the compiled leaf and interface implementations[2] to developers X and Y who plan to extend the desk to integrate their own device controllers.
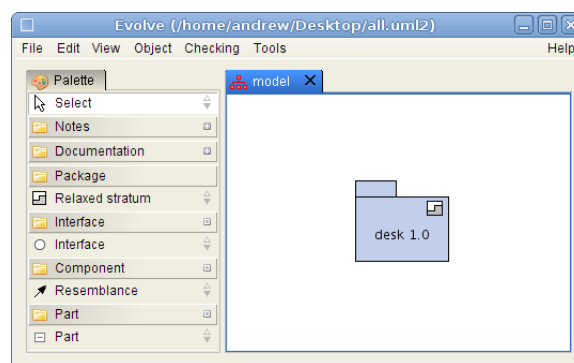
Figure 5.16: Audiosoft's model

Developers X and Y receive the `desk 1.0` stratum and each import it into their own projects. This stratum is imported as read-only, denoting that X and Y are not the primary owners and should not directly edit the definitions within it. X creates the `turntable` extension stratum as shown in figure 5.17(a). Y creates the `CD` extension stratum as shown in figure 5.17(b). Both X and Y use resemblance and replacement to create any alterations required to the base system represented by the `desk 1.0`

---

[2]The compiled Java implementations are usually packaged as a JAR file.

stratum. From the figures, we see that X and Y's models only include the strata that they own, or that they build upon.
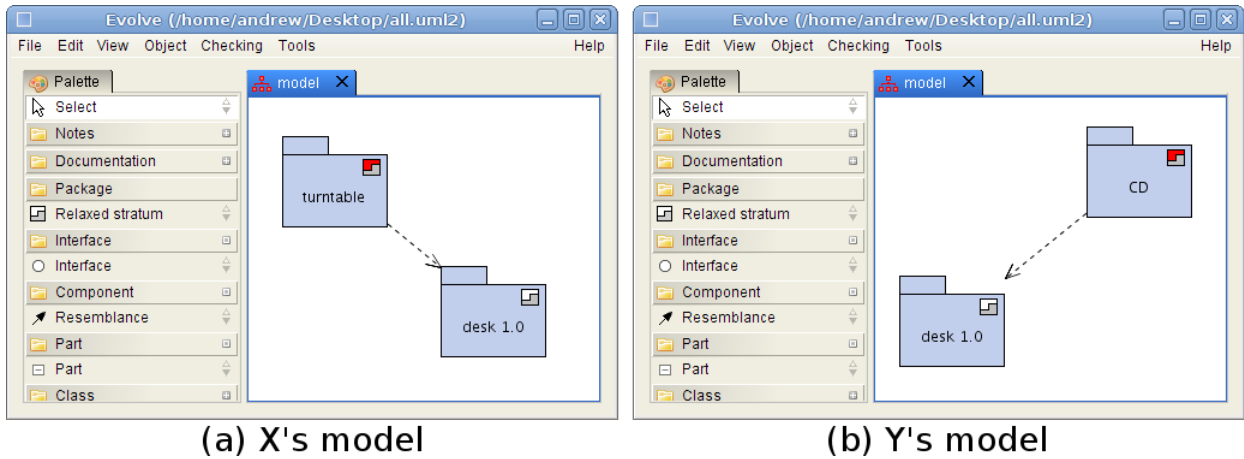
Figure 5.17: The models of X and Y

Upon completing the extension, X exports the `turntable` extension stratum into a file and distributes this and all associated implementations to radio station R. Similarly Y exports the `CD` extension stratum and also gives this to R along with any implementations. R imports the `desk 1.0`, `turntable` and `CD` stratum into their own model. Because all parties (apart from Audiosoft) treat the `desk 1.0` stratum as read-only, it does not matter who R receives this stratum from – it will be the same as that issued by Audiosoft.

To combine the two extensions, R further creates the `combined` stratum resulting in the organisation shown in figure 5.18. Note that in the scenario, only R's model holds all of the strata. Other parties only hold a subset of the system depending on which areas they are responsible for.
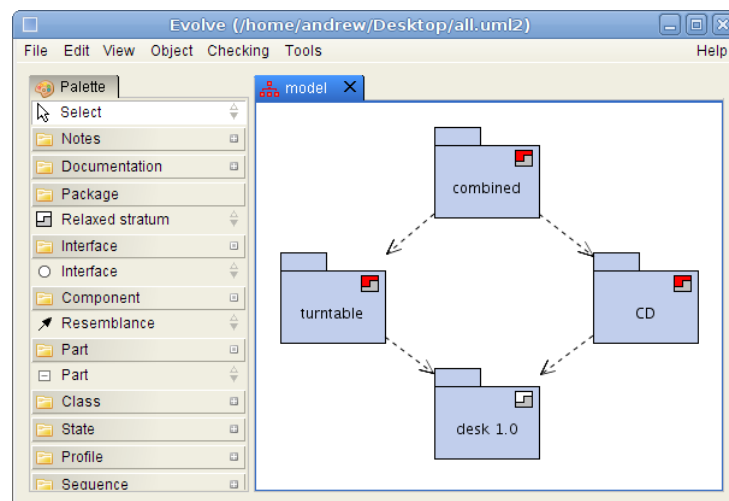
Figure 5.18: R's model

Resolving Conflict in Combined Extensions

After importing the extension strata and combining them by creating `combined`, R runs a full error check which shows a number of indirect errors listed against `combined`. The combination of extensions has resulted in structural conflict.

To determine what has gone wrong and to effect a resolution, R navigates into the `combined` stratum and places a copy of the `Desk` component on the diagram. As shown in figure 5.19, this reveals a number of connector errors. As per section 3.3.7, to correct these we require an evolution of `Desk` to replace the `CuingMixer` part with an `IntegratedMixer` part and also introduce a further mixer part for the cue bus.
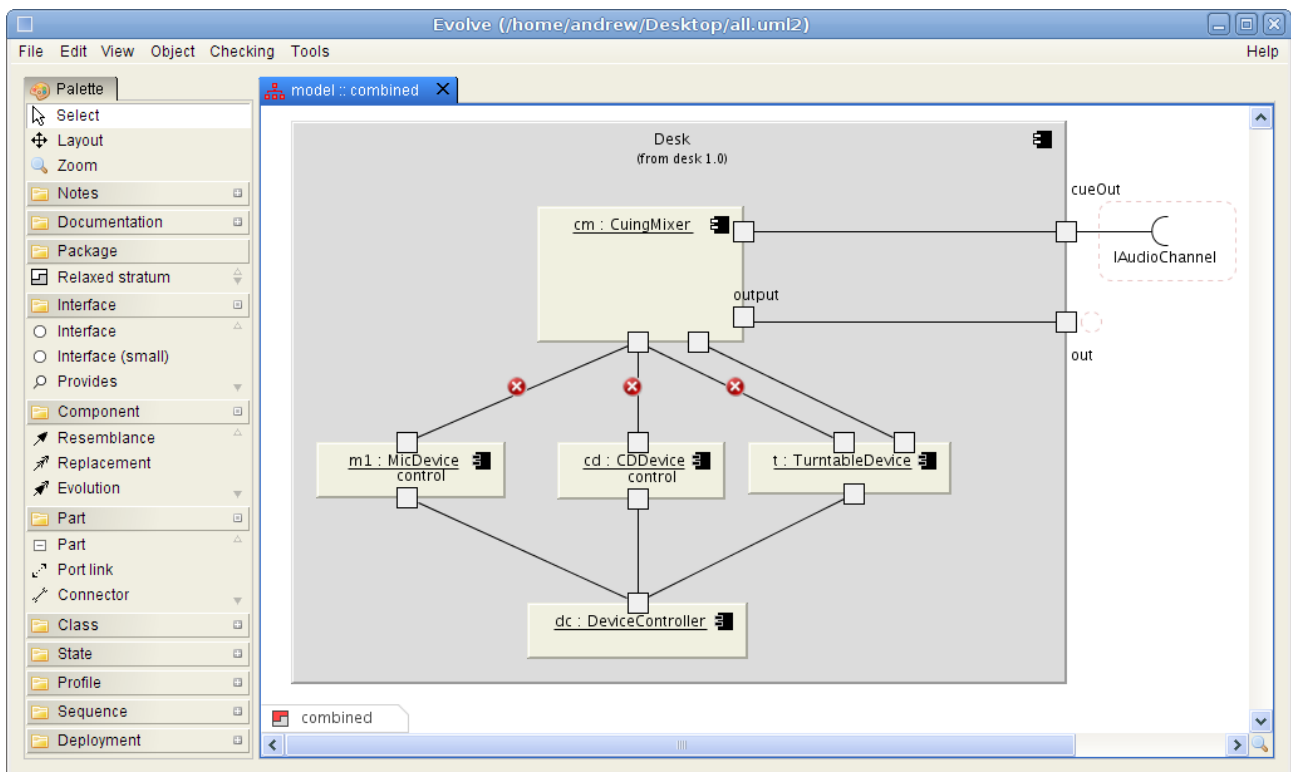
Figure 5.19: The Desk component has errors from the combined perspective

Right clicking on the `Desk` component gives the developer the option to evolve it. Upon choosing that, an evolved `Desk'` component appears in the original's place. We can now add, replace or delete the structure to correct any errors.

To accomplish the structural corrections mentioned above, we first add an `IntegratedMixer` part for the cue audio, and replace and reroute two of the connectors to it, as shown in figure 5.20.
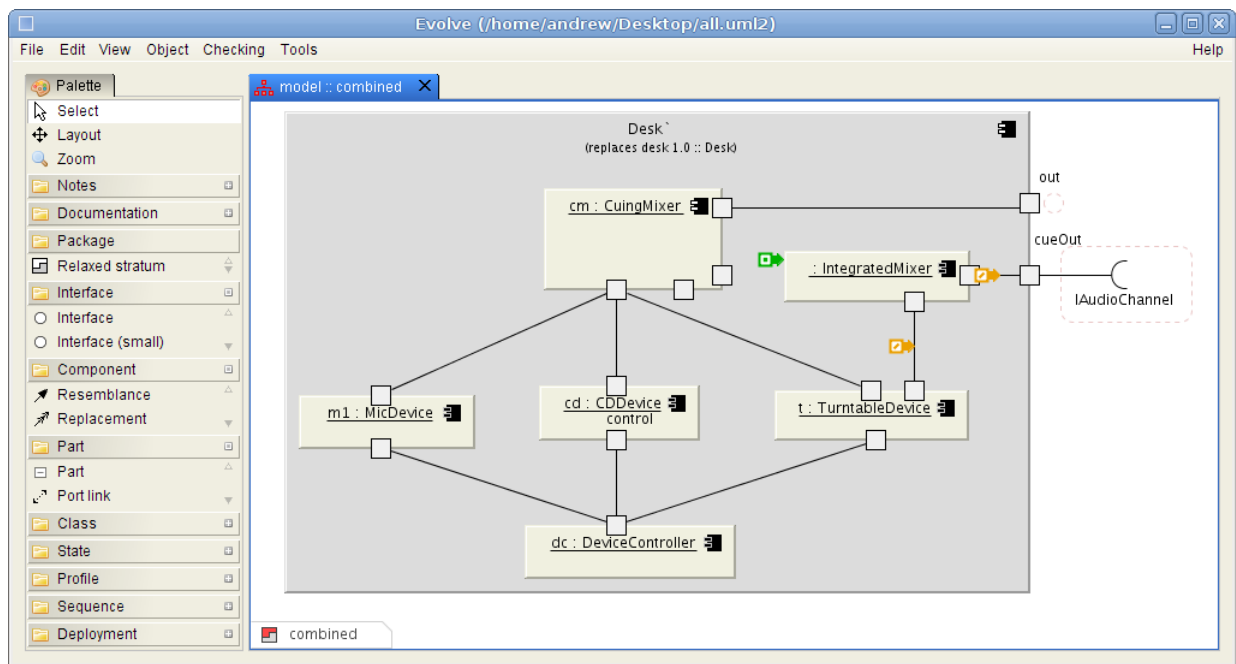
Figure 5.20: Adding a mixer for cue audio

Finally, we must replace the CuingMixer part with an IntegratedMixer part. To accomplish this graphically, we add the new part, select it, then right-click on the old part and choose the "Replace (with existing part)" menu item as shown in figure 5.21. This creates the appropriate replace delta. Because both CuingMixer and IntegratedMixer resemble Mixer and the ports are inherited from Mixer in each case, the replacement part will also work with the existing connectors. The replacement results in the corrected structure of figure 5.22.
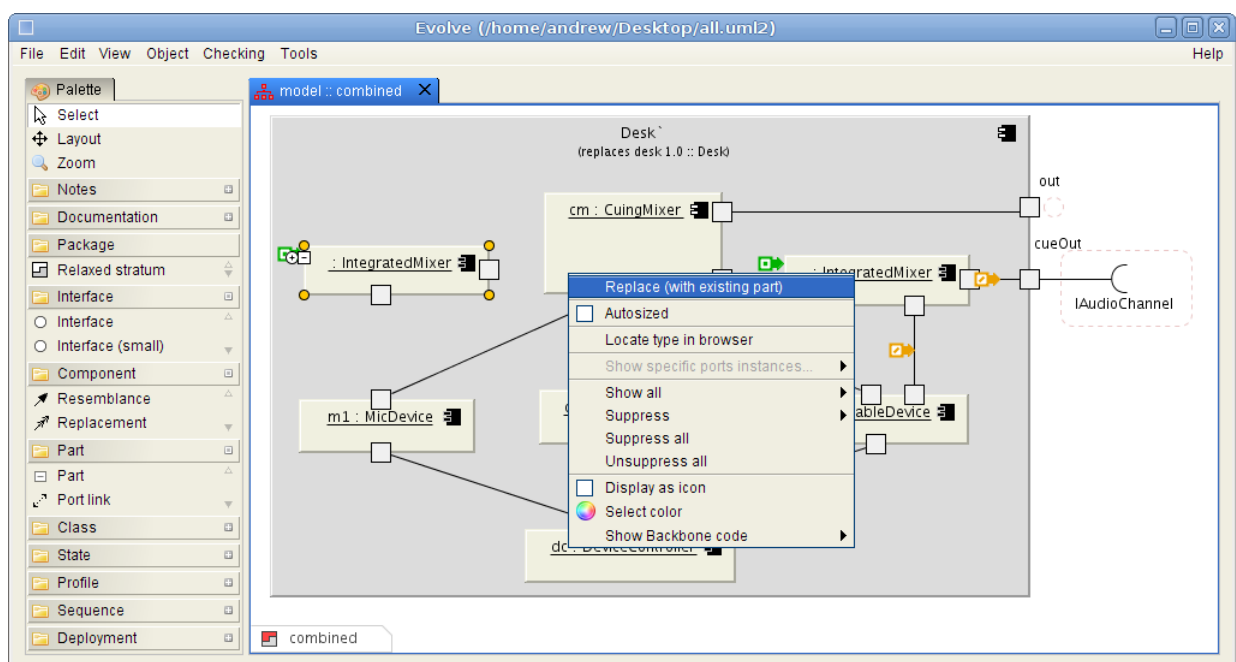
Figure 5.21: Replacing the mixer
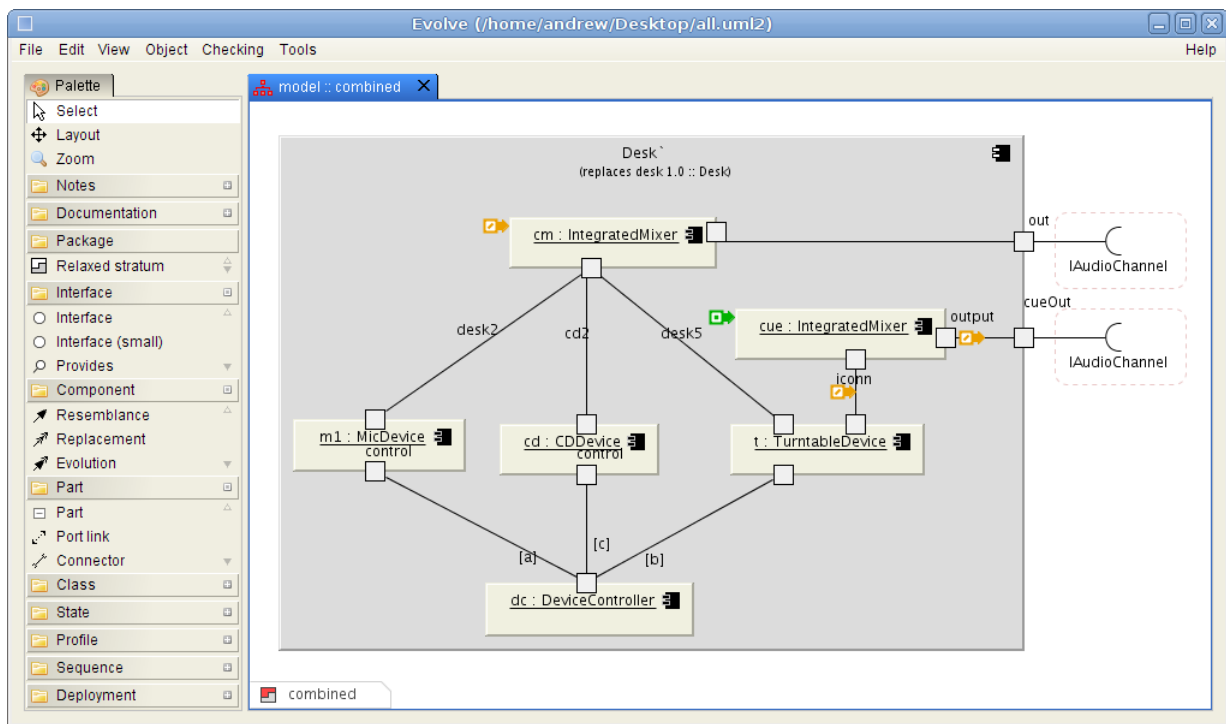
Figure 5.22: The corrected Desk component

Distributing Upgrades and Fixes

When Audiosoft decides to upgrade the `Desk` component, they have two choices: they can directly edit and redistribute the `desk 1.0` stratum which they own, or they can make a further stratum and use resemblance and replacement to express the upgrade. Evolve supports the former option by allowing a stratum to be replaced upon import, which allows others to import an edited version. The use of UUIDs minimises the impact for consumers of modified strata, although clearly the scope for disruption depends upon the nature and extent of the changes.

In our scenario, Audiosoft chooses to use the `upgraded` stratum to phrase the upgrade to `Desk`. They also create a stratum `fixes` that R can use to hold fixes for the upgraded desk. Audiosoft will own and distribute the `upgraded` stratum to R. However, R will own the `fixes` stratum and distribute this back to Audiosoft. With this arrangement, X and Y have no dependency on the `upgraded` stratum and therefore are not affected by it. The dependency structure of `fixes` ensures that no fix can reference the extensions of X and Y. Figure 5.23 shows the final structure of Audiosoft's model.

5.2.13   Methodology Agnostic

Evolve does not prescribe an analysis or design methodology. Instead, it aims to support the building blocks of such approaches, by allowing both top-down decomposition and bottom-up system construction in an iterative context.
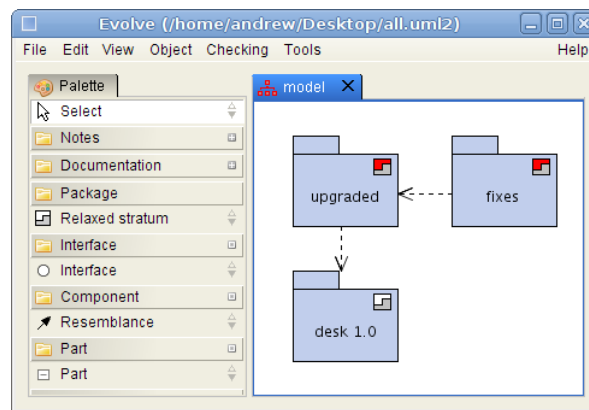
Figure 5.23: Audiosoft's upgraded model

A top-down approach typically starts with a component representing the entire system and then proceeds to break that down, until over time the system is structured as a composition hierarchy with fine-grained leaf components at the bottom. To support this activity, Backbone allows leaf components to be further decomposed by direct editing, thereby turning them into composites. The use of UUIDs enables clients of the component to be unaffected. As an alternative, an extension can be used to decompose leaves into composites via evolution. Placeholders can also be used judiciously during the elaboration process to earmark parts of the architecture that are speculative or have not been fully decomposed.

Bottom-up construction involves defining the leaf components representing the base functionality of the system, and then progressively assembling these into higher level abstractions over time. Backbone supports this through composite components. Resemblance and replacement can be used to adjust and reuse parts of a hierarchy, which ameliorates one of the problems in a constructive approach where abstractions get buried deep within the hierarchy and cannot be changed.

Backbone further supports design in an iterative context, where additional functionality is built up over successive releases by refining a system. In this case, different workflows (particularly testing of the old release and design of the new) can overlap. Backbone enables this by allowing new releases to be phrased as extensions, allowing both the old and new system to co-exist.

5.2.14   Summary

Evolve is a graphical modelling tool which fully supports the Backbone approach. The extensibility concepts deeply affect many aspects of the design of this tool, including the approach to performance, delta management and element expansion. The import and export of extension and base strata from the environment allow the sharing and distribution of software required in extension scenarios.

Evolve diagrammatically shows the fully expanded structure of components, whilst recording deltas in the underlying repository as the developer creates and edits. This allows the benefit of both approaches – developers see the full structures they are working with, but deltas can still be viewed, manipulated and deleted as required.

By using UML2 composite structures to depict the structure of a system, Evolve draws on a widely accepted graphical notation for depicting software. On the negative side, UML2 is complex, convoluted in parts, with overlapping diagram types and no precise semantics. Backbone, by retaining its own formal specification and mapping its concepts onto those of UML2, sidesteps these issues thereby retaining the benefits of a familiar graphical approach without incurring too much of the underlying baggage. Evolve is not a full UML2 tool as it does not cover all the UML2 notation, although it could be used as the basis for such a tool.

Evolve also retains many of the advanced features of mature UML2 tools. It is a full graphical editor, designed using the tool-command-view approach pioneered by Unidraw [VL89]. It separates diagrammatic views from the underlying model representation, allowing components to be shown in alternative ways in multiple diagrams if required. This graphical flexibility provides ways to elide complex parts of a design via multiple views, which neatly complements the ability to control complexity through compositional hierarchy alone.

Evolve uses the same DeltaEngine as the runtime environment, ensuring consistency. It uses a full UML2 repository to store its models, and base or extension strata can be exported and imported in a way which tolerates direct editing of a stratum after it has been distributed. Evolve can also be configured to use a central object database to support strata sharing and closer collaboration between developers.

Evolve is methodology agnostic, and does not prescribe a particular approach to designing or elaborating a model. It supports a combination of top-down decomposition and bottom-up system construction in an iterative context.

5.3   Runtime Environment

The runtime environment allows a Backbone architecture to be instantiated and executed.

A Backbone system is executed by flattening the specification of a composite representing the system into a set of connected leaf component instances. The classes representing the leaf components are then instantiated and connected appropriately and control is handed over to a specified port. The runtime model and facilities are made available through selected interfaces to instantiated components. It is not necessary for components to have any dependency on the runtime if desired, allowing leaves to be developed and tested completely outside of Backbone.

Figure 5.24 shows that deltas to a model can be applied at various points along a spectrum in order to achieve extensibility. At the far left, we can apply the deltas to a source code representation of an architecture to produce an extended system. This is useful if a developer is using Backbone to produce variants of a system rather than delivering extensions to other parties. The middle point of the spectrum shows that we can alternatively apply the deltas to an architectural representation at program startup in order to instantiate the extended system. The far right of the spectrum shows that we can also potentially apply the deltas at runtime. The last form of application is not currently

covered by the Backbone toolset, and can be complex. In that particular case we must consider the transition of state between the old and new running systems, and take into account quiescence to determine when it is safe to remake the structure of the components as described in [MK96].
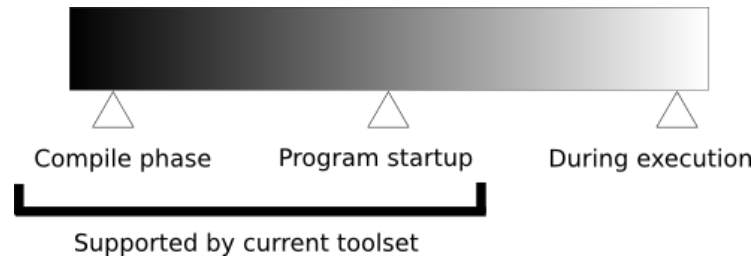
Figure 5.24: The spectrum of possible delta application points

Many extensibility approaches utilise the middle of the spectrum and use a configuration to form the extended architecture at program start time. This requires a restart of the system whenever an extension is added or removed. OSGi-based approaches have started moving in the direction of runtime extension via a service-oriented architecture [OSG03, OSG09], but data transfer from old to new components is rarely considered or even allowed. In some architectural styles, the problems of state transfer can be avoided by using components that are stateless and by re-reading any required data from a central database. This suits a certain type of application, but is not generally applicable.

5.3.1 Flattening the Composition Hierarchy

A composite is flattened by starting with the leaf parts of the composition hierarchy, and recursively pushing these into upper levels until a single level remains. Although not described formally in the Backbone specification, the flattening occurs in a similar manner to that of Darwin configurations [KME95, MDEK95].

The Evolve modelling tool allows the flattened form of a given component to be displayed. Figure 5.25 shows this for the original `Desk` version 1.0 component of figure 5.19. A component is divided into a number of factory segments, reflecting that some parts of the model may be instantiated dynamically, as described in section 6.3.2.

5.3.2 Applying the Deltas at Startup: Configuration-Based Instantiation

The default mode of execution is to generate a Backbone configuration file for an extension directly from an Evolve model. This configuration represents the deltas that are to be applied to the system. The configuration can then be added to another configuration which represents the base to be extended, and at program startup time the DeltaEngine reads in, checks, flattens the extended composite and instantiates and runs the system.

The Evolve modelling tool allows a system to be executed in this manner, providing a convenient
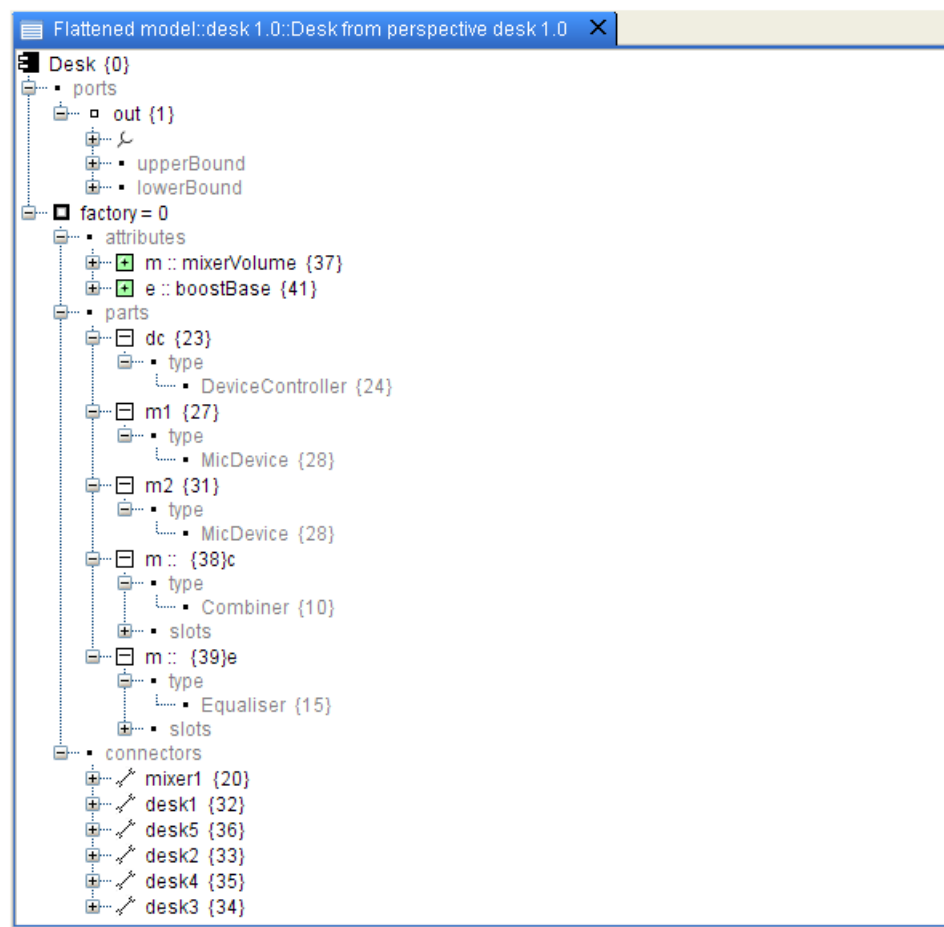
Figure 5.25: The flattened, unified Desk component

environment for a rapid edit-check-execute cycle. Figure 5.26 shows Evolve running a small Swing-based program.

The performance profile of this execution mode is interesting. Once instantiated and connected, there is little or no extra overhead involved in executing a system this way. It has the same performance profile as a conventional Java program expressed using classes and interfaces. However, component instantiation (including startup and dynamic component instantiation) is slower as this is done via Java reflection using the implementation class names from the configuration.

5.3.3   Applying the Deltas at Compile Time: Generating Code for a Configuration

The toolset is also able to generate a set of classes representing a flattened configuration. This has the benefit of improving the performance of component instantiation. An extended system compiled in this way has no extra overhead over a conventional Java program: no reflection or any other runtime techniques are necessary to produce a running system. Because composite components are purely structural entities, the classes corresponding to these are able to be generated completely from the Backbone architecture.
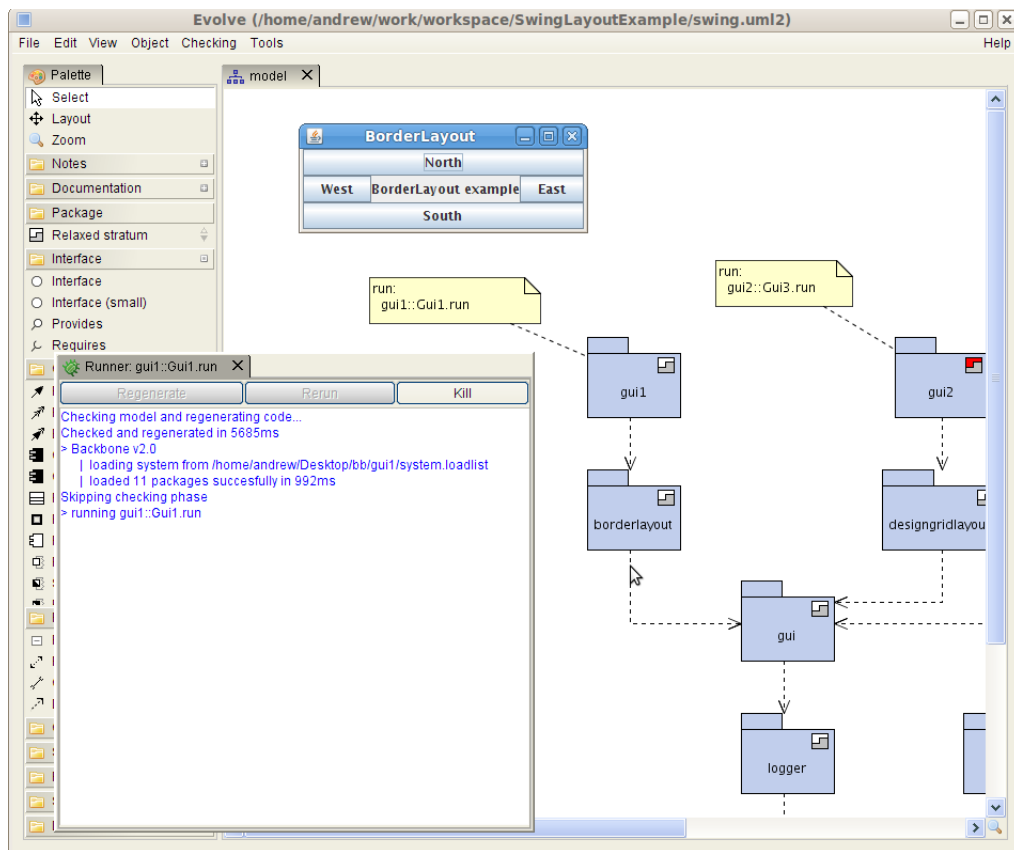
Figure 5.26: Executing a Backbone system inside Evolve

5.3.4 Interface and Primitive Type Evolution

As mentioned in section 4.3.2, interfaces are elements and can therefore be evolved using a combination of resemblance and replacement. The name, retirement status, operations and implementation class name constituents of an interface can be adjusted by an extension stratum in this way. As previously discussed, the evolution of components is able to be handled by the runtime, as it is responsible for all component instantiation. Interface evolution, however, is subtly different in that a change to a single interface can affect many components which provide or require that interface, and the interaction between the Backbone runtime and underlying platform runtime must be considered.

To examine the effects of interface evolution, suppose that developer X of the turntable extension in section 3.3.5 had decided to evolve the `IDevice` interface to add extra methods or change existing ones for cuing support. This would imply that any of the components that implemented this interface in the base stratum would no longer be correct, as they would not automatically provide the extra or changed methods. Further, if the evolved interface were not subtype compatible with the original definition, then all components in the base that required this interface would similarly be incorrect. This situation could be corrected by evolving any affected components in another stratum.

The Backbone runtime environment supports the evolution of the interface constituents of name, retirement status and operations. It does not, however, support the evolution of implementation class name. This limitation is related to how the Backbone runtime utilises the underlying Java platform

it is currently built on. Interface evolution is handled by placing the classpath entries for evolving stratum earlier than those of base stratum, as per the strata dependency graph. The Java runtime will then pick up the evolved definition, which must have exactly the same package / class name in order to shadow the earlier definition. This support, though, is contingent upon the way that the (unadorned) Java runtime works. The same implementation strategy would not work with Java and OSGi bundles (section 2.3.4), as classpath shadowing does not occur in the same way.

Primitive types need to be evolved using a similar mechanism. Primitive type instantiation is usually performed by component implementations, and cannot therefore easily be controlled by the Backbone runtime platform. As such, classpath shadowing must be used, which leads to the correct conclusion that the current runtime supports the evolution of primitive type constituents of name and retirement status, but not implementation class name.

5.4 Summary

The Backbone modelling tool and runtime environment are based on an implementation of the formal specification. A previous incarnation of the toolset used an informal description of resemblance and replacement, and this resulted in inconsistencies and unforeseen corner cases. The Backbone toolset demonstrates, in a compelling manner, the power and applicability of a formal specification to our modelling approach.

Evolve is the Backbone modelling tool, which uses UML2 composite structure diagrams for its graphical notation. It was built from the ground up to support the Backbone approach, and it records the deltas made to a base system by an extension, even though it allows the developer to design with fully expanded structures. This retains the benefits of the Backbone delta philosophy, whilst preserving a familiar approach to the construction and extension of a system. Evolve supports transitional models, which may contain errors, in a robust and intuitive manner. Evolve is methodology agnostic, and provides facilities to allow both top-down and bottom-up modelling. Strata, representing either extensions or the base itself, can be imported and exported to support the sharing and distribution of software that occurs in an extension setting.

An architecture expressed in Backbone can executed in one of two ways. A configuration for an extension can be generated from the architecture, and this can be combined with another configuration representing the base to extend a system at startup time. The performance of this approach is identical to an equivalent Java program once instantiated, although component instantiation is slower as it involves reflection. Alternatively, Java code representing the extended architecture can be generated directly from a model. This approach, although less flexible, suffers from no performance overhead relative to a conventional Java program. This facility is useful for developers producing variants of a system, as opposed to developing extensions to distribute and share with others.

Appendix E contains download instructions for obtaining the Evolve modelling tool, Backbone runtime and the example models used in this thesis.

Chapter 6

Advanced Modelling in Backbone

In this chapter we build on the foundation of the Backbone hierarchical component model and extensibility concepts, in order to demonstrate several advanced modelling techniques.

Firstly, we introduce a Backbone-specific interpretation of stereotypes as a convenience for automatically expanding the structure of a component. We also show how hyperports allow connections across different levels of the compositional hierarchy.

We then use these facilities to build component-oriented variants of several well known design patterns, with a strong emphasis on improved extensibility. This permits an extension to extend any patterns used in a base architecture, in a way which respects their intent.

6.1   Stereotypes as a Modelling Convenience

Stereotypes in Backbone are a way of tagging model elements, offering a classification mechanism distinct from that provided by a resemblance hierarchy. The stereotype that a component is tagged with can expand the component's structure, and alter its visual form in diagrams. This expansion facility offers a convenience that allows us to avoid the manual creation of parts and connectors in common situations. The importance of this facility is that it allows us to omit constituents that can be created automatically, thereby reducing the number of alterations that an extension must make.

Although UML2 stereotypes have a more complex technical definition than that of Backbone stereotypes (section 2.4.4), in practice these also tend to be used primarily for tagging purposes. The mapping between the Backbone and UML2 stereotype concepts is described in section 5.2.11.

The expansion performed by a stereotype occurs inside the DeltaEngine portion of the runtime. Constituents added in this way are not shown on diagrams, which allows us to avoid cluttering up the visual presentation.

Each Backbone component or interface can be tagged with at most one stereotype. This constraint was added to prevent issues regarding the order of expansion of multiple stereotypes. As the stereotype

applied to an element is modelled as a delta, it is inherited from resembled elements, and can be deleted or replaced.

Stereotypes in Backbone can have attributes. Tagging an element with a stereotype means that we can also supply values for these attributes, and these values are held with the element.

Autoconnect ports provide a simple, but very useful example of the use of stereotype expansion.

6.1.1 Autoconnect ports

An *autoconnect port* is automatically connected to any compatible, unbound ports of a component's parts.

The concept of port compatibility is described in section 4.3.2. Essentially, two ports are compatible when for each required interface on each port, the other port provides the same interface or a sub-interface. Only compatible ports can be legally joined together using a connector.

An autoconnect port can be used to make a service available to several parts, without explicitly requiring connectors to that service. Connectors for autoconnection are created by the «component» stereotype, which all Backbone components are tagged with upon creation.

Autoconnection is a convenience that allows us to omit common connectors that would otherwise clutter up both diagrams and the model. Connectors added by the expansion are not shown graphically. Regardless, they are full constituents and therefore influence port type inference and other features. Table 6.1 shows the autoconnect port graphical symbol.

Symbol	Backbone port type
▣	Autoconnect port

Table 6.1: The symbol for an autoconnect port

In the following example, we examine how autoconnection works. Figure 6.1(a) shows the definition of a `PrinterQueue` component that requires a printer service via the `printer` port. Each of the three parts in the component also require this service, and hence have direct connections to the port.

Figure 6.1(b) shows that we can achieve the same structure using an autoconnect port, freeing us from needing to create the connectors manually.

We can take the presentation a step further by visually eliding any ports which are autoconnected, giving figure 6.2. In general, it is assumed that an autoconnect port will be connected wherever needed, and we therefore purposely omit any ports of parts that will be connected in this way.
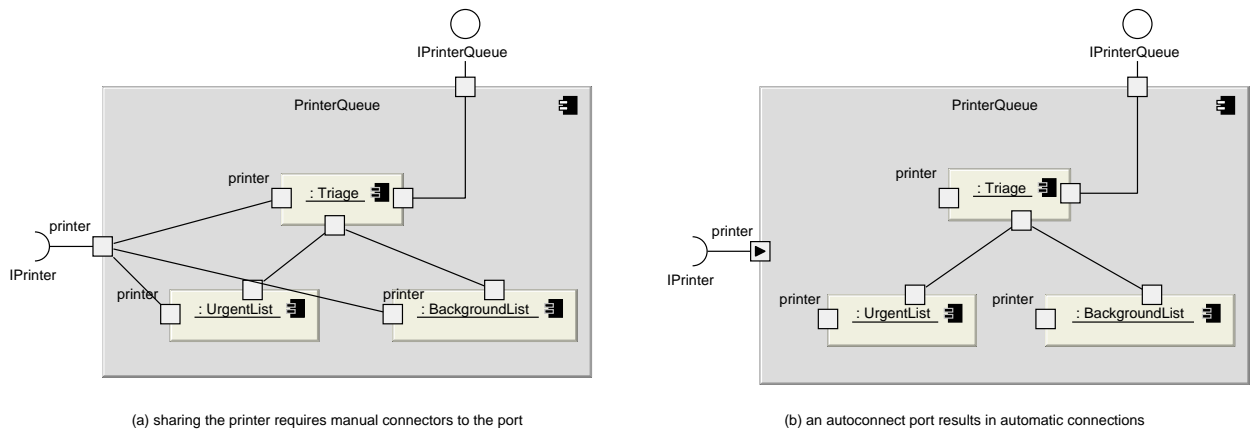
Figure 6.1: Autoconnect ports allow connections to be made automatically
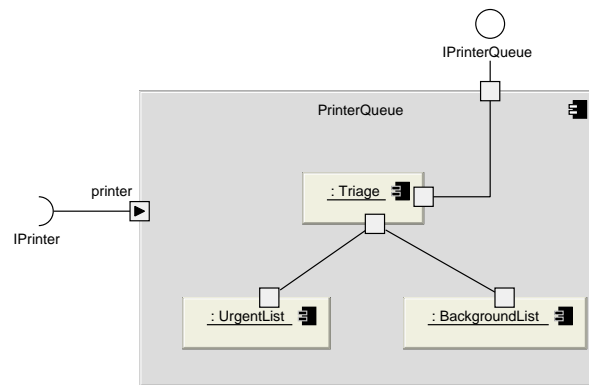
Figure 6.2: Visually eliding auto-connected ports

6.1.2 Summary

Stereotypes are a way to tag and classify elements, and each Backbone element may be tagged with a stereotype. The stereotype that an element is tagged with is able to expand that element's structure. Stereotypes also have a visual aspect, and can alter the presentation of an element. These facilities allow components to be shown with a customised appearance. Backbone currently has a fixed repertoire of stereotypes, although we plan to allow user defined stereotypes at a future time (section 8.3).

To prevent issues around order of expansion with multiple stereotypes, a Backbone element can have at most one stereotype applied to it. Stereotype application (or tagging) is handled using deltas, and the applied stereotype is inherited via resemblance and can be replaced or deleted if required.

One example of the use of stereotypes is for the addition of connectors for autoconnect ports. These ports are automatically connected to any compatible, but unbound ports of parts. This is a useful convenience which allows us to omit the connectors for commonly used services in a component, potentially reducing the number of changes an extension has to make.

Several other patterns and techniques listed in this chapter also make use of stereotypes.

6.2   Hyperports: Connecting Across the Composition Hierarchy

It is common to want to provide a service to an entire subtree of the composition hierarchy. *Hyperports* support this by automatically establishing connectors which cut through a hierarchy, allowing an extension to connect parts without having to create the intermediate connectors and ports to facilitate this.

Consider how this works: each *end hyperport* is automatically connected to at most one other compatible *start hyperport*, either at the same level or above it in the composition hierarchy. The choice of possible connections is decided on the basis of compositional proximity, and a connection is preferred if it minimises the distance in the hierarchy between the hyperports.

A start hyperport must be capable of accepting multiple connectors, as it represents a service that will be shared. If the start hyperport only provides interfaces, this is no problem as such a port can handle any number of connectors even if it has a multiplicity of `[1]`. If, however, the start hyperport also requires interfaces then it must have a multiplicity of `[0..*]`. Conversely, the end hyperport, must be optional (`[0..1]`) if it involves any required interfaces to reflect that there may not be a compatible hyperport above it in the hierarchy to make a connection with.

Table 6.2 shows the hyperport symbols.

Symbol	Backbone hyperport type
	Start hyperport
	End hyperport

Table 6.2: The hyperport symbols

Consider how hyperports might be used to share a printer service with all parts of the `PrinterQueue` component of figure 6.1(a). To represent the printer service, we define the `Printer` component in figure 6.3(a) as providing the `IPrinter` interface through a start hyperport. Figure 6.3(b) shows the definition of `Triage` that consumes the printer service via an optional end hyperport which requires `IPrinter`. The definitions of the `UrgentList` and `BackgroundList` components can be inferred from the definition of `Triage`.

(a) Printer provides its service via a start hyperport   (b) Triage consumes the printer service via an end hyperport
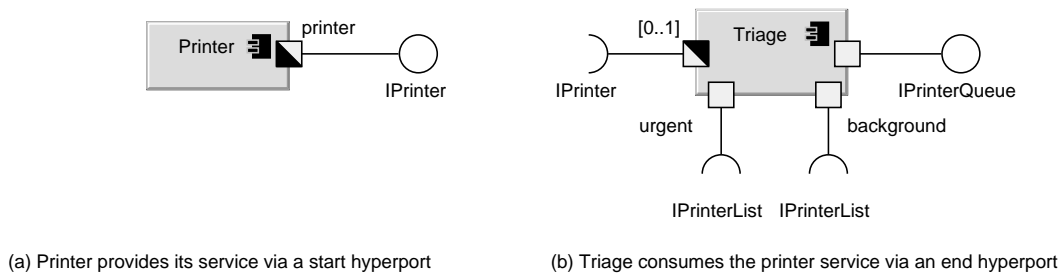
Figure 6.3: The hyperport definitions for the Printer and Triage components

We can now express the `PrinterQueue` component as shown in figure 6.4. We do not need to create

the `printer` port that was required in the definition of figure 6.1(a), as hyperports can cut through the hierarchy and we therefore have no use for this intermediate port.
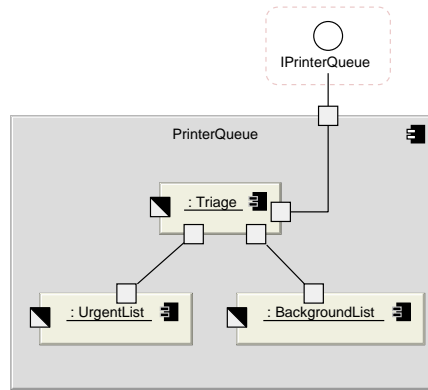
Figure 6.4: The PrinterQueue component

To complete the picture, we create the `Bureau` component of figure 6.5, which features two `PrinterQueue` parts and a single `Printer` part. The use of hyperports means that the single printer will be available throughout the hierarchy. This is shown in the composition hierarchy of figure 6.6, where each `Triage`, `UrgentList` and `BackgroundList` part is automatically connected to the start hyperport of the `Printer` part, bypassing the boundaries of the `PrinterQueue` parts.
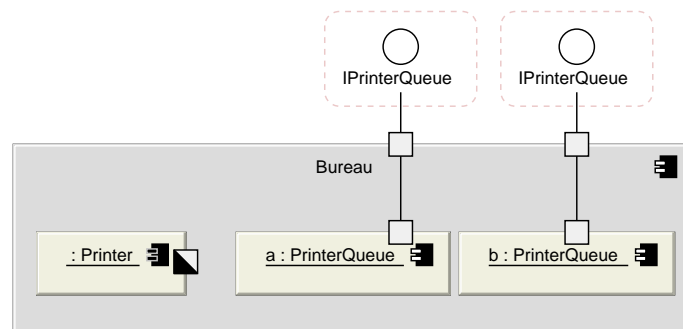
Figure 6.5: Autoconnection of the Printer part

Hyperports are handled inside the DeltaEngine by adding connectors to the flattened representation of the system. We can use Evolve to view the flattened structure as shown in figure 6.7. The hyperports have resulted in six connectors being automatically added. The first of these has been expanded, showing that it is between the `Printer` part and a `BackgroundList` part.

If a hyperport of a part is already bound, no automatic connections will be made to it. This allows us to also treat a hyperport of a part as a normal port by binding it with a connector.

Section 6.3.1 discusses the use of hyperports to provide an extensible and more flexible variant of the Singleton design pattern.
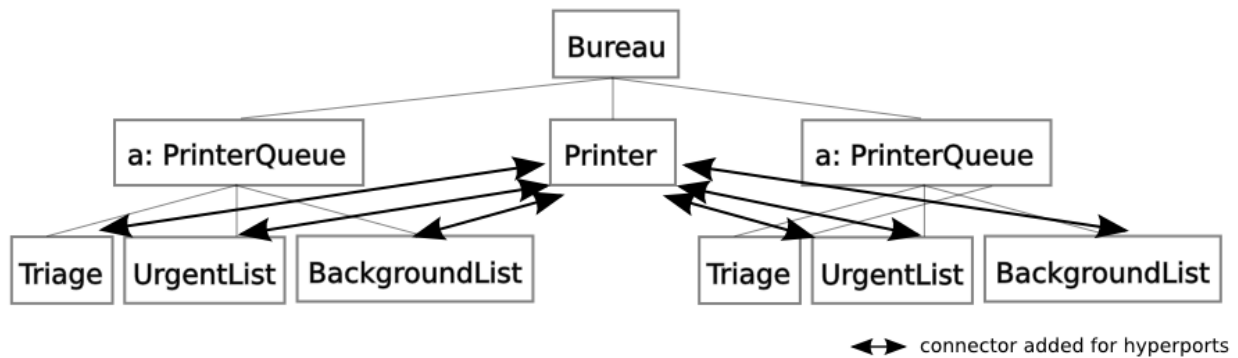
Figure 6.6: Hyperports result in connectors that cut through the composition hierarchy
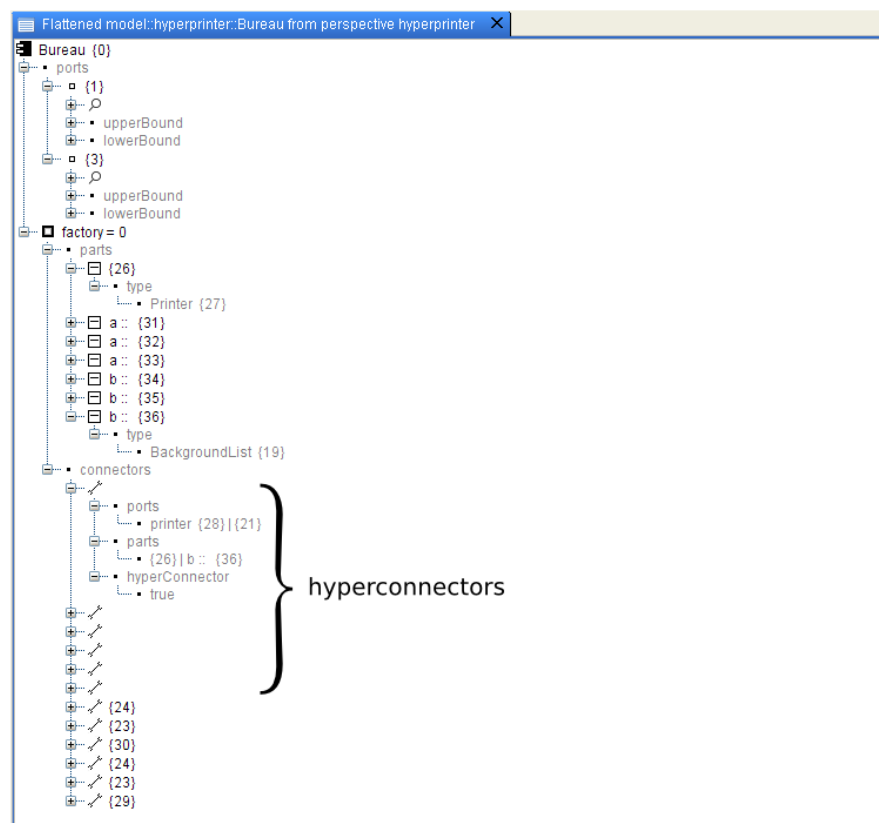
Figure 6.7: The flattened Bureau component, showing automatically added connectors

6.3   Design Patterns in Backbone

A software design pattern is a solution to a recurring design problem, in a particular context [Bud03].
The aim is to convey and record design knowledge, rather than to act as a precise implementation
guide. A well known catalogue of design patterns for object-oriented systems is found in [GHJJ95].
These patterns address the design of parts of an application, and their names have formed a common
lexicon for discussing the design intent in a system. Another consideration when using design patterns
is how multiple patterns can be combined in a coherent way to form larger designs.

In this section, we show how Backbone can be used to model component-oriented variants of several of the more common design patterns. We particularly focus also on improving the extensibility of these patterns, allowing an extension to alter a base pattern in a way which respects its intent. We also use the hierarchical model and extensibility constructs to mitigate some of the other limitations.

We address design patterns in this section, rather than architectural patterns [BMR⁺96] which instead address architecture and architectural style.

6.3.1 The Singleton Design Pattern

The Singleton pattern ensures that a class has only a single instance that is globally accessible, modelling a resource that is shared over a system. This is generally implemented by hiding the constructor of a class, and providing a single static `getInstance()` method which returns the single instance.

A Critique of the Object-Oriented Singleton Pattern

Any class using a singleton must contain a direct reference to it: the class is bound directly to the singleton via code reference. This restricts extensibility, as there can be no guarantee that only a single instance will be wanted in future iterations of a system. This also prevents easy unit testing of the class, as the singleton cannot be easily substituted for another in a test case.

Spring [JHA⁺05] pushes the description of singletons into configuration. A bean can be marked as a singleton in the configuration, and a reference from a bean to a singleton can be configured rather than being embedded in code. This resolves the unit testing issue, as it allows the bean to be configured differently for a test case. However, Spring configurations are not hierarchical and this limits the solution to a flattened architectural approach [McV09].

The Singleton Pattern in Backbone

Rumbaugh insightfully notes that a singleton is simply an object instance at a single point in the composition hierarchy of a system [Rum96]. The notion of singleton therefore turns into a single instance which is available to a subtree of the hierarchy. This effect can be achieved by the use of hyperports, as illustrated in figure 6.6. As such, Backbone uses a part with a start hyperport to represent the effect of a singleton. The benefit of our approach is that it fits well into the component and extensibility model: the singleton part can be added, moved or replaced using an extension. Further, singletons are marked in the configuration rather than in the implementation itself, allowing for unit testing via alternate configurations.

Consider how we might add a singleton logging service to the original `Desk` component of figure 3.12, whose compositional hierarchy is shown in figure 3.1. We start by defining the `Logger` component to represent the logging service with a start hyperport, as shown in figure 6.8(a). Suppose also that the

`Equaliser` component is evolved to use the logger, as in figure 6.8(b) where `ILogger` is required using an end hyperport.

(a) Logger provides ILogger via a start hyperport   (b) The Equaliser is evolved to require ILogger via an end hyperport
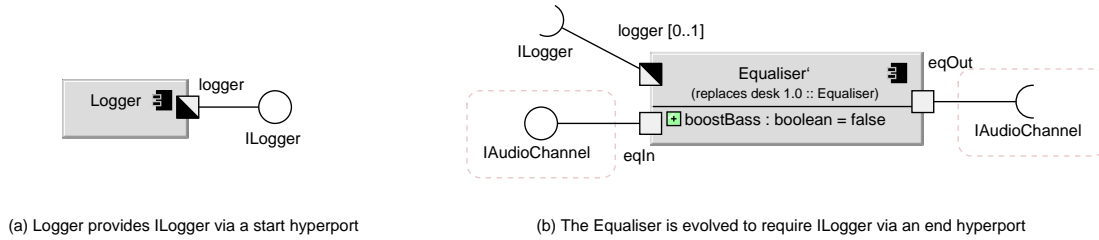
Figure 6.8: Providing and requiring a logger via hyperports

To make the logger available to the entire application, we use an extension to evolve the `Desk` component with an instance of the logger, as shown in figure 6.9. The logging service is now available to any part in the entire hierarchy which declares an end hyperport requiring `ILogger`, including the `Equaliser` part which is buried inside the mixer.
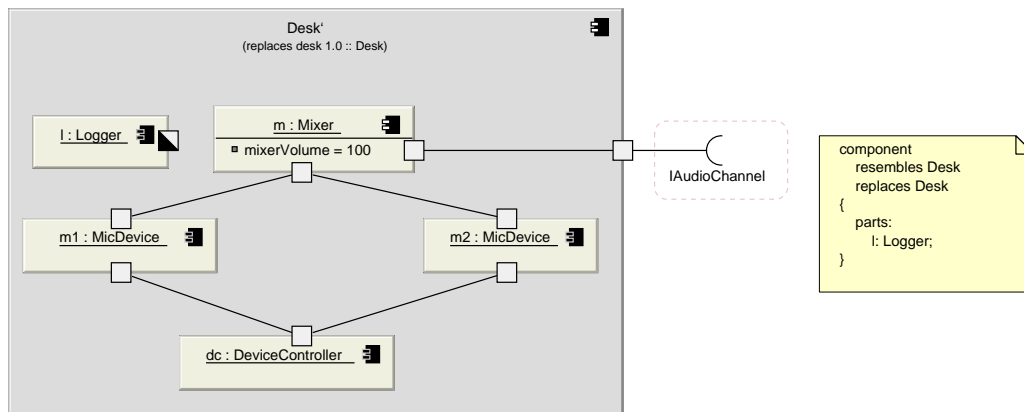
Figure 6.9: Evolving the Desk component to add a singleton logging service

Examining the composition hierarchy of the desk with the logger present (figure 6.10) shows how the hyperports have resulted in a connection which cuts across the `Mixer` compositional boundary.
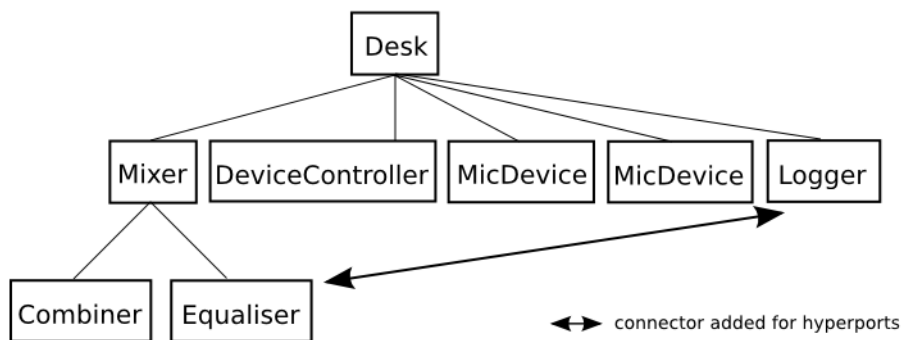
Figure 6.10: The composition hierarchy of Desk with the logger added

The hierarchical component model and extensibility constructs allow this to be a flexible arrangement. If at any point a single instance of `Logger` is not enough, then it is a simple matter for an extension

to evolve the `Desk` component such that multiple instances are present, with each potentially being pushed further down the composition hierarchy to make them available to different subtrees. Other architectural possibilities are having multiple loggers at different levels in the hierarchy, or explicitly connecting some parts to specific loggers rather than relying on hyperports. Figure 6.11 shows an evolution of the `Mixer` component of figure 3.15 which has instantiated its own private logger, connecting it directly to the `Equaliser` part's hyperport. This arrangement is compatible with the logger at the higher level of composition, superseding that arrangement for this component only.
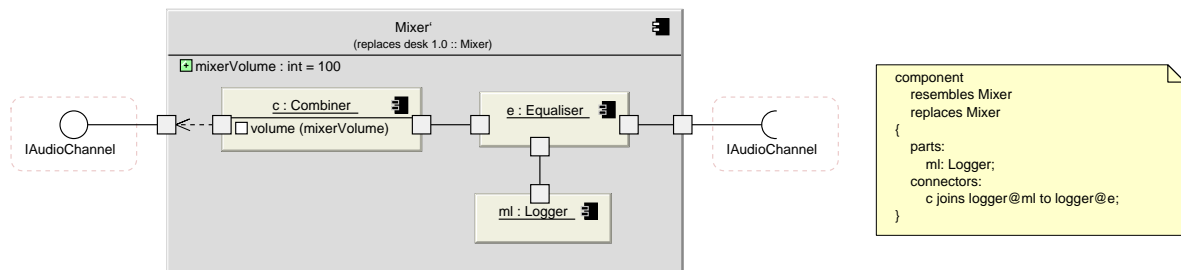
Figure 6.11: Adding a private logging service to the mixer

Note that hyperports can be used with ordinary connectors in which case the hyperport symbol is removed. This allows us to use a component with start or end hyperports as a conventional component if desired.

6.3.2 Isomorphic Factories for Dynamic Instantiation

We can dynamically instantiate sections of an architecture using *factory* components. As per the Darwin ADL, which pioneered the approach [CDF+95, DSE97], Backbone factories are also termed *isomorphic* as they have a similar port and connector "shape" to the set of components they instantiate. The factory construct encompasses several of the creational patterns described in [GHJJ95].

Factories are a type of composite component, which can be evolved and resembled using the Backbone extensibility constructs. This approach allows an extension to also alter the dynamic parts of a base architecture.

The insides of a factory are instantiated upon demand using the *create* port which every factory has as part of its structure. This instantiation can occur multiple times. The symbols for factory and the associated create port is shown in table 6.3.

Symbol	Backbone construct
◼	Isomorphic factory component
▣	Create port

Table 6.3: The symbols for the Backbone factory constructs

Consider how a factory works in the context of the `Desk` component example of figure 6.11. We wish to dynamically instantiate the two microphone devices. Firstly, we define a factory for the creation of

the two devices, as per figure 6.12.
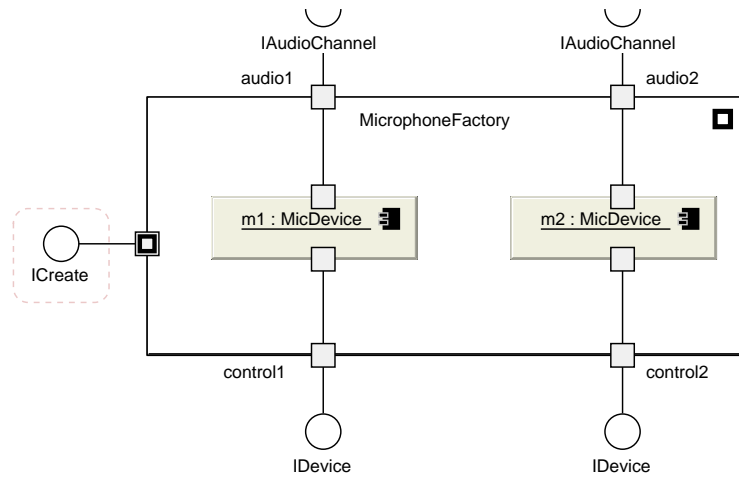
Figure 6.12: A factory for instantiating two microphone device parts

Note that the create port does not appear to be internally connected. In actuality, the «component» stereotype of the factory expands the structure by inserting a `Creator` part and a connector to the port: the added part handles the mechanics of instantiation and the connector results in the inferred type of the create port becoming the provision of `ICreate`. Each factory automatically resembles the Backbone `FactoryBase` component, from which it inherits the create port.

We need a further component which requires the `ICreate` interface, in order to programmatically request the instantiation of the factory. Figure 6.13 defines such a component, which (hypothetically) uses some form of input from the user to trigger an instantiation request.
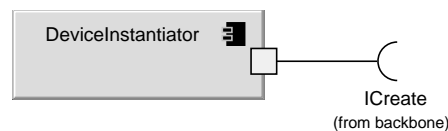
Figure 6.13: A component to trigger the factory instantiation

An extension can now evolve the original desk component to use this factory and instantiator, as shown in figure 6.14.

After the first instantiation, two microphone devices will be connected up. After the second instantiation, a total of four microphones will be present. Note that the connectors from the factory part are allowed to connect to the ordered port of the `DeviceController` part, even though they do not specify an index. In this case, the index will be allocated when instantiation occurs. It is allowable for such a connector to specify a fixed index, such as `[2]`, but then multiple instantiation will result in any previously created connections to that index being overwritten.

A factory can make use of the Backbone features available to other component types, including attributes, aliased slots, hyperports and autoconnect ports. This uniform treatment ties well into the
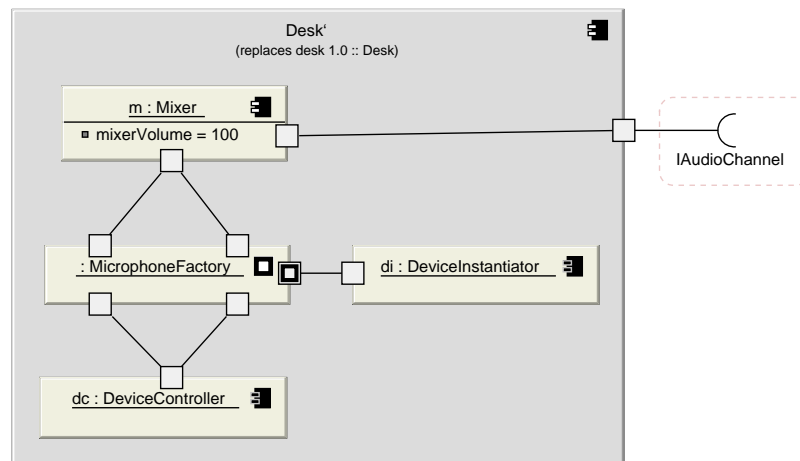
Figure 6.14: The Desk component evolved to dynamically instantiate microphones

Backbone extensibility approach, allowing an extension to resemble, reuse and evolve factories. We can also resemble a factory and use this to define a different kind of component, and vice versa. For instance, we can define a placeholder component which a set of factories resemble.

Factories and Creational Design Patterns

In effect, the internal architecture of a factory allows us to represent a variant of the Prototype design pattern [GHJJ95], where a prototypical instance is copied upon demand. In our case, the internal configuration of the factory is copied, and the use of aliased slots allows attributes to be shared between instantiations.

The Builder design pattern separates the construction of a complex object from its representation, allowing the same process to create potentially different structures. This can be achieved in our simple example by using a placeholder instead of the factory, as per figure 6.15. We can then use resemblance or evolution to create variants of the Desk component where the placeholder part is replaced with an appropriate factory part.

The components instantiated via a factory can also be destroyed. Upon creation, a handle is passed to the instantiating component This can be passed to the destroy() method of the create port to destroy the created structures and remove any connectors.

No restrictions exist with regards to the nesting of factories. A factory may further contain other factories and so on.

Factories in the Flattened Representation

The flattened representation of the Desk component with a factory is shown in figure 6.16. The flattened structure is divided into a number of factory sections. The top level Desk component is represented by flattened factory 0, which is instantiated when the application is started. The
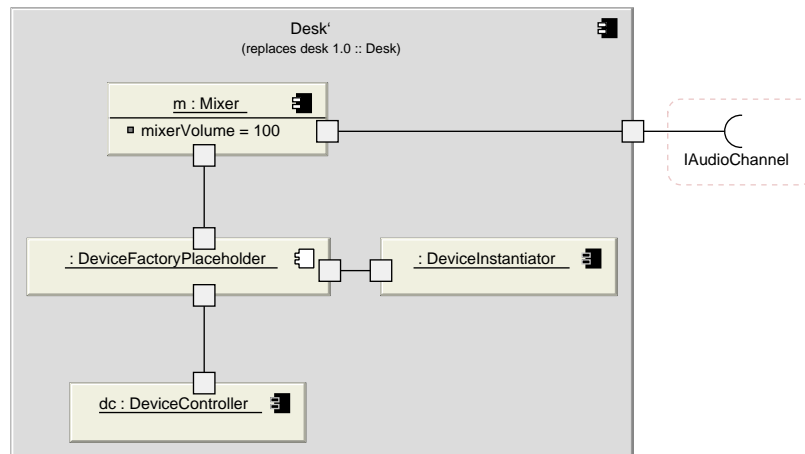
Figure 6.15: Evolving Desk to use a placeholder instead of a concrete factory

`MicrophoneFactory` component is represented by flattened factory 1. The `Creator` part, added for the `MicrophoneFactory` component, is propagated up to level 0 where it is used to instantiate flattened factory 1 on request.
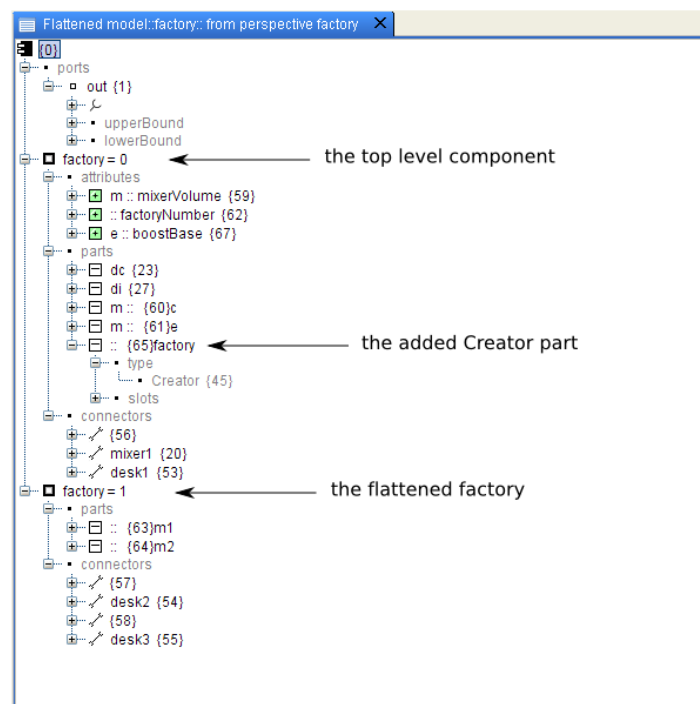
Figure 6.16: The flattened Desk with microphone factory

Factories within factories are handled by inserting the `Creator` part due to the expansion of the former, inside the latter.

6.3.3 Extensible State Machines

The State design pattern allows an object to change its behaviour when its internal state changes. This provides a primitive and lightweight form of a state machine.

We have built a variant of this pattern using Backbone components and stereotypes, as described below. This is more extensible than the object-oriented version in that it allows an extension to add or alter the events and also the states and transitions of a state machine in a base application. This in turn improves the extensibility of an application which uses state machines to express its logic.

A Critique of the Object-Oriented State Pattern

An example of the this pattern is shown in the UML class diagram of figure 6.17, where the state of an audio device is modelled. The three possible states (on, off, paused) are modelled as subclasses of `DeviceState`, and the `DeviceStateMachine` class represents the overall state machine, which holds a reference to the `current` state.

Events (on button, pause button) are modelled as operations on the `DeviceStateMachine` class. This class simply delegates any operation onto the current state. The current state object can effect a transition from one state to another by requesting that the `DeviceStateMachine` parent object switch to using another instance for `current`.
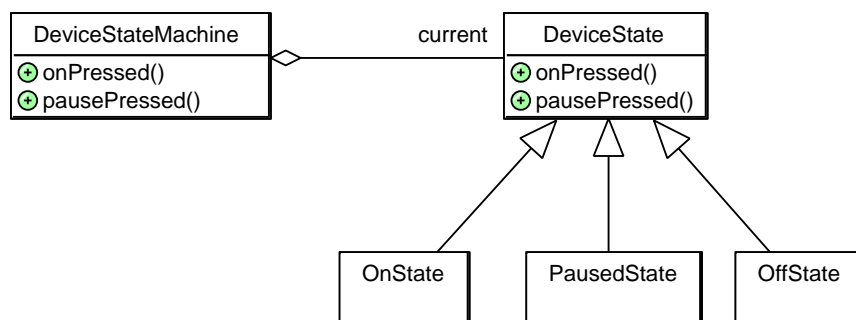
Figure 6.17: The State pattern for modelling device states

Although undoubtedly useful, this pattern presents many restrictions and unpleasant trade-offs which severely restrict extensibility. Firstly, all state transitions are hardcoded inside the individual state subclasses. As an example, consider that when an `OnState` instance receives an `onPressed()` event call, it wants to effect a transition to the off state. To do this, `OnState` creates an `OffState` instance and asks the `DeviceStateMachine` object to change to this as the next state. Both new state creation and transitions are embedded in code and therefore difficult to visualise and change.

Extending the state machine is problematic, whether we wish to add or change states, adjust transitions or simply add events. For instance, adding a cue state, to represent when a device's audio is sent to the cue bus, is difficult because we have to alter the transition logic inside the states. Adding an event is cumbersome because we need to add the new event method to every class.

The problems extend further to the context required by each state. A state requires a set of services and data from its surrounding environment: this is its context. As state creation is spread out and embedded in the other states, the connection between the state and its context is difficult to establish. To remedy this, it is usual that all services and data are available through a context object, which in our examples is the `DeviceStateMachine` instance. However, we need extend the context if any state requires another service, which makes this arrangement fragile. We effectively need to include the superset of all services and data required for all states in the context. This tends to bind the states together in dependence on a complex context object.

The tight coupling of states to their context (`DeviceStateMachine` in this case) prevents the reuse of the states in other machines and restricts extensibility. We can use a context interface to decouple in this case, but the state will still be bound to that interface which limits the scope for reuse.

The above limitations pose major problems when implementing anything but a trivial state machine, which has little or no context required by states. Nested state machines are difficult to build due to the need to propagate complex context information between nested levels.

The State Pattern in Backbone

Rather than present the final Backbone design straight away, we instead build towards this by explaining how the underlying mechanism works.

To model a naive representation of the device state machine, we first introduce the dispatcher component of figure 6.18. This holds the index of the current state port in an attribute called `currentStatePort`, and acts as a switch by directing any `IDevice` method calls to the currently selected `states` port. In this way, it acts to pass on the event to the currently selected state, as per the `DeviceStateMachine` class in figure 6.17.
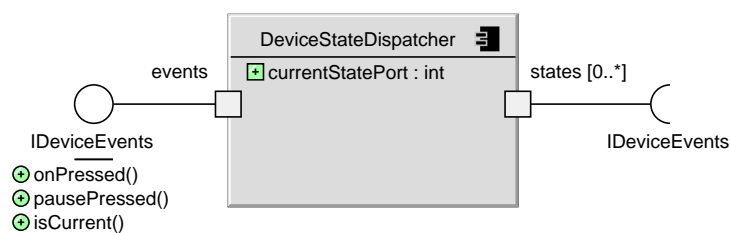
Figure 6.18: The component form of the device state machine

We then model each state as a component providing `IDeviceEvents`. We wish to use connectors to represent transitions between states, so we also provide each state with one `in` and several `out` transition ports. To effect a transition from the current state to another, the current state asks via an `out` port if the state at the other end of the connector is willing to be next. If it accepts, the next state indicates this by answering `true` to the `isCurrent()` method call. This is queried by the `DeviceStateDispatcher` component between events, and the next state is then promoted to the current state: its index is now stored in `currentStatePort`.

Figure 6.19 shows the definition of the `OnState` component. This has two `out` ports – one for when the on button is pressed, and another for when the pause button is pressed.
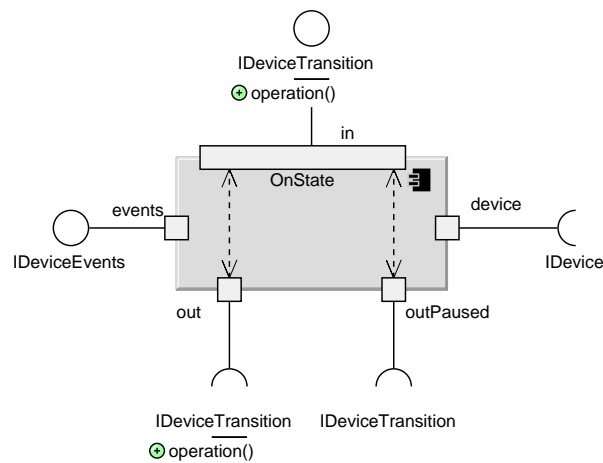
Figure 6.19: The OnState component models

To demonstrate how context can be provided to states, we have also made `OnState` require access to an audio device via its `device` port. A final point worth noting is that a port link is used between the `in` and `out` ports. Consequently if an `OnState` part is placed in an environment where the `out` ports are supplied with a subinterface of `IDeviceTransition`, then this interface will also be provided via the `in` port. This allows us to use more complex transition interfaces if required, without perturbing the existing state components. The `PausedState` and `OffState` components are defined in a similar way.

We can now connect instances of these states up into a composite component along with our state dispatcher part, using connectors to represent event paths and state transitions. This is shown in the somewhat busy component of figure 6.20, which details the complete state machine. All possible transitions are shown by connectors between `in` and `out` ports. To enter the state machine initially, we call into the `in` port of the state machine, which makes `OnState` active. This is the start state.

If we are in the on state, and the on button is pressed, then the `OnState` part will call through its `out` port and attempt a transition. This port is connected to the `OffState` part, and the transition to this state will then occur. If the on button is pushed again, `OffState` uses its `out` port to transition to the connected state which is `OnState`. Similar transitions exist for `PausedState`. Note that the transition protocol allows for a potential next state to refuse the transition. We can also loop from an `out` port back to the `in` port of the same state, expressing a self-transition.

`OffState` is the terminal state of this machine. Via the `outFinal` port, we can exit this machine if the pause button is pressed while we are in an off state. This can potentially pass control onto another state machine if it is connected up to the machine's `out` port. This is optional however, and if no further machine is available, then `OffState` will remain the current state but ignore all events indicating that the state machine has completed.

The `device` port of the `OnState` part has been connected to the `device` port of `DeviceState–`
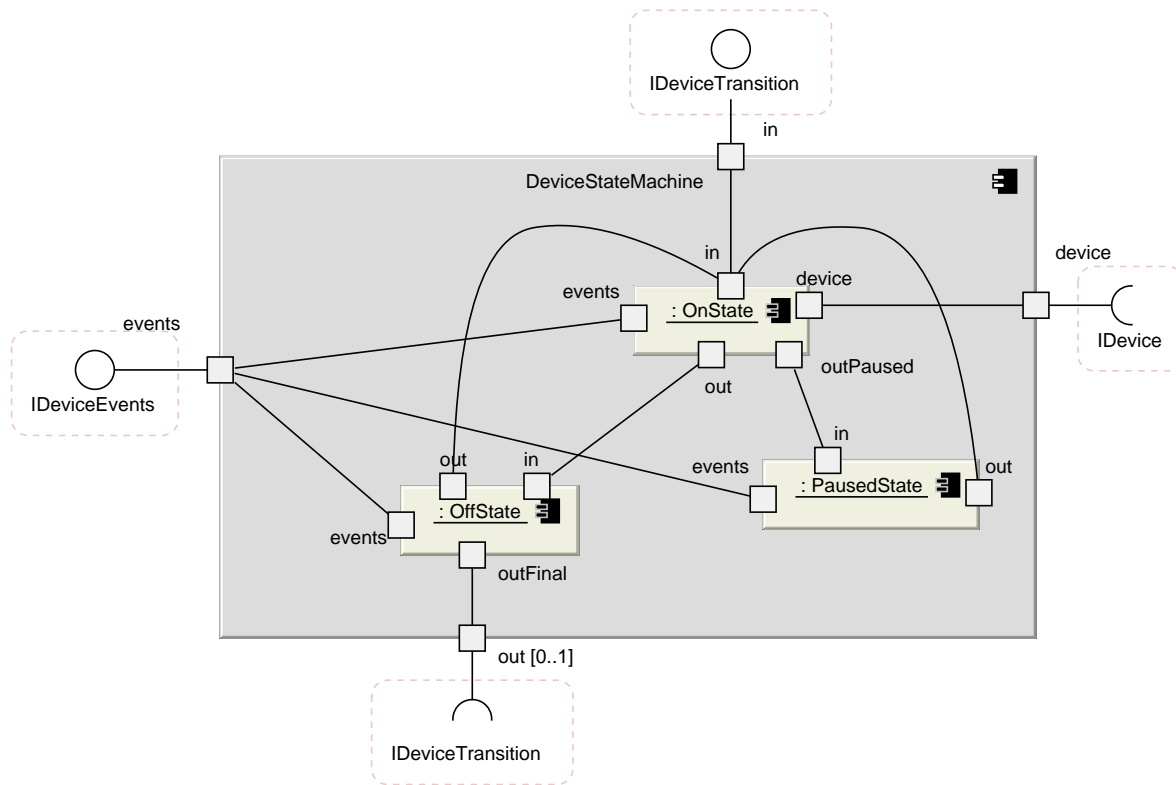
Figure 6.20: The full audio state machine as a composite component

`Machine`, thereby providing the required context for the state. We can deal with any further context required for additional states via evolution and extra connectors. No cumbersome context object is required as per the object-oriented variant.

Simplifying State Machines Using Stereotypes

The state machine of figure 6.20 has a number of connectors which obscure the actual transitions. Further, there is a lot of design overhead involved in creating the states and the state machine. To simplify this, we have created a state machine profile for Backbone using stereotypes.

If a component has the «state» stereotype applied, its visual appearance is altered to look like a UML2 state. Evolve contains a palette entry which creates a leaf component with this stereotype, inheriting its `in`, `out` and `events` ports from a base component. Defining the `OnState` component in this way, as shown in figure 6.21, is visually appealing and reduces the design effort considerably. Leaf states can be created with a minimal effort.

The «state» stereotype also aids in the design of composite states. The expansion rules for this stereotype automatically add a dispatcher part to a composite, connecting it up to any state parts present. The standard dispatcher component is defined using a port link between its ports, ensuring it can be reused for any state machines whose event interface resembles `IEvent`.

Evolve contains a palette entry for creating a composite state component with this stereotype, in-
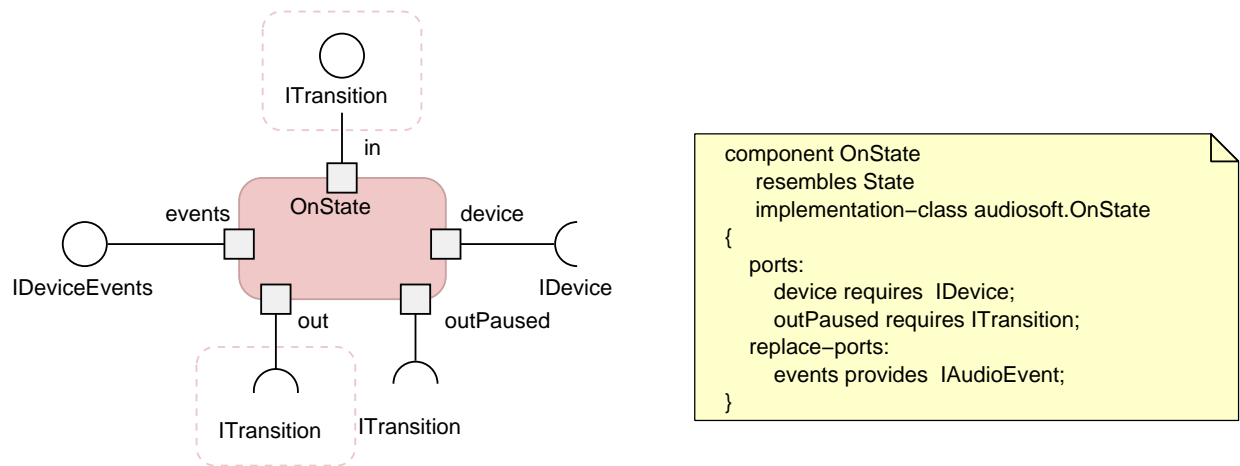
Figure 6.21: OnState defined using stereotypes and resemblance

heriting the `in`, `out` and `events` ports from a base component. It also inherits the start and end states, which are pass-through states with standard UML2 visual representations. Defining the `DeviceStateMachine` using this structure ensures that it visually resembles a UML2 state diagram, and that any non-transition connectors can be generally omitted. This is shown in figure 6.22. Note the use of the autoconnect port to connect `OnState`'s `device` port. We have named the connectors after the events that trigger them. The duplicated connector names are not a problem in practice as Backbone only considers the UUIDs of each element for logical identity.
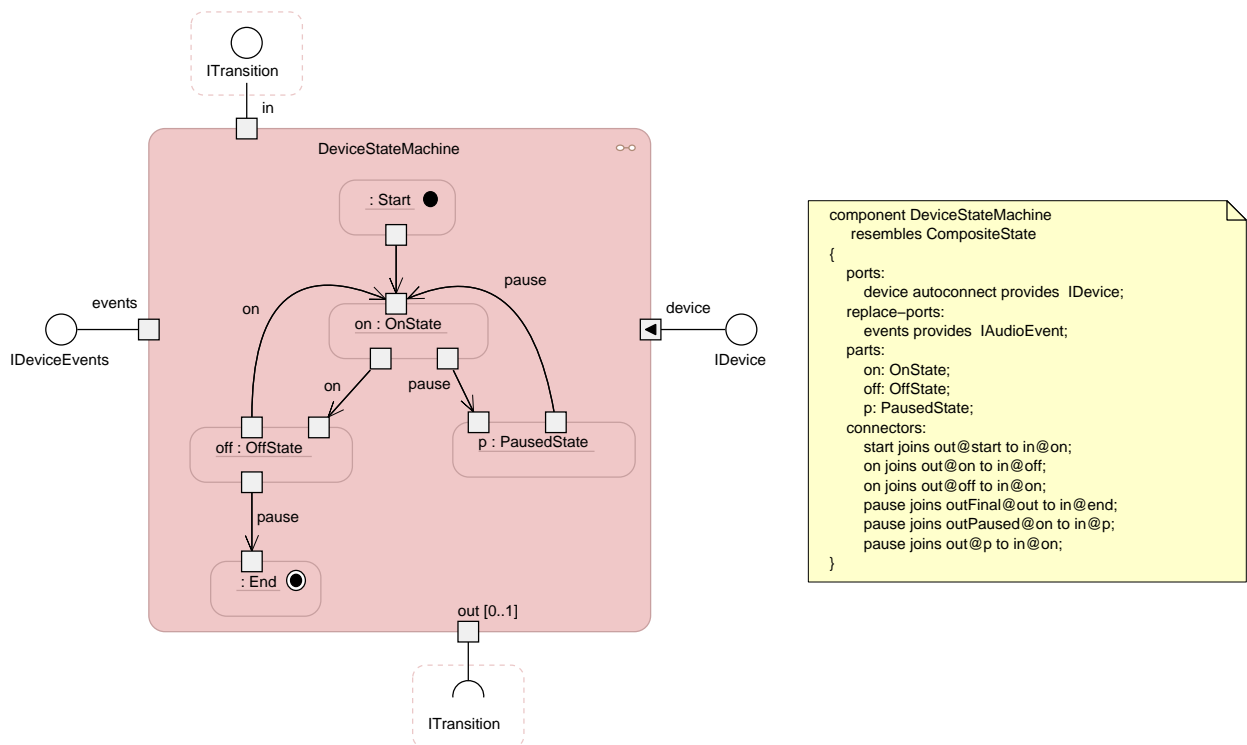
Figure 6.22: The full state machine using stereotypes and resemblance

Using the supplied stereotypes and base components means that the creation of state machines in

Backbone is visually appealing with low design overhead. Creating a complex state machine in Backbone requires a similar level of effort to creating the equivalent UML2 state diagram. Despite the visual similarities with UML2 state machines, the underlying structures are still Backbone components which can be used with resemblance and replacement. All of the advantages of our extensibility approach also therefore apply to state machines, as we will soon demonstrate.

Composing and Chaining Backbone State Machines

Backbone state machines can be nested and chained using the standard component mechanisms. Figure 6.23 shows a state machine which chains two nested `DeviceStateMachine` parts together. This composite machine will start in the `first` state machine, and then enter into `second` when `first` exits.
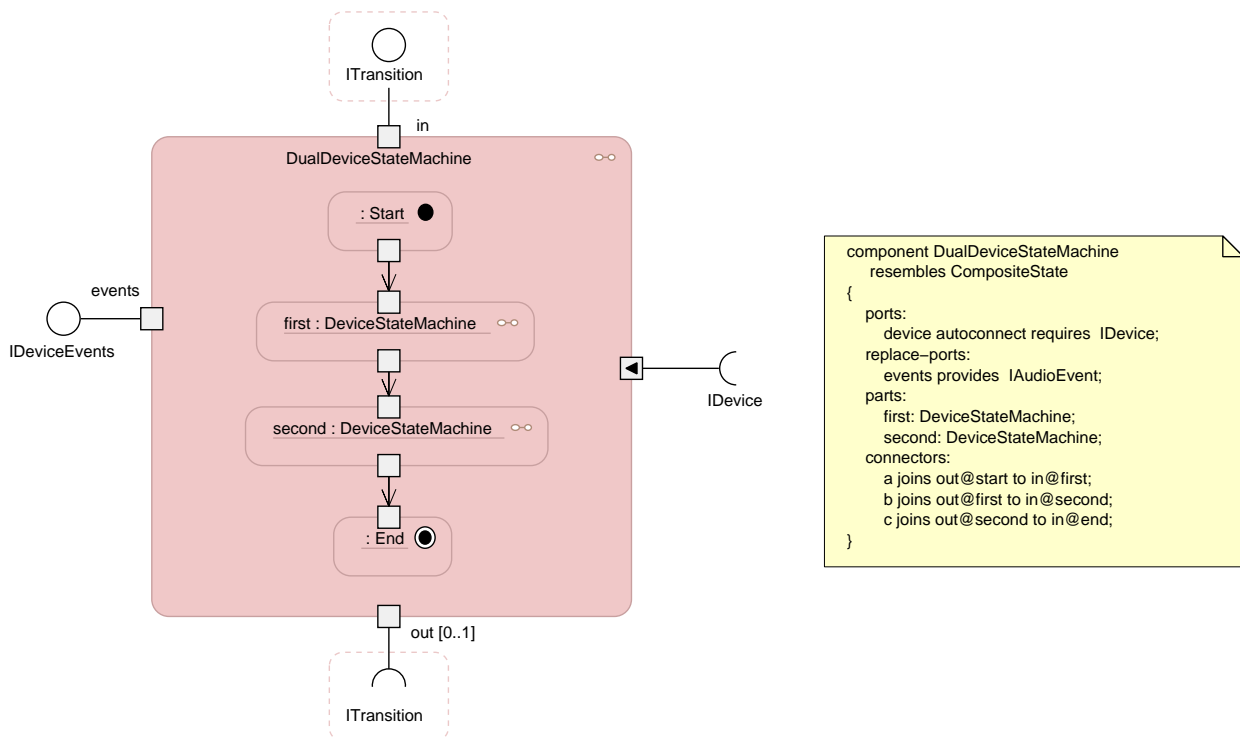
Figure 6.23: Nesting and chaining state machines

As they are components, it is possible to decompose states using standard Backbone techniques. We could, for instance, express a state using separate parts for entry and exit logic. This facilitates reuse via resemblance, or evolution when the logic requires adaptation or enhancement in an extension.

Extending a Backbone State Machine

An extension may wish to change a state machines in two ways. Firstly, states and transitions may be added or altered. Secondly, events may need to be added or altered.

Backbone state machines can be extended in either way. As state machines are components, they can participate in resemblance and replacement relationships. This, coupled with explicit transitions modelled as connectors, allows a state machine to be conveniently augmented.

By way of example, consider adding a cue state to `DeviceStateMachine` in an extension. Pressing the cue button in the off state results in a transition to the cue state, where audio is played through the cue bus. Pressing the on button in the cue state moves to the on state, where audio is played through the main audio bus. Pressing the cue button in the cue state moves back to the off state, turning off the audio. To incorporate this, we need to evolve the state machine to add the new state and any transitions, and also to add the extra cue button event.

Firstly, we define the `CueState` component, as per figure 6.24. Rather than evolve the `IDevice` interface to add the cue button event, which would require us to then evolve all other states to handle the upgrade, we have chosen to introduce a further event interface called `ICueEvent`.
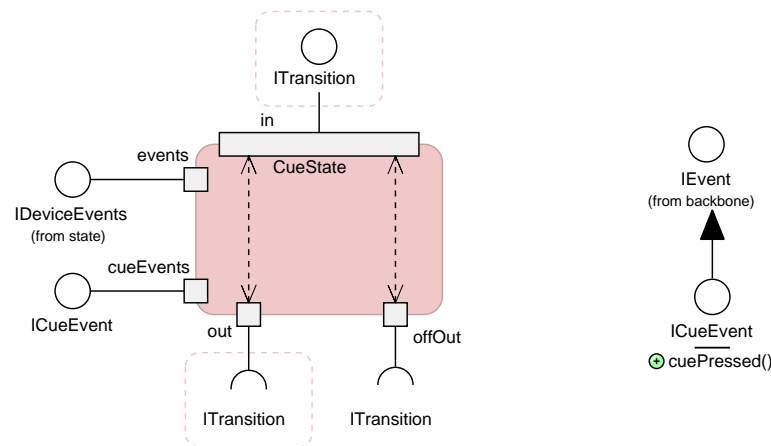
Figure 6.24: The CueState definition

Transitions to the cue state can occur from the off state. As such, we need to evolve the `OffState` component to handle cue button events via provision of the `ICueEvent` interface, and to allow a possible out transition via the `cueOut` port. This is shown in figure 6.25.

We can now evolve the `DeviceStateMachine` component to add in the transitions and the extra event interface as shown in figure 6.26. Note that the presence of a second event interface will cause the «state» stereotype to generate a second dispatcher part. The underlying design of the dispatcher component allows multiple of these to work correctly together.

Using an extension, we have been able to add an extra event and state with minimal impact. Only the implementation code for `OffState` required alteration, and this could be handled by defining a new class if the source code is not available to the extension. If, instead, `OffState` happened to be a composite, then incorporating the change would have been possible without source code via evolving and inserting an extra part into its definition.

In contrast, extending the object-oriented form of this pattern (figure 6.17) is not possible unless we are prepared to change the source code for every single class in an intrusive way. We would have to
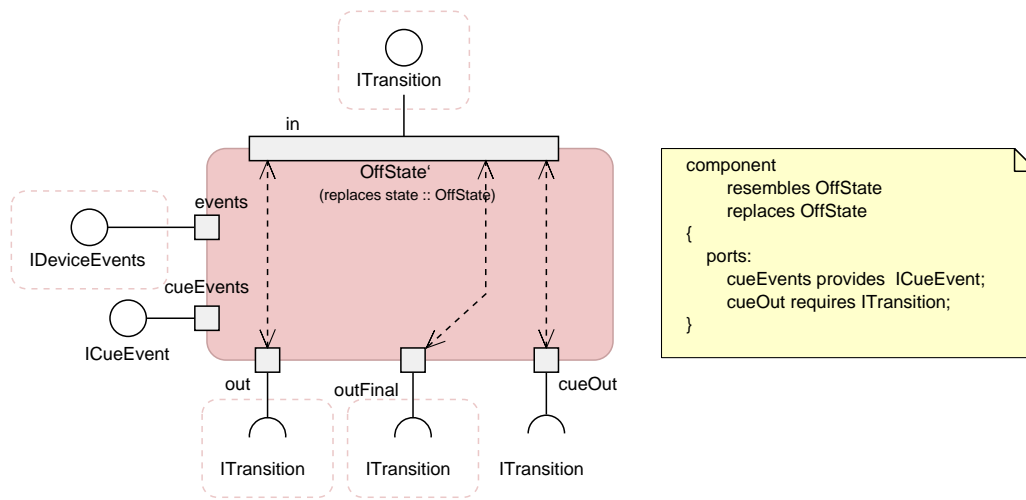
Figure 6.25: Evolving the OffState component to handle cue button events

adjust the hardcoded transitions encoded across the implementations, and alter the context class and every state class to handle the extra event method. In essence, the object-oriented form of this pattern is not readily extensible in either dimension. The Backbone form is naturally extensible because a composite state machine decomposes neatly into hierarchical components, which can be restructured using the extensibility constructs.
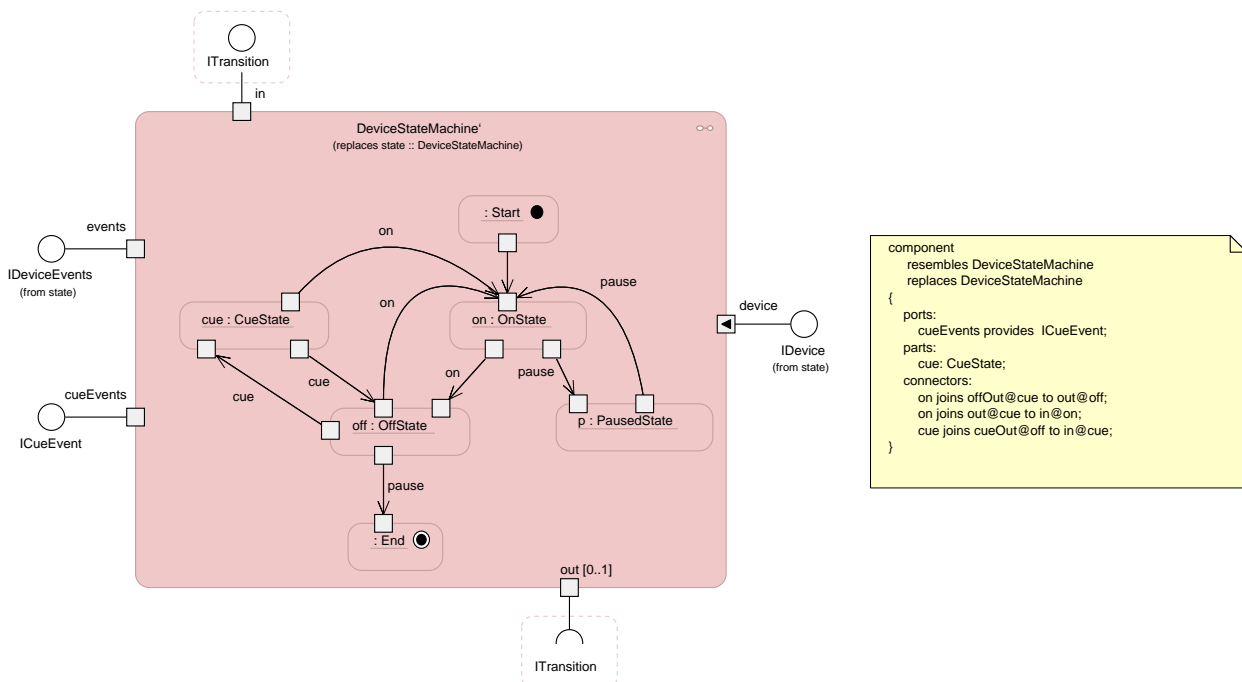
Figure 6.26: Adding the cue state to DeviceStateMachine involves minimal changes

Summary of Backbone State Machines

We have used stereotypes to allow Backbone to express state machines. Although these machines look visually similar to UML2 state machines, they are actually full component structures. This meshes well with the extensibility approach allowing a state machine to be augmented in an extension with the addition or customisation of states, transitions and events. Machines can also be nested and chained. In effect, our approach builds an elementary and extensible subset of nested statecharts [Har87] into Backbone.

Backbone component-based state machines have many advantages over the object-oriented State pattern. In particular, transitions are represented as connectors and are therefore explicit and not hard-coded within each state implementation. States and events can be added or altered easily, and handling the context requirements of each state is possible via ports and connectors. Resemblance of a state machine allows the reuse of potentially complex structures.

Both the ROOM ADL [SGW94a, SGW94b] and UML2 [Sel03] allow for state machine inheritance. Although these state machines are more expressive than their Backbone equivalent, neither facility offers the same extension possibilities. AHEAD [Bat06] provides a way to refine state machines expressed in a domain-specific language, although this has limitations related to how refinement works (section 7.2.3).

6.3.4 The Visitor Pattern

The Visitor pattern allows an "operation" to be performed on the elements of a complex object structure, by allowing the operation to visit each element of the structure in turn. The aim of this pattern is to allow further operations to be created without having to add logic to the classes representing the traversed object structure. The operation is also known as a *visitor*.

This pattern has several facets. Firstly, the traversal of the structure is encoded within the structure itself, allowing us to reuse this logic between many visitors. Secondly, the type information encoded within the structure is also managed by the elements of the structure, such that the visitor is informed of the type when it visits each element.

This pattern suffers from extensibility problems when adding a new type of element to the structure [ZO04]. This limitation can be ameliorated by our approach, which utilises components and hyperports, allowing an extension to add both new elements and new operations.

A Critique of the Object-Oriented Visitor Pattern

As per [GHJJ95], consider that we have a set of classes whose instances can be structured to represent a parse tree for a computer language. This is shown in the UML2 object diagram of figure 6.27. We wish to create one operation to perform a type check of these parse trees, and another to generate code from them. Further operations will be added in the future.
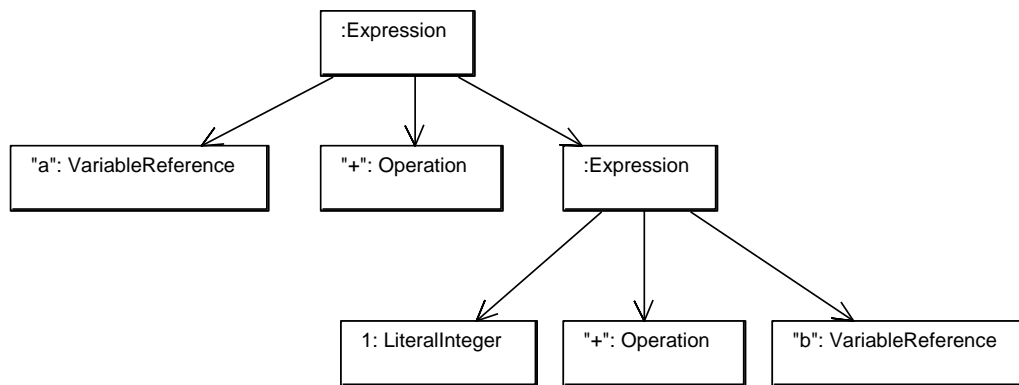
Figure 6.27: A parse tree

We could embed the logic for the type check and code generator directly into the classes making up this tree. However, this would lead to a problem when adding further operations, as we would have to place the new logic into every tree class.

The Visitor pattern addresses this by separating out the operation from the traversal of the structure. This allows a visitor object to be passed between all of the elements of the structure, leaving the structural knowledge within the tree. When an element receives a visitor, it calls back to provide it with information (and the type) of the current node. The visitor is then passed onto any children or related elements in the structure for further visiting.

To implement this we create an interface `IVisitor`, which any visitor must implement. Consider the following Java code which contains a method per element type:

```
interface IVisitor {
  void visitExpression(Expression e);
  void visitVariableReference(VariableReference v);
  void visit LiteralInteger(LiteralInteger l);
  void visitOperation(Operation o);
}
```

The code for handling traversal inside the `Expression` class then looks like the following, where `Term` is the superclass of both `LiteralInteger` and `VariableReference`:

```
class Expression {
  private Term t1;
  private Operation o;
  private Term t2;
  ...
  public void accept(Visitor v) {
    v.visitExpression(this);   // (1)
```

```
        t1.accept(v);              // (2)
        o.accept(v);
        t2.accept(v);
    }
}
```

Line (1) in the listing above provides a form of double-dispatch, where the visitor is given information and the type of the node it is currently visiting. The passing of the visitor further down the tree can be seen in (2).

Although this pattern can easily be extended for further operations by implementing a new visitor, it is brittle when faced with new element types. To extend this with a new type, we have to add a further method to the `IVisitor` interface, and update every element class to cope. The pattern is therefore extensible for adding operations, but not for adding element types.

The Visitor Pattern in Backbone

Using hyperports, we can make the connections between the visitor and the elements in the tree without requiring explicit connectors. This allows an extension to add both new element types and new operations.

Consider how the `Expression` component is defined in figure 6.28. The `visit` end hyperport provides `IExpression`, which is how the visitor accesses any services offered by this element type. The port requires `IVisitor`, which is the interface the visitor provides. This hyperport will be connected up to start hyperport of the visitor, allowing traversal to take place.

The extra ports allow an expression to have structure: each expression is made up of a term followed by an operation and another term. The `LiteralInteger`, `Operation` and `VariableReference` components are defined similarly, except that they provide their own interface via their own `visit` port.
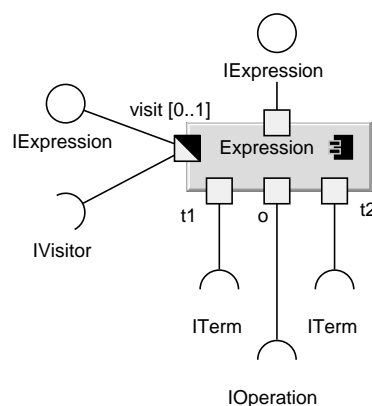
Figure 6.28: The Expression component

We now define the visitor component in figure 6.29. This has a start hyperport for every element component type, mirroring the visit ports provided by each element.
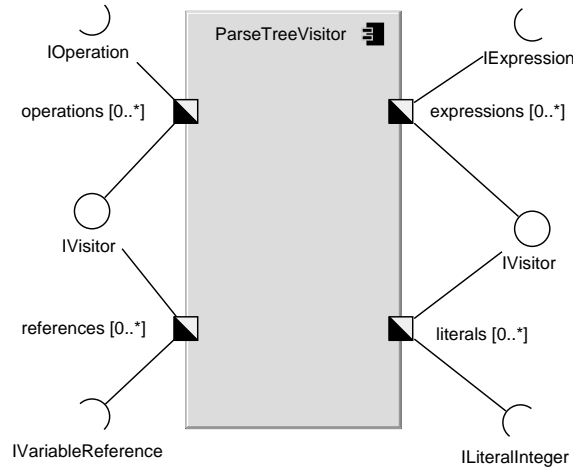
Figure 6.29: The parse tree visitor component

Finally, we can add all of these definitions together into a composite, as shown in figure 6.30. The configured structure of the parse tree is the same as that in figure 6.27. The end hyperports of each element part are automatically connected to the appropriate hyperports of the visitor.
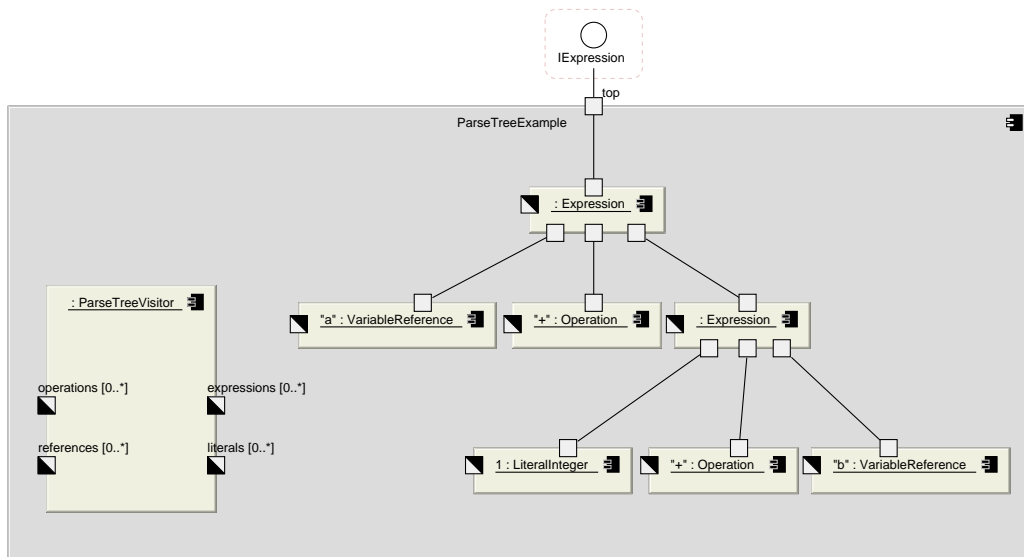
Figure 6.30: Visiting a parse tree using component structures

Consider how visiting occurs in Backbone variant. To start the visit going, we use the `top` port, which is connected to the first `Expression` part. This part uses its end hyperport to inform the visitor that this element is being visited. The visitor is then able to use the same connector between the hyperports to invoke any required operations on the `IExpression` interface provided by the expression. Control then returns to the `Expression` part, and it then uses its connectors linking it to the other parts of the parse tree to delegate visiting down the tree.

Extending the Backbone Visitor Pattern

An extension can add a further element type by defining a new element interface and evolving the visitor component to add a further start hyperport. Note that if we do add a further element type (say an `Assignment` element), then any existing visitor without a hyperport for this will still work. However, the visitor will not be told about the newer element type when visiting occurs.

Adding a further visitor type can be accomplished in a number of ways. For instance, we could make the visitor part a placeholder, and replace it with the correct operation in an extension. Alternatively, we could use factories to instantiate the required operation on demand. Upon instantiation, the hyperports will be connected and visiting can then commence.

6.3.5   Other Design Patterns

A number of other design patterns have interesting variations in Backbone which provide improved extensibility. For instance, the Adapter pattern is used to wrap an existing class and present the interface that clients expect. In Backbone, an extension can instead evolve a composite component and insert the adapting part directly into the structure, allowing us to work with the existing component rather than a wrapper.

The Strategy pattern, which allows for interchangeable algorithms can be modelled in Backbone using a subset of the extensible state machine facilities.

6.3.6   Summary

The Backbone component model allows us to create more extensible (and versatile) counterparts of common object-oriented design patterns. In several cases, hidden artifacts and structures are made explicit when expressing the patterns using components. For example, state transitions (which are implicit in the State design pattern) become explicit connectors when modelled in Backbone. An extension can then use evolution to add new states, transitions and events with minimal effort.

Others patterns are equally at home in the Backbone component model. Singletons, which are problematic constructs in an object-oriented setting, fit naturally into the notion of providing a service to a subtree of a component composition hierarchy. An extension can replace the singleton part, shift its location in the composition hierarchy, or add direct connectors to parts requiring singletons. This pattern is therefore extensible, unlike its object-oriented counterpart.

Similar extensibility advantages were found with the Backbone visitor pattern which, unlike the object-oriented variant, can be extended for both operations and node types.

6.4   Summary

This chapter has demonstrated a number of advanced modelling techniques, which are built on the foundation of the Backbone hierarchical component model and extensibility approach.

We introduced stereotypes as a convenient way of automatically expanding the structure of elements. Stereotypes can also customise the appearance of the components that they are applied to. One use of stereotypes is for autoconnection, where common connectors are created automatically. The use of these facilities means that an extension often has to alter less of the architectural structure in order to achieve its intended effect.

We also introduced hyperports, which are a way of connecting across compositional boundaries. Using hyperports an extension can flexibly make a service available to a subtree of the compositional hierarchy.

These techniques were combined to show how Backbone can model a number of common design patterns. We demonstrated that the Backbone variants are more extensible, and make the structure of the solution more explicit than their object-oriented counterparts. For instance, Backbone state machines allow the addition and alteration of states, events and transitions via an extension. Backbone singletons utilise hyperports, and have the advantage of not tightly coupling clients to the singleton implementation. This facilitates unit testing. It also allows an extension to replace the singleton, or move it around the hierarchy.

Backbone also provides isomorphic factories, which allow the dynamic instantiation of parts of the architecture. This facility can be used to represent several of the creational patterns, with the advantage that the connections between the static and dynamic parts of the architecture can be explicitly represented. Factories are also composite components and can be evolved, allowing an extension to alter the dynamically allocated structures.

By building on the robust foundation of the Backbone model, we are able to bring greater flexibility and extensibility to common design solutions.

Chapter 7

Evaluation

In this chapter we evaluate Backbone against other techniques for creating extensible systems and system variants. In particular, we compare and contrast Backbone to an advanced plugin system and assess how well each approach handles the same extension scenario. We then turn our attention to a compositional approach for producing product lines, and model the desk scenario from section 3.3.

In the previous chapter, we showed how Backbone could be used to produce more extensible variants of common object-oriented patterns and idioms. We now further demonstrate the applicability of our approach to other environments by showing how Backbone can also be used to construct web-based user interfaces. The intention is not to propose Backbone as a superior way to construct user interfaces, but to demonstrate the flexibility and applicability of the extensibility approach even in domains which are conventionally regarded as a natural "home" for objects. We also make the point that Backbone can fully leverage existing investment in object-oriented libraries.

Backbone naturally builds extensibility into an architecture as the system is decomposed hierarchically into fine-grained components. This approach works well if a system is designed using Backbone from the start. However, it is often desirable to retrofit an extensibility architecture into a mature system. In the last section of this chapter, we look at restructuring a complex, mature system using Backbone in order to extend it. We consider the advantages and limitations of the approach in this context.

7.1   Backbone versus Plugin Architectures

In this section we compare and contrast Backbone to plugin architectures, using a common extension scenario.

Plugins are pre-packaged units of software that can be "plugged into" a base application to extend its functionality, as described in section 2.1.3. This architectural style offers a powerful (and very popular) way of structuring an extensible application. One of the best known applications in this style is the Eclipse integrated development environment [Obj09f], which places all significant functionality, outside of the plugin system itself, into plugins.

We examine the characteristics of plugin architectures in detail, and consider how well they meet the extensibility requirements listed in section 1.3. To assess how these architectures cope with the different facets of extension, we extend a part of Eclipse. By also modelling the same extension in Backbone model, we are able to compare and contrast the two approaches. We show that our approach encourages a more flexible and granular architecture, with the advantage that the cost of introducing an extension is more closely aligned to the size of the change required to the architecture.

7.1.1 The Plugin Approach

A plugin architecture is structured around a base application with predefined extension points where the application can be extended. Developers create plugins, which "plug into" the extension points, and these plugins can then be selectively added to an installation of the application to customise it. The base application acts as a platform, providing a substrate for a family of applications. The basic concepts can be succinctly described using a simple design pattern [MMS03]. The notion of a set of plugins is loosely analogous to the Backbone extension concept, in that both are added to a base application in order to extend and customise it.

Plugin architectures work at two levels: the addition of new plugins (level 1), and the upgrade of existing plugins (level 2).

The addition of new plugins to a system (level 1) allows simple additive change, where the added plugins hook into extension points defined already in the base. Plugins can be created by developers who are not affiliated with the creators of the original application. The original creators are then free to concentrate on the base platform without having to continually expand the system to cater for every requirement.

The upgrade of existing plugins (level 2) is needed when the base application (including plugins already added) do not provide the required extension points. To allow this, advanced plugin architectures utilise a versioning scheme that allows for plugin evolution. Adding a new version of a plugin to a system may result in the previous version being completely replaced, or a situation where more than one version is instantiated simultaneously.

7.1.2 The Eclipse Plugin Architecture

At the heart of Eclipse is an advanced plugin architecture, which allows plugins to be versioned and also to have extension points themselves. This allows plugins to be extended by further plugins. Each plugin is an OSGi [Ecl09a] bundle which must fully specify any other plugins that it depends upon, allowing a plugin to be reliably added to a system knowing that all the dependencies are satisfied. The Eclipse plugin system is mature and production-tested, and has supported the creation and evolution of a number of prominent applications including the rejuvenated Rational modelling tool family [IBM09b, IBM09a]. An impressive number of applications use the Eclipse Rich Client Platform, which is also based on the same plugin approach [Ecl09b].

In terms of extension capabilities, the Eclipse plugin architecture represents a superset of other plugin platforms we have investigated. Therefore, the insights we can gather by analysing this system also apply, to a greater or lesser degree, to other plugin systems. As such, we study the Eclipse plugin model in detail in this section in order to compare and contrast it to the Backbone approach. Eclipse version 3.3 was used for the analysis and evaluation in this chapter.

Figure 7.1 expands on the basic plugin design pattern of [MMS03] to present a UML class diagram of the Eclipse plugin model.
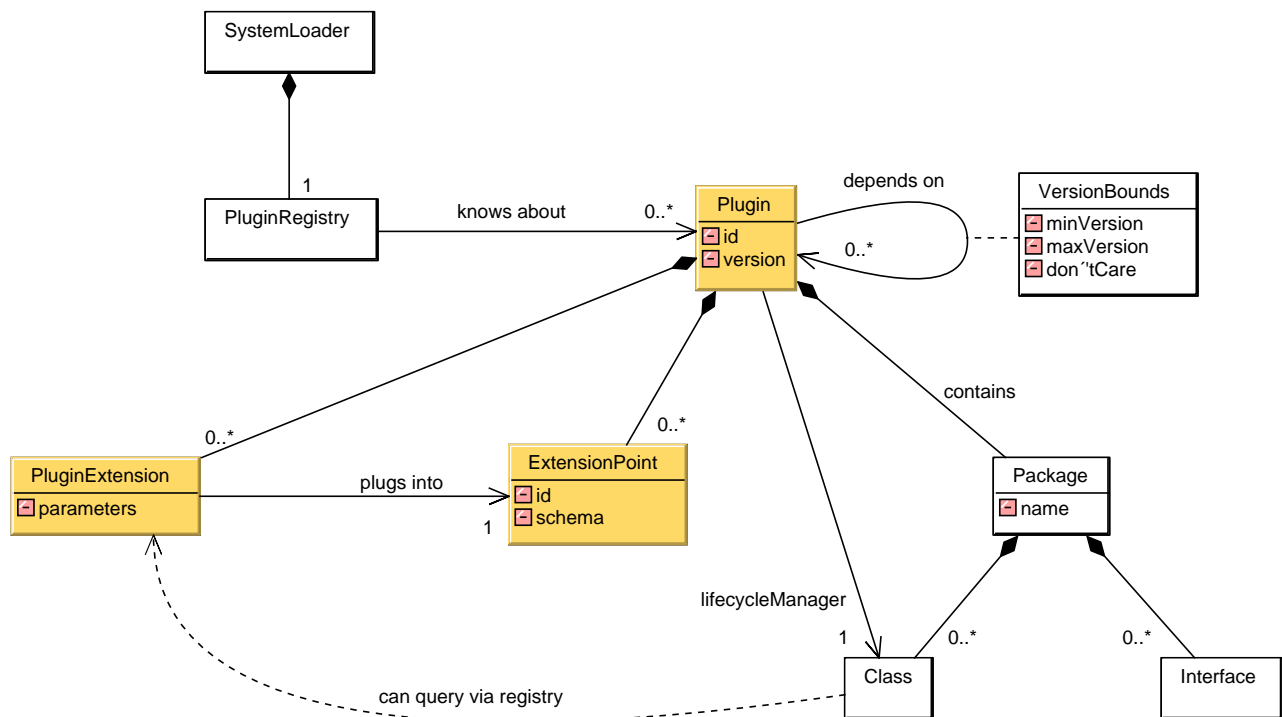
Figure 7.1: The plugin model

`SystemLoader` is the class which bootstraps the initial system, discovers any `Plugin`s and registers them with the `PluginRegistry`. The registry knows about all `Plugin`s, and can be used to query for a specific `Plugin` via the `id` and `version` attributes.

A `Plugin` is a versioned entity that contains packages, classes and interfaces. A `Plugin` may depend on other `Plugin`s, for extension points, services and library classes. The `VersionBounds` association class shows that this dependency can be expressed as a reference to a specific `Plugin` version, a bounded set of versions, or no version in particular (`don'tCare`).

A `Plugin` may provide a set of `PluginExtension`s[1] which "plug into" the `ExtensionPoint`s of other `Plugin`s. A `PluginExtension` uniquely identifies the relevant `ExtensionPoint` by its identifier (`id`). A `Plugin` may further define its own set of `ExtensionPoint`s, each of which

---

[1]In Eclipse, the actual terminology for a PluginExtension is "extension". We have used PluginExtension to differentiate this from the Backbone extension concept.

declares which parameters must be supplied (conforming to `schema`) by a `PluginExtension`. The model allows an `ExtensionPoint` to accept multiple `PluginExtensions`, although each `PluginExtension` can only plug into one `ExtensionPoint`. The motivation is that if an `Extension-Point` could only accommodate a single `PluginExtension`, then multiple `Plugins` could all try to fill the point, and conflict structurally[2].

At runtime, the `Plugins` are discovered and registered, and the `PluginExtensions` matched up to `ExtensionPoints`. Control is then passed to a distinguished `Plugin` for bootstrapping. Each `Plugin` is able to query the `PluginExtensions` which extend its points, and the parameters passed can include class names (for object instantiation) and values.

Normally the latest version of a plugin is instantiated. The exception to this is when other plugins explicitly indicate their dependence on an older version. In this case we can arrive at a situation where different versions of a single plugin must be loaded simultaneously. Eclipse outlaws the instantiation of multiple versions if the `Plugin` contributes to any extension points.

The extensibility of the approach arises because `Plugins` do not need to know what `PluginExtensions` will be provided for their extension points until runtime. The actual `PluginExtensions` for an extension point is a function of how many extending `Plugins` are discovered in the environment.

An `ExtensionPoint` is equivalent to a Backbone port with a required interface and `[0..*]` multiplicity, and a `PluginExtension` is equivalent to a port with a provided interface. Rather than model these concepts directly using interfaces, however, the Eclipse model expresses the data required and provided via metadata (XML files). Some of this data can refer directly to class names, and a `Plugin` can choose to instantiate an object based on a class name passed to its extension point.

7.1.3   Extending Eclipse Using Plugins: Adding a Column to the Task View

As a case study, we chose to enhance a small aspect of the Eclipse task view. As shown in figure 7.2, this displays a list of tasks along with certain columns. Consider that we wish to *insert* a further column "Assigned to" between the "Path" and "Location" columns. Our new column is designed to show which person has been allocated a particular task. We chose this scenario based on a belief that it represented a relatively simple and reasonable type of extension to the functionality of Eclipse. To prevent bias, at the time of choosing the scenario we had no knowledge of the underlying implementation structures.

The first step in making our alteration was to find the plugin responsible for viewing tasks, and see if an extension point existed for registering extra columns and controlling the order they are shown in. Finding the plugin proved to be straight forward: a class called `TaskView` existed in the *org.eclipse.ui.ide* plugin. Unfortunately, looking at the source code for this class showed that the set of columns was hard-coded.

---

[2]Eclipse does actually allow an `ExtensionPoint` to be defined with single multiplicity, but this is discouraged for the reason indicated in the text.
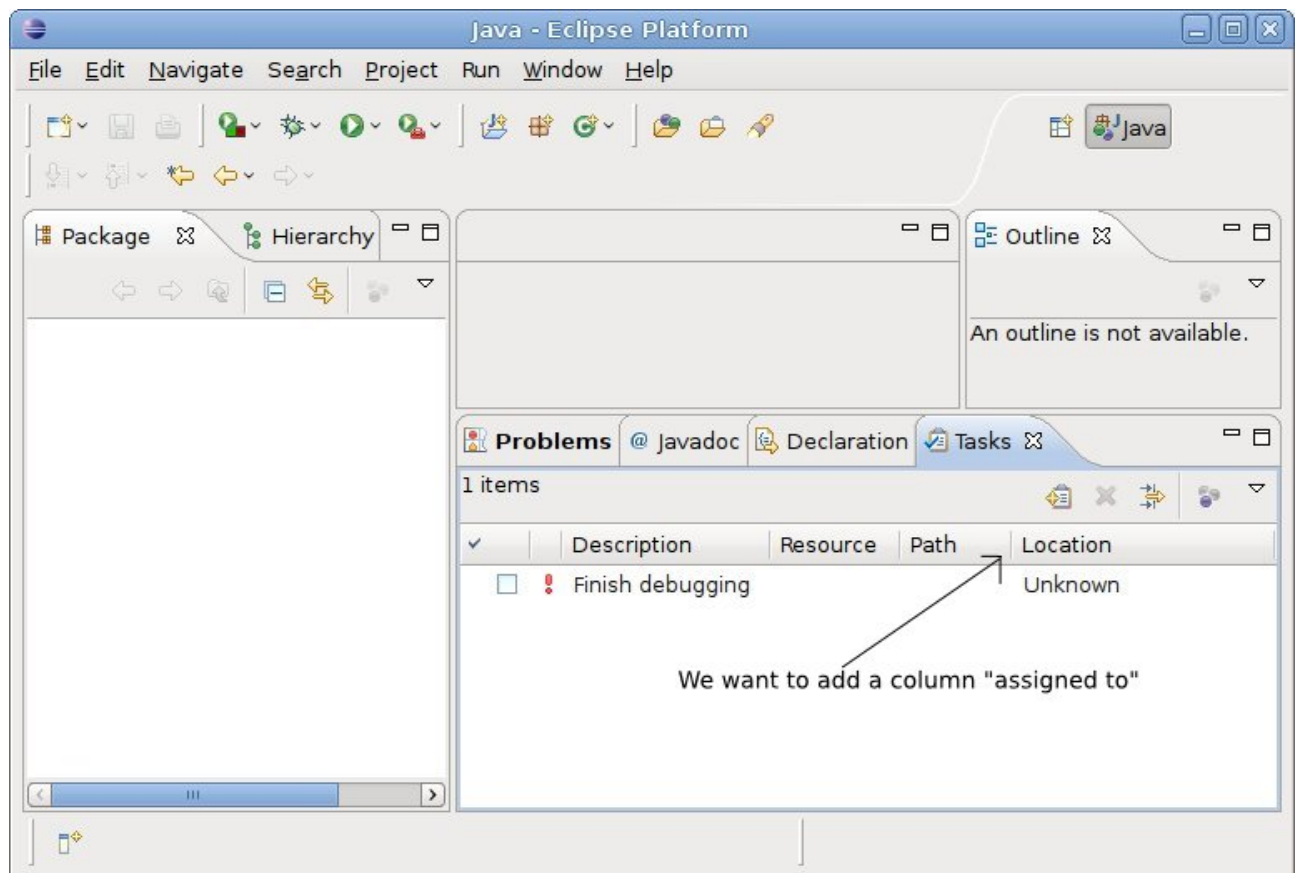
Figure 7.2: The Eclipse task view

```
public class TaskView extends MarkerView {
    ...
    private final IField[] VISIBLE_FIELDS =
        { new FieldDone(), new FieldPriority(),
          new FieldMessage(), new FieldResource(),
          new FieldFolder(), new FieldLineNumber() };

    ...
```

The developers clearly did not anticipate this scenario by providing an extension point for the extra column. Both the fields and the order are hardcoded. As we clearly cannot use the addition of a new plugin to add the column (level 1), we had to use component replacement (level 2).

Creating a new, replacement version is not a perfect solution, however, due to other characteristics of the model. Firstly, the plugin consists of around 300 Java classes. The effort required in forking this plugin to create a new version is heavily out of proportion to the small architectural addition required.

Introducing a new version will also cause a problem if any other plugins explicitly declare a dependency on the old version. We will end up in that case with two referenced versions of the plugin (old and new), and simultaneous versions of the same plugin are not allowed in Eclipse for anything that contributes to extension points. Even if this were allowed, having two versions of a plugin holding shared state would not be a desirable outcome.

To prevent the situation where different versions of the same plugin are required, Eclipse plugin versions

tend to follow a convention. All dependencies on plugins are expressed as a bounded version range from 3.0.0 up to (but not including) 4.0.0. The leftmost digit of the versioning scheme indicates API-breaking[3] changes. Without this approach, we would not be able to easily introduce even a minor, non-breaking change as explicit dependencies on the old plugin version would mean having both old and new present.

It is also not possible to introduce a breaking change without also creating a new version of all plugins (incrementing the leftmost digit) which depend on this plugin, and so on in cascade fashion. This certainly constrains the type of change we can introduce, even if we are willing to update any upstream plugins which have issues with the change. We will end up having to update most of the plugins in the system.

Even creating a new non-breaking version is likely to be a short term solution. A future version of Eclipse will likely introduce a new version of this plugin, to fix defects and enhance functionality[4]. As it is not possible to instantiate the two versions at the same time, we will have to accept the fact that we must merge our source changes into each new release of the plugin. Regardless of how important we view our change, it is unlikely that the maintainer (the Eclipse Foundation) will incorporate our (and everyone else's) changes into the plugin that they own and maintain. Our version will be superseded by any new ones from the maintainer, and our changes will be overwritten.

7.1.4   Coarse-Grained Plugins

A key problem in the above scenario is the coarseness of plugins. If the plugins could somehow be made more fine-grained then the problem would be easier to solve, as we would be creating a new version of a smaller artifact.

A tension exists, however, between making the architecture fine-grained and making it manageable and understandable. If we make plugins too small, we will end up with literally many thousands. As shown by figure 7.3, the plugin structure for Eclipse is already complex even though most plugins are coarse-grained. The figure shows around 80 plugins: a typical environment contains around 200, and an enterprise product based on Eclipse is known to contain over 500 [GB03]. Having more plugins implies a less manageable system. Some form of nesting or composition would address this, providing a way to view the system at multiple levels of abstraction[5].

As plugins are necessarily coarse-grained then, plugin replacement also works at a coarse-grained level leading to extra overhead if a plugin must be replaced.

---

[3]A breaking change is one which makes the public API incompatible with previous versions.

[4]Eclipse 3.4 was recently released with an updated version of the *org.eclipse.ui.ide* plugin.

[5]Eclipse plugins have names conforming to Java-like conventions. However, like Java packages, the name is just a convention and does not indicate any form of composition.
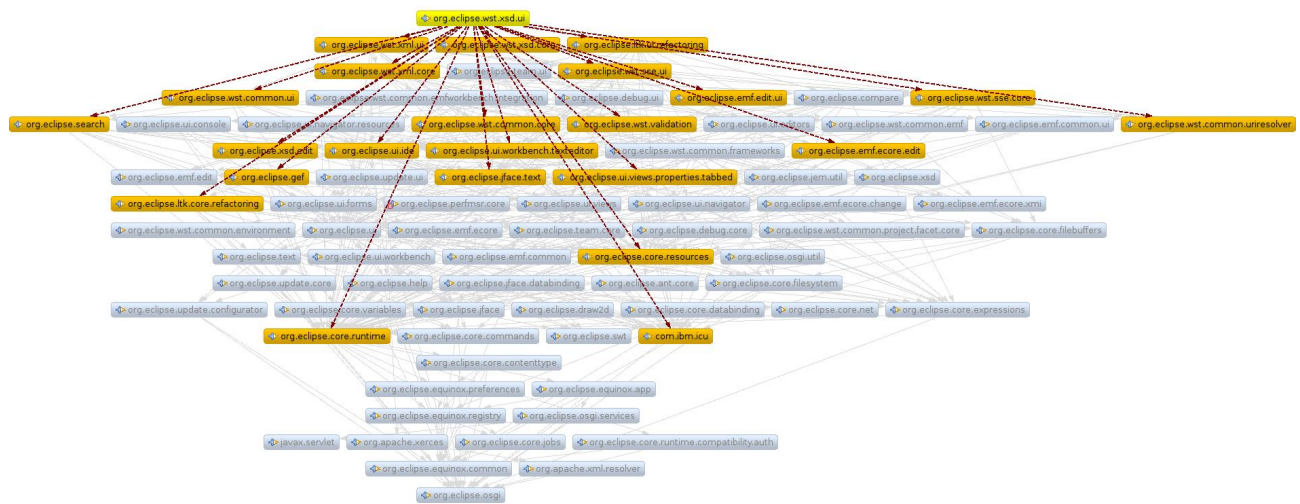
Figure 7.3: A partial plugin dependency graph of Eclipse

7.1.5 Characteristics of the Plug-In Model

As discussed in section 7.1.1, the plugin model operates at two levels. Adding functionality via a new plugin (level 1) relies on existing extension points in the application. If, however, these points do not allow for the type of extension required, then we must upgrade or replace an existing plugin with a newer version (level 2) and introduce the extension points. As shown, even minor changes can move us into the need for plugin replacement.

Our small extension necessitated a new version of a plugin. This presented several problems. Firstly, the plugin is a necessarily sizable artifact due to the lack of composition, and creating a new version requires effort that is out of proportion to the size of the small architectural change actually required. Next, introducing a new version can lead to a situation where multiple versions of a plugin need to be instantiated, which is usually prohibited according to the rules of the platform. Finally, creating a new version leads to its own problems in that we are not the main developers of this plugin and will therefore have to acquiesce to performing a source level merge of our changes whenever a new version is published by the Eclipse Foundation.

As it turns out, our small change was not planned for – the creators of the `TaskView` plugin did not anticipate or cater for this type of change via extension points. This is an interesting characteristic in that unplanned changes, even those notionally adding a feature, must be modelled as as coarse-grained plugin replacement. If the requirement had been foreseen, an extension point could have been provided to allow the registration of extra columns for the task view. Clearly, however, anticipating all future changes is a costly and largely unrewarding exercise. The architecture will become polluted with extension points, creating a lot of extra development work, which in turn may not in fact capture all possible future requirements.

An underlying problem when using the plugin model is that extension points do not arise naturally out of development of an architecture. The points have to be explicitly planned for. Further, code needs

to be written for extension points to utilize any registered `PluginExtensions`, which increases the
development overhead of this approach.

It is a useful exercise to consider how well the Eclipse plugin architecture meets the requirements
outlined in section 1.3. In our scenario, the system fails to provide sufficient flexibility because we
are unable to make changes without copying and modifying the source code for the plugin to handle
the upgrade (ALTER, NOSOURCE). We are also unable to seamlessly accept a new version without
having to perform a source level merge of our version with the newer, official one (UPGRADE). The
requirement that changes have no impact (NOIMPACT) on existing consumers of the plugin is partially
met, as there is no need to force the upgrade on those who do not wish to see the new version. However,
introducing an upgrade may require multiple versions to be simultaneously loaded which is not possible
when a plugin contributes to extension points.

As it is, the plugin model contains practical and undesirable limitations on how easily a system can
be extended, understood and managed. Plugin addition (level 1) is simple, but is only applicable if
the required extension points are already available. Essentially we are saying that extension is easy if
the type of extension has already been planned for. Even simple extension requirements move beyond
this, however, leading to the need to use coarse-grained plugin upgrade or replacement (level 2).

7.1.6   Modeling the Task View in Backbone

In this section, we model the task view scenario in Backbone. We then show how an extension can use
evolution to add the extra column, avoiding the issues from the plugin example.

Going back to the task view example of 7.1.3, consider modeling the concept of a grid (view) column
(e.g. "Description") in Backbone. Figure 7.4 shows `GridColumn` as a leaf component. It has a single
attribute `name` and provides the `IGridColumn` interface. It requires the `IGridDataProvider`
interface which it uses to retrieve data in order to populate the column. We will later use instances of
this component to configure the task view's visible columns.
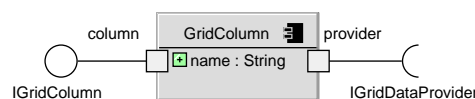
Figure 7.4: A column component

The `GridWidget` component, defined in figure 7.5, is the user interface widget that displays the grid.
The grid is made up of a set of columns, and this information is supplied by the `r` port.

Figure 7.5: A grid widget component

Backbone ports that require interfaces are analogous to extension points in the plugin model. Ports that provide interfaces are analogous to `PluginExtensions`, which provide data to extension points. Interfaces are more expressive however, as object references can be passed in addition to data and class references.

To handle the logic of the task view display, we define a controller component, as shown in figure 7.6. Although not shown here, it has become established practice in Backbone to break down this type of logic using a state machine. This allows for decomposition of a component featuring complex logic into smaller states (section 6.3.3).

Figure 7.6: The controller handles the display logic of the task view

We can now use the components to form the task view as a composite component, as per figure 7.7. We have configured four columns ("Description", "Resource", "Path", "Location") up to the `GridWidget` part. Note the use of alphanumeric indices to control the order: `GridWidget`'s r port is ordered.
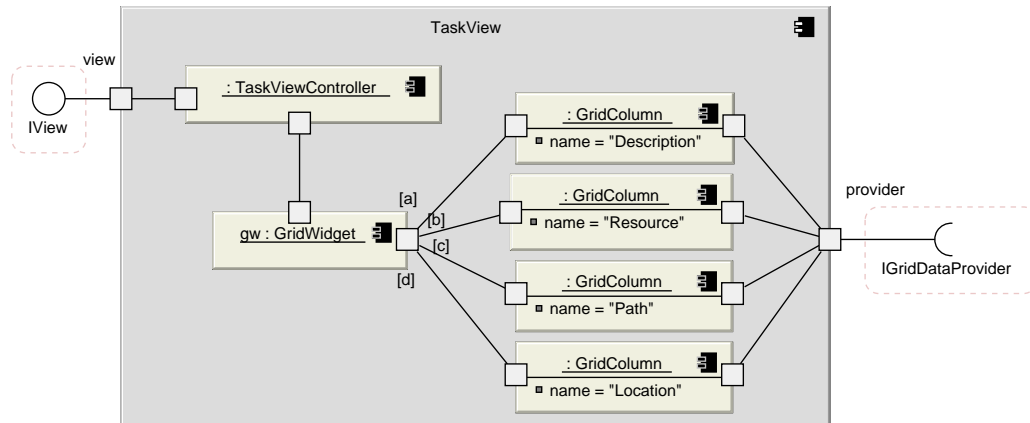
Figure 7.7: The task view component

Running the backbone example gives the screenshot shown in figure 7.8.

Figure 7.8: The Backbone task view example, showing four columns

Adding the Extra Column Using an Extension

The `TaskView` component and associated elements model the existing Eclipse task view, before the requirement to add the "Assigned to" column. Our extension will need to insert the extra column in between two existing columns.

To achieve the above effect, we create an extension stratum, called `enhanced`, which contains an evolution of TaskView. This evolution is shown in figure 7.9.
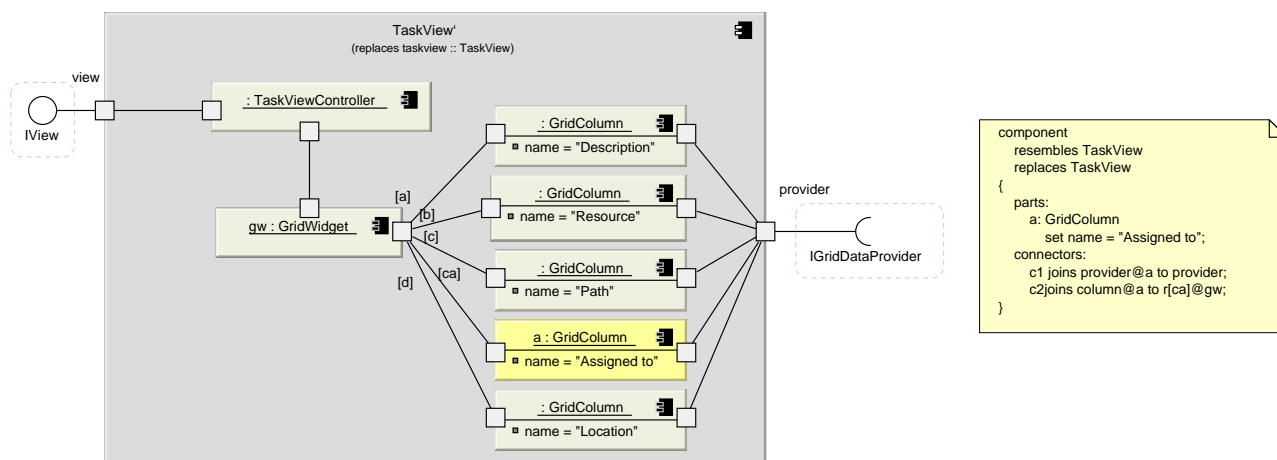
Figure 7.9: Evolving TaskView to insert another column

The `TaskView`' evolution has added a `GridColumn` part for our new column. All other parts and structure are inherited from the original `TaskView`. To insert the column in the correct place, the `[ca]` index is chosen. The effect of the extension can be seen in the screenshot of figure 7.10.

Figure 7.10: The Backbone task view example, showing the inserted column

The evolved component is packaged into an extension stratum as shown in figure 7.11. We can create a system that includes both this stratum and `taskview`, thereby applying the replacement and getting the additional column, or we can exclude it and recreate the original list of columns.

The Backbone extension facilities are more flexible than this example can demonstrate. For instance, by replacing connectors, we could switch the order of the columns. We could upgrade the controller part, add in extra display logic or even remove or rename columns.
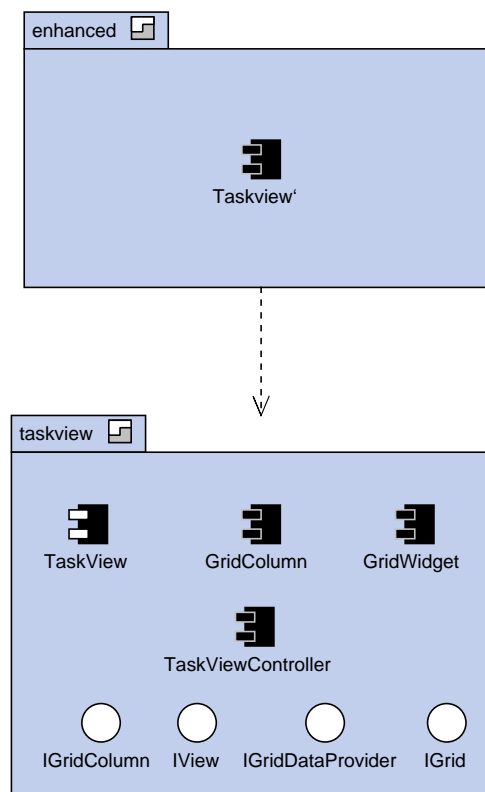
Figure 7.11: Packaging up the extension

7.1.7   Characteristics of the Backbone Model

The Backbone component model is hierarchical, allowing components to be decomposed to a fine-grained level. This in turn allows any replacement to be more targeted than in the plugin model. Changes can be made at the appropriate level of abstraction (APPROPRIATELEVEL). Large architectures can be represented and managed using hierarchy, as it is a scalable concept.

Further, any element of the model (component, part, attribute, connector etc) is a natural extension point, as it can be replaced or evolved. Unlike the plugin model which requires advance planning even for addition or insertion, the ability to extend is a seamless part of creating a system in the Backbone model. As developers model and refine a Backbone architecture, extension point possibilities become naturally more numerous.

The analog of the plugin model level 1 (addition of functionality using predefined extension points) is Backbone evolution where constituents are only added to existing components. The analog of level 2 (plugin upgrade) is evolution with alterations to delete or replace parts and connectors.

As all component creation in Backbone is expressed via parts, the architecture is more explicit than a class-based model that hides object instantiation in code. This allows and extension to adjust the configuration through the replacement construct to make any changes required to component instantiation.

7.1.8   Summary

The plugin model facilitates additive change, via plugging into existing extension points (level 1). If modification of existing plugins is required, because the required extension points are not present, then a new plugin version must be created (level 2). This characteristic of turning notionally additive change into replacement interacts badly with the coarse-grained nature of the plugins, leading to a disproportionate effort for simple changes. Further, distributing a new version of a plugin can lead to version conflict, particularly if others (including the primary source) are also releasing independently updated versions. Key to these limitations is the lack of plugin composition, leading to a trade-off between plugin size and manageability.

The plugin model blurs the distinction between component and module concepts. In the Eclipse model, a plugin is akin to a module as as it controls dependencies and packages up a set of classes, interfaces and other resources. A class (contained within a plugin) is akin to a component, as it forms the unit of instantiation and composition. However, the unit of upgrade or replacement is a plugin (module) rather than a class (component). As noted above, replacing a plugin can be very expensive.

Backbone offers a more flexible alternative, ameliorating or directly addressing the extensibility limitations of the plugin model. The replacement and resemblance constructs, coupled with a hierarchical component model, allow an extension to perform "surgery" on the architecture at the appropriate level of abstraction. Unlike the plugin model where extension points must be pre-planned, in Backbone extension points arise naturally out of normal system decomposition.

Although the Backbone facilities are useful for creating extensions, the constructs are also valuable in their own right for system modelling and design. Resemblance, in particular, allows component structures to be inherited and altered which facilitates reuse.

Another benefit of the Backbone approach is that it encourages and supports the explication of system structure to a fine-grained level. Whereas the plugin variant of the task view encoded much of the system structure inside code, the Backbone model was explicit about the structure and connections. This allowed us to alter the structure without changing the implementation code. It is important to make the point that this is not a situation that can be rectified in the plugin model by providing more, or finer-grained plugins. The plugin model simply does not cover architectural connection or composition, and as such must rely on implementation to instantiate and connect various parts of the system. This type of instantiation cannot be easily visualised or controlled by configuration.

The expressiveness of the Backbone approach has a cost, which is also present in the plugin model: we cannot guarantee that combining independently developed, but potentially structurally overlapping, extensions into a single application will not produce some conflicts. To address this, we previously demonstrated (section 3.3.7) that a further extension can be used to rectify structural conflicts. As mentioned, a form of this dilemma occurs also in plugin architectures when plugin versions collide in a single application, caused by combining independently developed plugins which have conflicting dependencies. In this situation, however, there can be no guarantee that a further plugin can resolve the situation, and a cascade of plugin replacements may be required.

7.2 Backbone versus a Software Product Line Approach

A software product line (SPL) can be used to create a portfolio of closely related system variants [CN01]. We create a variant by choosing from a set of possible features, and then assembling the application by combining the software artifacts representing each feature. Clements and Northrop show how organisations using this technique have been able to demonstrate dramatically improved software metrics over traditional approaches for managing a set of related applications.

The first step in creating a product line is to perform an analysis of the target domain. This is used to produce a feature model which defines a tree of optional and mandatory features. This technique is known as feature-oriented domain analysis [KCH$^+$90, Bat05].

A number of different techniques exist for assembling a variant once the features have been chosen [Kru06]. These compositional approaches rely on a mapping from features to software artifacts. Further, the artifacts must be phrased so that the assembly process can allow different implementation sections (corresponding to features) to be combined. An SPL approach must therefore take a strategic and planned view towards the creation and reuse of underlying assets.

A prominent technique and toolset for creating product lines is AHEAD (Algebraic Hierarchical Equations for Application Design) [BSR03]. In this section, we model the audio desk extension scenario from section 1.2 using AHEAD, allowing us to compare and contrast this to the Backbone model of the same scenario. AHEAD uses the concept of refinement to express variations in existing classes, in order to adjust them for new features. We show that the characteristics of refinement result in a poor match against the extensibility requirements of section 1.3.

7.2.1 Using AHEAD to Model a Product Line

AHEAD is a compositional technique and toolset for modelling a product line. It is able to manage and assemble more than just source code artifacts: it provides a general approach which can also apply to other implementation artifacts (e.g. makefiles) and domain-specific languages.

AHEAD introduces *constants* and *functions*, and allows these to be combined using a simple algebra. A constant is a base program. A function refines a constant, according to a notion of refinement which is specific for each type of artifact. Functions can be concatenated. The order of concatenation is important because some forms of refinement involve overriding parts of other artifacts. A *collective* can be used to group constants, functions and other collectives, providing a level of composition.

To assemble a system, we first choose the set of features we wish to have, and next choose the collectives corresponding to these. Finally we use the toolset to combine the constants and functions to create a complete system.

This is easier to understand in concrete form. When using AHEAD with Java, a constant is simply a set of class definitions. A function can contain both class definitions and refinements of existing class constants: each refinement comprises a list of field additions, method additions, and method overrides

to a base class. This works in a similar way to defining a class via inheritance, although refinement affects the existing class definition rather than making a new class.

Consider applying AHEAD to a small calculator program in order to augment it with additional features. This example is a subset of the one given in the AHEAD tutorial [Bat06]. The language actually used is JAK, which is a conservative extension of Java that supports the approach.

Firstly, we define the base calculator as a constant. This class holds a result, but cannot yet perform any operations.

```
// calculator.jak
class Calculator {
  double result = 0;
  public String result() {
    return "" + result; }
}
```

Consider the `addition` refinement below, which adds the ability to perform addition to the calculator. The `result()` function is also overridden to return more information. The invocation of `Super().result()` refers to the base calculator's `result()` method defined in the constant.

```
// addition.jak
refines Calculator {
  public void add(double val) {
    result += val; }
  public String result() {
    return "Result is: " + Super().result(); }
}
```

The `subtraction` refinement gives the ability to perform subtraction.

```
// subtraction.jak
refines Calculator {
  public void subtract(double val) {
    result -= val; }
}
```

We can combine the constant and both functions by using the following equation.

```
addition(subtraction(calculator))
```

The same variant is also produced by a permutation, showing that order is not always important.

```
    subtraction(addition(calculator))
```

These equations both result in the following system variant

```
    class Calculator {
      double result = 0;
      public void add(double val) {
        result += val; }
      public void subtract(double val) {
        result -= val; }
      public String result$$calculator() {
        return "" + result;
      public String result() {
        return "Result is: " + result$$calculator(); }
    }
```

Overriding the `result()` method in `addition` has caused the base calculator's method to be renamed to `result$$calculator()`. The `Super().result()` invocation inside `addition` now redirects to this method, reflecting that if structured as an inheritance hierarchy, the super call would invoke the method in the base class.

Alternatively, we could use `subtract(calculator)` to create a variant which just features subtraction.

```
    class Calculator {
      double result = 0;
      public void subtract(double val) {
        result -= val; }
      public String result() {
        return "" + result }
    }
```

AHEAD allows each feature to be modelled as a set of new classes and refinements to other classes. Design rules can then express feature dependencies and whether or not certain features can be combined. The result is a powerful and flexible way to assemble a collection of system variants.

In the AHEAD literature, an extensible system is defined as one where a change to a system requires a proportionate effort to modify its source code [BJMvH02]. This definition is more or less contained within our ALTER requirement for extensibility, where the cost of an extension should be proportionate to the size of the extension. The AHEAD papers do not specify an equivalent of the NoSource requirement, although it is implied that extension should primarily involve the refinement of classes and definition of new classes rather than the direct modification of existing source code.

A constant or function is represented by a directory with a set of files.

7.2.2   Modelling the Audio Desk Scenario in AHEAD

Consider an AHEAD representation of the audio desk scenario from section 1.2. As we have previously modelled this in Backbone in section 3.3, we can compare and contrast the two solutions.

Reproducing the strata diagram from figure 3.11, we now label it with a set of constants and functions, as shown in figure 7.12.
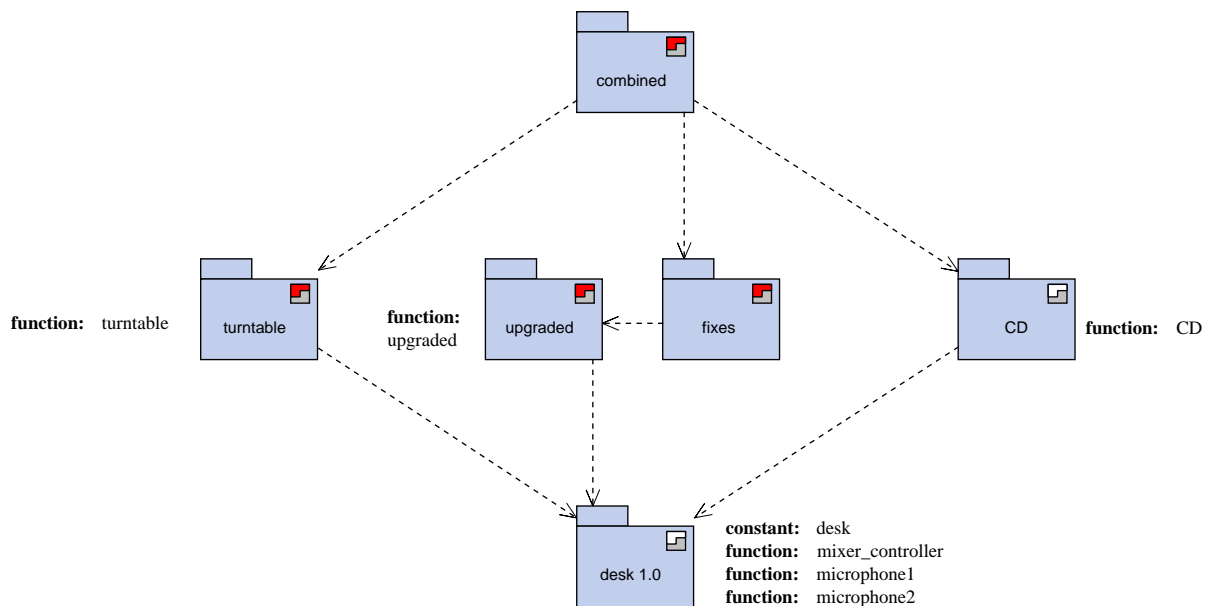
Figure 7.12: The AHEAD functions and constant for the desk scenario

Modelling the Base Application

The base application consists of a desk with a mixer, device controller and two microphones.

To start, we create a constant, which defines a minimal `Desk` class with a single method which connects up required devices.

```
// constant desk
class Desk {
  void connectUp() {}
}
```

To add the mixer and device controller, we refine the constant using the `mixer_controller` function.

```
// function mixer_controller
class Mixer {
  int volume = ...
  ... }
class DeviceController {
```

```
    ... }
  refines Desk
  {
    Mixer mixer = new Mixer();
    DeviceController controller = new DeviceController();
  }
```

To add the first microphone, we define a `MicrophoneDevice` class and refine the `connectUp()` operation to connect it to the device controller and mixer. As the controller represents a set of physical controls, we must be careful to specify an index for this. We use 0 in the `controller.connect()` method, to indicate that this microphone is in the zeroth position.

```
  // function microphone1
  class MicrophoneDevice {
  ... }
  refines Desk
  {
    MicrophoneDevice firstMic = new MicrophoneDevice();
    public void connectUp() {
      Super().connectUp();
      controller.connect(firstMic.getAudio(), 0);
      mixer.add(firstMic.getAudio());
    }
  }
```

We now add the second microphone using a similar function, except that we connect it up to the first position in the device controller. Although it may seem strange to describe the two microphones separately, there is an underlying rationale that will shortly be revealed.

```
  // function microphone2
  refines Desk
  {
    MicrophoneDevice secondMic = new MicrophoneDevice();
    public void connectUp() {
      Super().connectUp();
      controller.connect(secondMic.getControl(), 1);
      mixer.addAudio(secondMic.getAudio());
    }
  }
```

We can now combine the constant and functions to create the base desk application. The following equation produces an equivalent program to the Backbone model of figure 3.12.

```
microphone2(microphone1(mixer_controller(desk)))
```

Modelling the CD and Turntable Extensions

We now model the CD extension using refinement.

```
// function CD
class CDDevice {
... }
refines Desk
{
  CDDevice cd = new CDDevice();
  public void connectUp() {
    Super().connectUp();
    controller.connect(cd.getControl(), 1);
    mixer.addAudio(cd.getAudio());
  }
}
```

Now consider if we wanted to create a desk with a single microphone at position zero, and a CD player at position 1. We can do this by adding the CD function to the equation and omitting `microphone2`, as per the following equation.

```
CD(microphone1(mixer_controller(desk)))
```

Refinement is limited in that it cannot remove fields from a class – the only way to omit something is to not include the function in an equation. This is our rationale for separating out the two microphone functions previously. If the two microphone functions were combined into one, we would have to either have no microphones (i..e. omit the function from the equation) or both microphones together (i.e. add the function to the equation). As we want only one microphone, we must separate the two functions. This type of pre-planning required for extensibility will be shown to be a consequence of how refinement in AHEAD works.

Now consider how an extension developer might choose to phrase the addition of a turntable device. The mixer must be upgraded to support cuing (as per section 3.3.5) and the developer has chosen to express the entire extension as a single function called `turntable`. Suppose also that the cuing mixer can be defined as a refinement to the original mixer.

```
// function turntable
class TurntableDevice {
    ... }
```

```
refines Mixer {
   ... add cuing support }
refines Desk
{
   TurntableDevice tt = new TurntableDevice();
   void connectUp() {
      Super().connectUp();
      controller.connect(tt.getControl(), 2);
      mixer.addAudio(tt.getAudio());
      mixer.addCueAudio(tt.getCueAudio());
   }
}
```

We can create a desk with a CD, turntable and microphone device using the following equation.

```
turntable(CD(microphone1(mixer_controller(desk))))
```

Modelling the Mixer Upgrade and Resolving Conflicts

Our recreation of the desk scenario is not yet complete. As per section 3.3.6, we wish to upgrade the mixer and device controller. However, the new mixer implementation is obtained from an external party as per the scenario. It is not a refinement of the previous mixer[6]. As such, we have the following function called `upgraded`.

```
// function upgraded
class Mixer {
   CommercialMixerAlgorithm cm = ...
   ... upgraded mixer }
class DeviceController {
   ... }
refines Desk {
   Mixer mixer = new Mixer();
   Mixer cueMixer = new Mixer();
   DeviceController controller = new DeviceController();
}
```

An upgraded desk with a CD and microphone device is assembled using the following equation. Note that this omits the `mixer_controller` function – the upgraded mixer and controller are provided by the `upgraded` function.

---

[6]In the original scenario, the upgraded mixer is distributed as a binary component to protect the intellectual property. JAK currently requires the source code for all constants and functions, but this is a limitation of the current toolset rather than a conceptual issue. As such, we do not consider this issue further.

```
CD(microphone1(upgraded(desk))))
```

7.2.3   Limitations of the AHEAD Approach

Although we can combine the CD, microphone and upgraded desk, other combinations are not possible and reveal the limitations of AHEAD as an extensibility approach. For instance, we cannot combine the upgraded and turntable functions in an equation. They are incompatible because the former provides a different mixer definition to the one that the latter refines. Although there is a chance that these would work together and that the new definition would define exactly the same methods needed for overriding, this is highly unlikely because neither were created with knowledge of the other.

In general if a function defines a class, then it cannot be concatenated with a function that refines a different definition of that class. Field definitions are particularly problematic here as combining a class and all refinements gives us the union of all fields, and removing a field involves changing the source code or omitting the function that adds it.

We could potentially ameliorate this problem by expressing every single change (particularly field additions) as a separate AHEAD function, and then composing these fine-grained functions into a manageable set of collectives. Unfortunately, however, if one of the functions in a collective conflicts with a function in another collective, then the two collectives cannot be used together. Resolution involves defining a new collective, which may involve significant work as there may be an arbitrary amount of composition involved.

The issue of function granularity now surfaces. In [BSR03], the JTS system consists of 33k lines of JAK code and is modelled using 69 distinct features illustrating that a single function can be relatively large. However, if any part of a function conflicts with another, then we cannot include both functions in an equation and we must choose between the two. If we do wish to include the features of both, we then need to adapt or recreate a potentially coarse-grained artifact. Even a small syntactical conflict can result in the need to rewrite an entire function. The ever-present tension between making replaceable elements small for ease of extension and making them large for ease of management is a problem for the AHEAD approach, just as it is for the plugin approach (section 7.1.4), although for subtly different reasons. Function composition does not fully address the problem.

The granularity problems of AHEAD are further illustrated in [KAK08], where it is shown that even going down to the level of method granularity is sometimes not sufficient for certain changes. If we did put every method in a separate AHEAD function this would result in many fine-grained functions for even small programs, which would prove difficult to manage. The use of coarse-grained features in AHEAD and the potential difficulty of managing large numbers of small-scale refinements is discussed in [BJMvH02].

The use of names for identity in AHEAD makes refinement fragile, and means that naming conflicts between independently-developed refinements are possible. Once chosen, the name of a class or interface must stay constant or otherwise identity is lost. Backbone deals with this by using UUIDs to represent logical identity, which allows an extension to rename elements and constituents.

7.2.4   Summary

AHEAD is a powerful and flexible way to structure a product line. An AHEAD program consists of a set of constants, and functions which add to and refine those constants. When applied to Java, a function can refine the definition of an existing class. A variant is specified using a concatenation of functions and constants.

Refinement is related to inheritance in that it can add fields, and add or override methods. These characteristics colour the AHEAD approach. If one feature includes a definition or refinement that conflicts with those contained in another feature, then the features cannot generally be combined. Further, a conflict cannot usually be resolved by a refinement in another feature, as refinement provides no way to remove or rename objectionable fields. This characteristic interacts badly with the granularity of features – we can get into a situation where a coarse-grained feature must be recreated or destructively edited in order to allow it to be combined with another.

These characteristics violate a number of our extensibility requirements from section 1.3. In particular, only certain alterations can be expressed by refinement (ALTER) and more complex changes involve changing the source code (NOSOURCE). Introducing a feature with a new class or refinement can cause a conflict with the definitions of another feature, which means that the latter feature must be recreated or edited which will impact existing usage (NOIMPACT). It is not always possible to fix the conflict by a further refinement (DETECTANDCORRECT). This means that a disproportionate amount of work can result from the introduction of a feature. Upgrades in particular, where the change cannot be expressed as refinement, can lead to this situation (UPGRADE).

As such, AHEAD necessitates a strategic and premeditated outlook towards reuse. This is noted in [BJMvH02]. It does not cope well with unplanned extension.

By way of contrast, a Backbone strata is loosely analogous to an AHEAD constant or function. Evolution is analogous to AHEAD refinement. A conflict between two strata can always be resolved using a further strata. The effort to do so is proportional to the change required to allow the definitions to work together, limited by how fine-grained the underlying components are. This characteristic can be traced back to the fact that resemblance allows deletion and non-compatible constituent replace.

In section 8.3.2, we discuss the relationship between extensions and features and indicate how feature modelling could be incorporated into a future revision of the Backbone approach.

Finally, AHEAD relies on a conservative extension of Java called JAK, in order to provide refinement. This prevents existing Java tools being used for the approach. In contrast, Backbone uses Java, with a set of lexical conventions for the leaf implementations. Composites are specified using the Backbone ADL.

7.3   Using Backbone to Create and Extend a User Interface

In this section we demonstrate how Backbone can be used to construct a user interface, by utilising a complex, existing toolkit structured around an object-oriented perspective. We show that the full extensibility benefits of the Backbone approach apply even in this environment, allowing an extension to evolve the user interface to accommodate new requirements.

The underlying toolkit used is the Google Web Toolkit (GWT) [Goo09a]. This works by converting an underlying Java program into JavaScript. The code generated by Backbone can be used as a GWT program, thereby allowing a Backbone application (including all advanced facilities such as factories) to run completely within a web browser on the client side. This further demonstrates the applicability and flexibility of the Backbone approach.

In presenting this example, we are not advocating that Backbone represents a superior way to build user interfaces. Our intention is simply to demonstrate that Backbone is flexible enough to work with complex object-oriented libraries, bringing the full benefits of extensibility, in an unusual setting. Although the libraries chosen are made up of JavaBeans which can already be represented as Backbone components (section 5.2.9), the more complex beans require a level of restructuring in order to make their hidden structures explicit.

7.3.1   Constructing the Base User Interface

GWT is a library for creating web user interfaces featuring rich interaction [Goo09a]. Programs are written in Java, and then translated into JavaScript where they can be run in a web browser. A number of impressive applications and Google products are written using this toolkit [Goo09b].

As GWT is packaged as a JavaBeans library, Evolve can import and use the widgets. For the translation into JavaScript to work, programs need to adhere to certain conventions and respect the limitations of the approach. For instance, only limited reflection facilities are available. The Backbone code generator can generate Java code for a configuration (section 5.3.3) and this is suitable for the translator. Although state machines normally use reflection for the `StateDispatcher` component (section 6.3.3), GWT provides a facility whereby a program generator can be registered with the runtime and this was used to work around the limitation.

Importing the GWT toolkit into Evolve results in around 220 components and interfaces. Figure 7.13 shows some of the widget components after import, with their most important features visible. `HorizontalPanel` is a container widget that arranges its child widgets in a horizontal orientation. A `VerticalPanel` container also exists. `Label` allows a textual label to be shown, and `TextBox` allows a line of text to be entered by the user. `Button` presents a button that sends out events via the `clickListeners` port when it is pressed. `TabPanel` is a container that can display multiple tab pages.
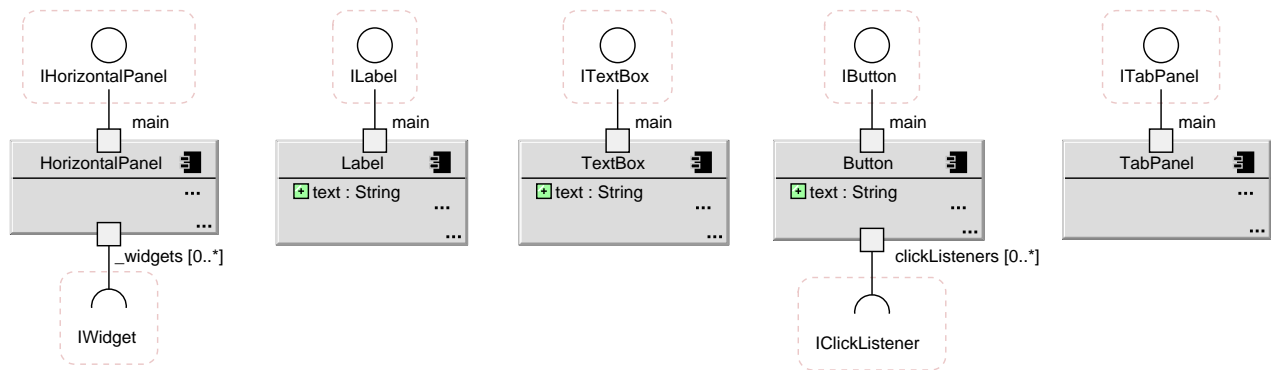
Figure 7.13: Some GWT widgets as Backbone Components

Using the TabPanel Container Widget

We wish to use the `TabPanel` widget to display a user interface. Looking at that component's structure, however, it does not appear that a port exists for adding tabs. Inspecting the documentation, it becomes apparent that when adding a tab that we must also specify a title at the same time. This is shown in the following listing.

```
TabPanel tp = new TabPanel();
tp.add(new Button("Hello"), "Page1");  // add a button to tab "page1"
```

This is not a convention that Backbone understands, and for this reason the extra port has not been configured correctly upon import. In short, `TabPanel` does not make all of its structure explicit in a way that Backbone can use, and we need to work around this.

To allow tabs to be added via connectors, we must build on `TabPanel`. The easiest way to do this is to create a new leaf, resembling `TabPanel`, that provides a `titleTab` port to add the extra widgets. The title for each tab can be taken from the `getTitle()` method on the existing GWT `IWidget` interface. We define the `SimpleTabPanel` component in figure 7.14.
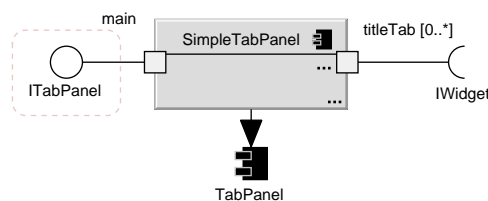
Figure 7.14: SimpleTabPanel allows ordinary widgets to be added as tabs

The implementation of this component is trivial. It inherits from the `TabPanel` class and adds each widget connected to the `titleTab` port as a separate tab.

As an example, we can now connect up two different tabs, each having a single button, as shown in figure 7.15.
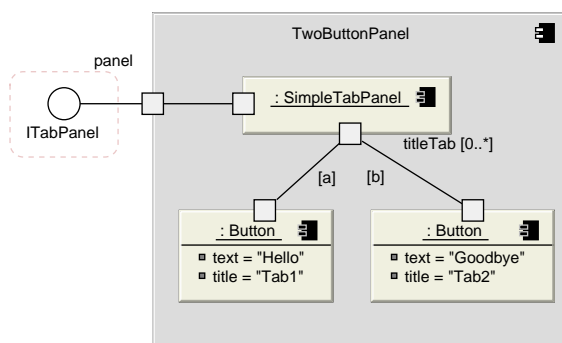
Figure 7.15: Using SimpleTabPanel to configure two buttons as tabs

Running this example in a browser produces the screenshot in figure 7.16.
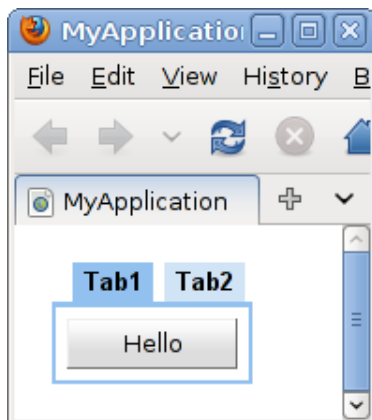
Figure 7.16: Screenshot of tabs using SimpleTabPanel

It is interesting to note that `TwoButtonPanel` is actually a composite widget. The creation and reuse of composite widgets in user interface toolkits is generally not a trivial task. However, this is intuitive and simple in our component model. If required, we could expose further services and ports, or use resemblance to reuse our composite's structure.

A More Complex Interface

Consider that we now wish to construct a more involved user interface that allows a user to enter their name and address. We create two widgets – one for entering a name, and another for entering an address. The `GWTNameWidget` component is shown in figure 7.17. It uses a `VerticalPanel` part to create a vertical arrangement of labels and text fields for entry of the user's first and last names.
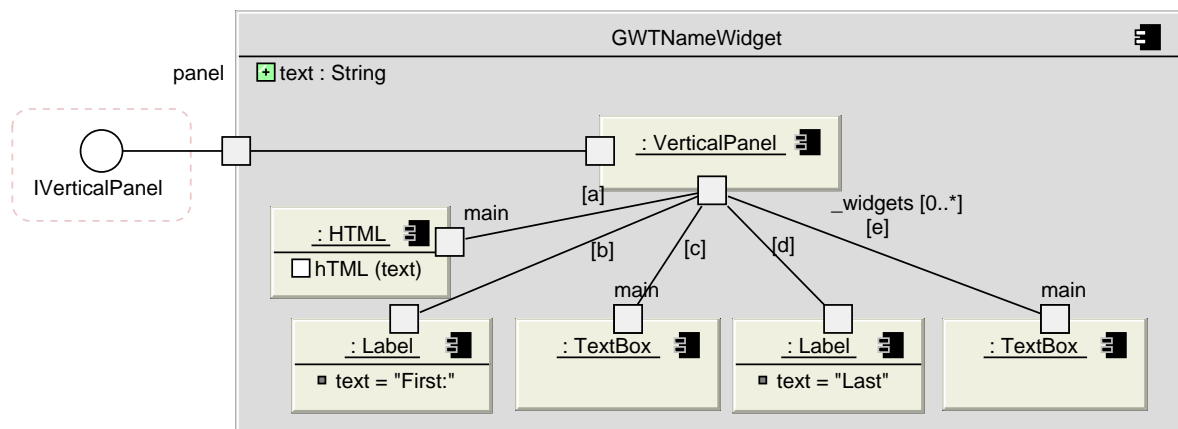
Figure 7.17: A composite widget for name entry

The widget for entering in an address is structured similarly, as shown in figure 7.18.
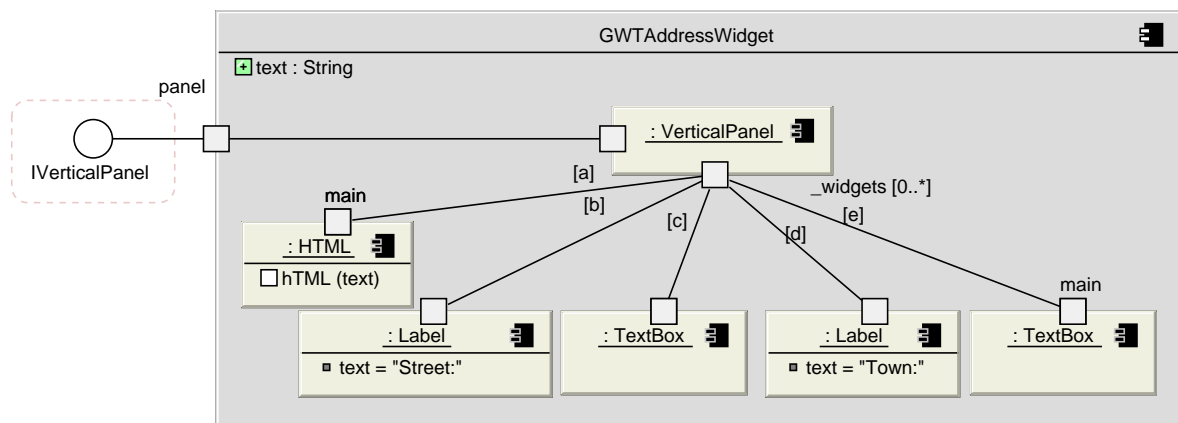
Figure 7.18: A composite widget for address entry

We can now bring these together in figure 7.19 to create a composite widget which allows both the name and address to be entered. We have used `HorizontalPanel` to place the child widgets alongside each other.
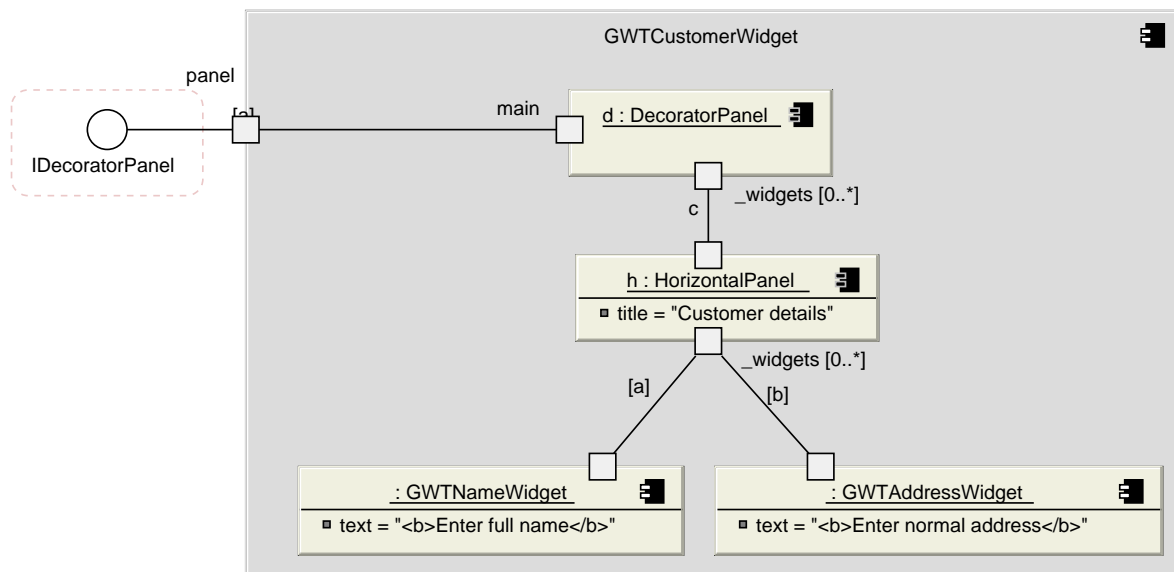
Figure 7.19: A widget that allows name and address entry

Running this gives the screen shot in figure 7.20.
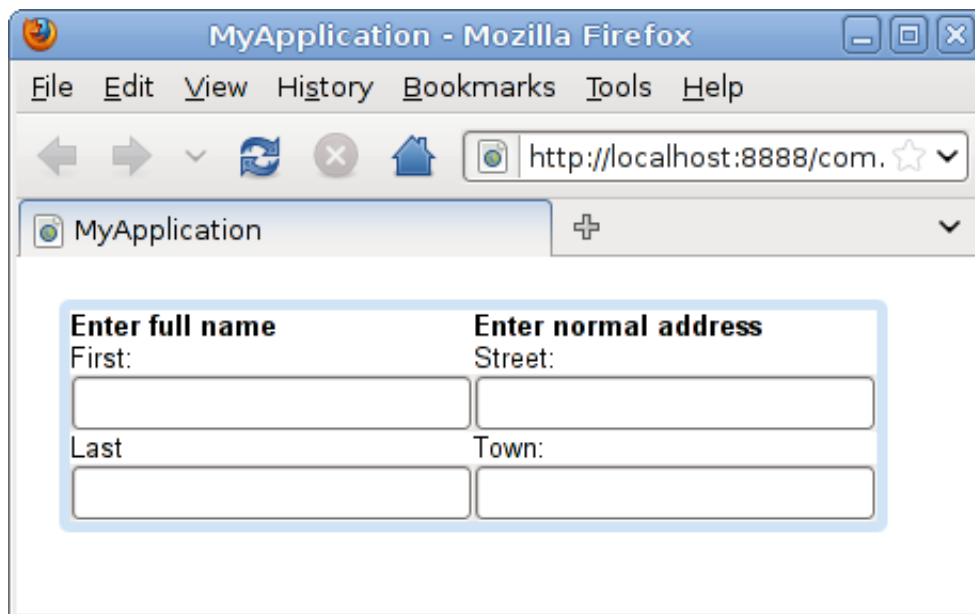
Figure 7.20: Screenshot of the name and address entry widget

7.3.2   Evolving the User Interface in an Extension

Suppose that an additional requirement is presented after using the base interface in applications for a while: commercial customers should be able to specify a billing address.

One way to achieve this is to use separate tabs for the normal and billing address, where the billing tab is created using dynamic instantiation. To accomplish this, we first define the `AddressFactory`

component using an isomorphic factory as per figure 7.21.
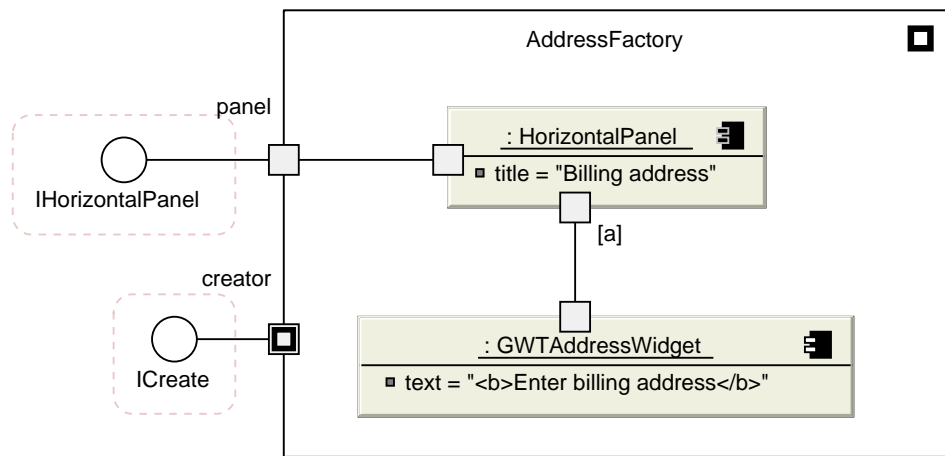
Figure 7.21: A factory to create address widgets

To incorporate this into our interface, we evolve `GWTCustomerWidget` as shown in figure 7.22. The textual definition is also provided to show the deltas.
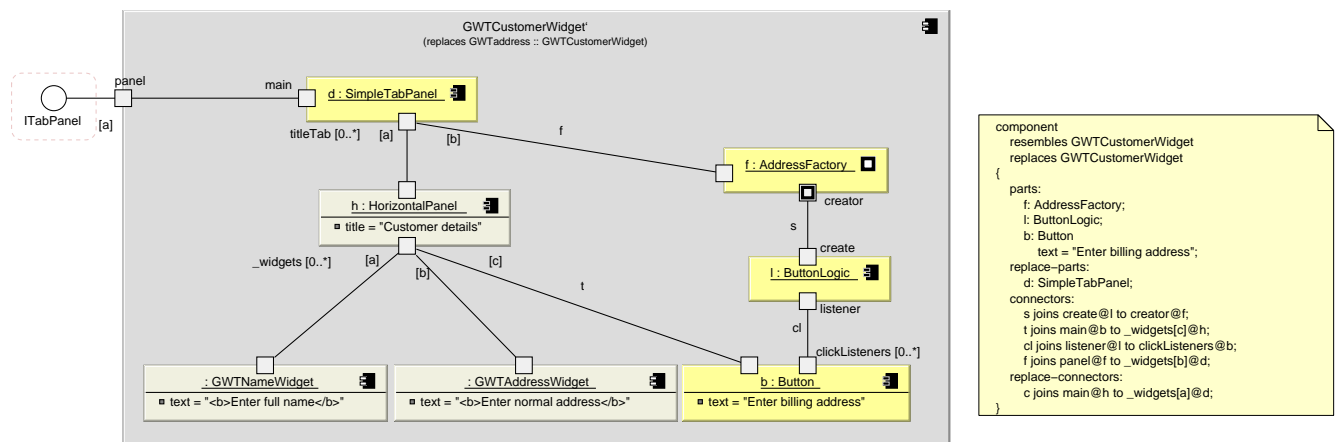
Figure 7.22: Adding a billing address entry

We have replaced part `d`, which was previously a `DecoratorPanel`, with an instance of `SimpleTab-Panel`. We have also added a `Button` part, which appears on the first tab, to the right of the existing fields. Upon clicking this button, the `ButtonLogic` part receives the event and instantiates the factory. This creates a further tab with a `GWTAddressWidget` part for entering in the billing address.

Running this results in the screenshot of figure 7.23.
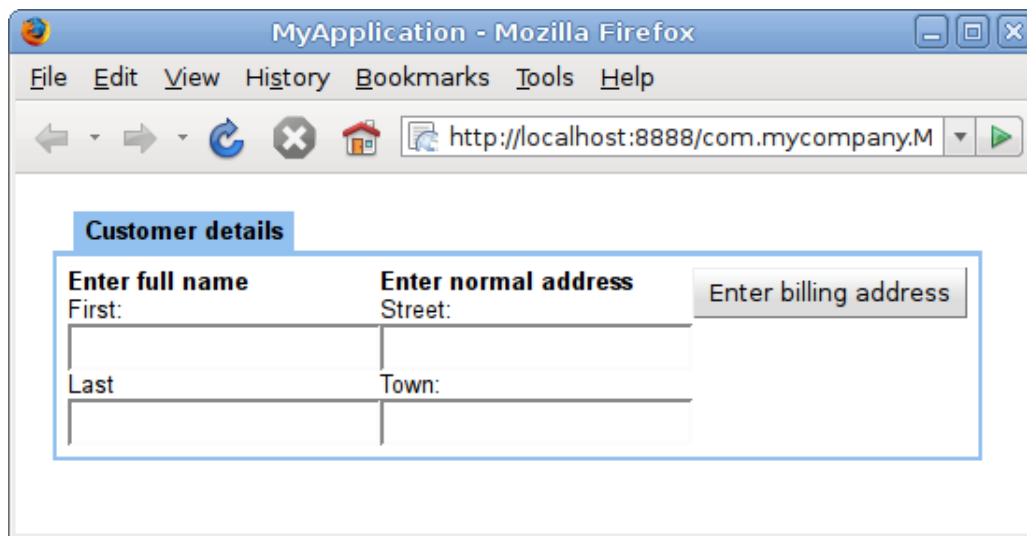
Figure 7.23: Screenshot of the extended widget with billing address button

Clicking on the button adds the extra tab and brings it into focus, as shown in figure 7.24, allowing the billing address to be entered.
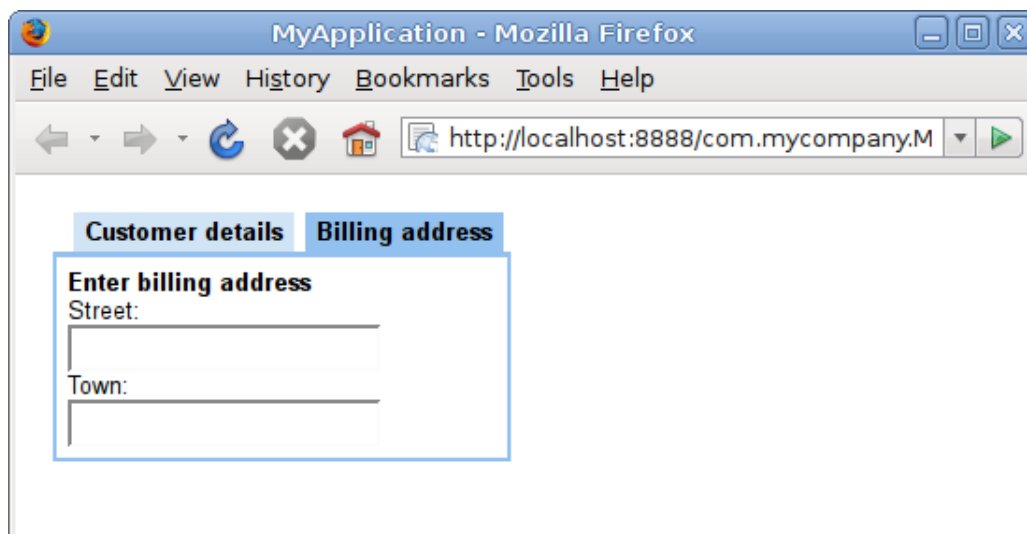
Figure 7.24: Screenshot of the extended widget with billing address tab

We have successfully created an extension that evolves the user interface to allow additional information to be entered, as per the new requirement. Any developer utilising the original, base interface can include this extension to add the facility for entering billing addresses.

7.3.3   Combining Extensions to Form User Interface Variants

Suppose that another developer decides, independently of the billing address extension, to extend the GWTNameWidget to add a field for the next of kin, as shown in figure 7.25. This is done by evolving the name widget to add an extra label and entry field.
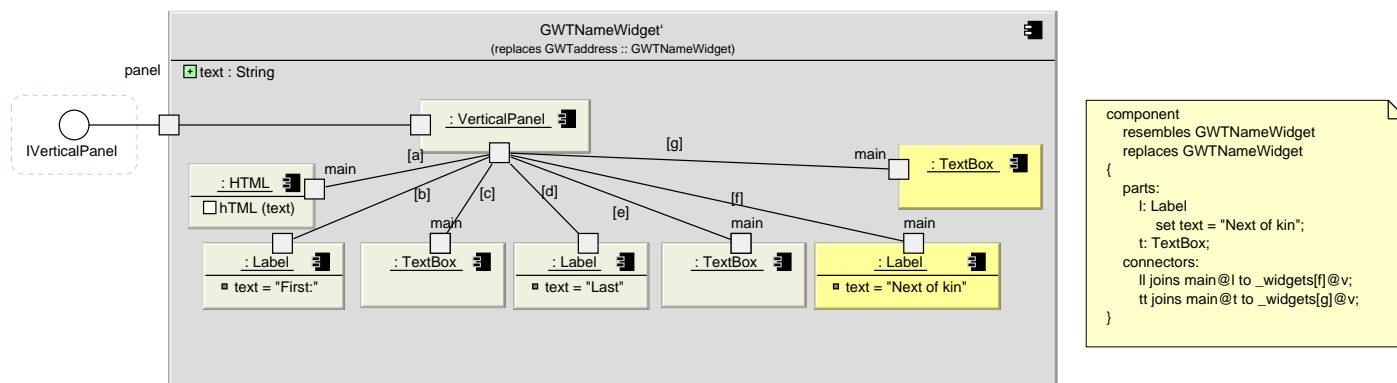
Figure 7.25: Adding a next of kin field

As a side point, it is worth noting that the Evolve modelling tool allows us to visually elide constituents from a view, in order to reduce visual complexity. The view shown in figure 7.26 is equivalent, but omits all of the inherited textbox and label parts allowing us to concentrate on the parts and connections that are added. The ellipsis in the bottom right corner of the component indicates that some constituents are not being shown.
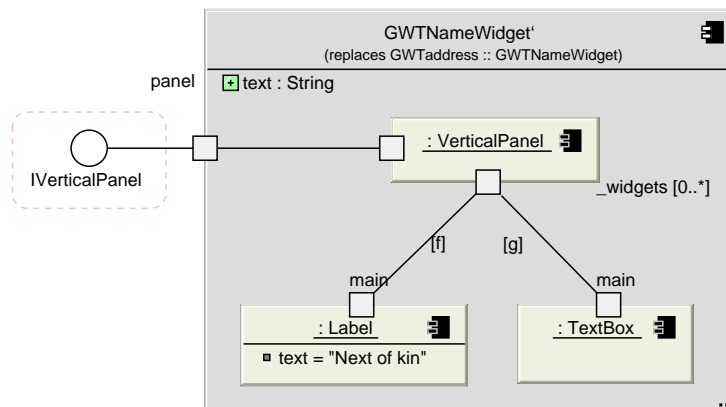
Figure 7.26: The next of kin extension with inherited widgets elided

We now have two independent extensions: one to add a billing address and another to add a next of kin field. We can structure these as per the strata diagram in figure 7.27, where the initial name and address entry components reside in the GWTaddress stratum, the billing extension resides in GWTbilling, and the next of kin extension resides in GWTkin. The GWTall stratum is empty, serving to simply combine both extensions. If the two extensions conflicted, evolutions to correct the structure would be placed in GWTall.

This allows us to form four different variants, as shown in figure 7.28. If we run the system from the perspective of GWTaddress, we get the original user interface as depicted in (a). If we run from the GWTbilling perspective, we apply the billing address extension (b). If we run from the GWTkin perspective, we apply the next of kin extension (c). Finally, if we run the system from the GWTall perspective, we get both extensions applied (d).
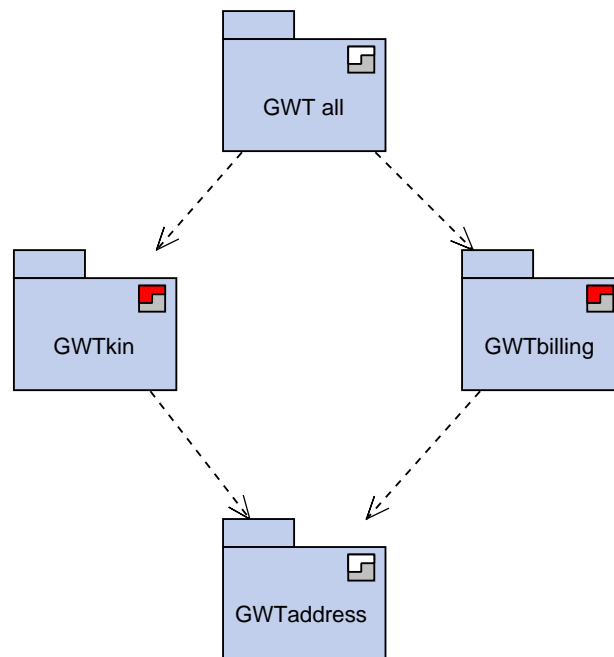
Figure 7.27: Strata organisation of extensions

(a) the original user interface

(b) the interface with the billing extension

(c) the interface with the next of kin extension

(d) the interface with both extensions applied

Figure 7.28: User interface variants formed by combining extensions

Rapid Turnaround of Changes

Evolve works well with the GWT environment to allow the rapid re-execution of the model when a change is made. After an Evolve model is modified, the developer regenerates the flattened representation as Java code. If the existing GWT hosted browser is currently displaying the previous version of the application, we can now simply refresh to display the new version immediately. GWT translates the Java into JavaScript on demand, avoiding the need for an explicit compile phase.

In practice, the turnaround time for these sorts of changes is around five seconds, depending on the size of the model.

7.3.4 Summary

In this case study, we created and evolved a component-based web user interface using two extensions. This demonstrates the flexibility and applicability of Backbone, in that it can adapt to an exotic environment and utilize a complex, existing library. The full benefits of the extensibility approach are retained. All facilities, including factories, autoconnection, hyperports, state machines and evolution, are available even in the limited GWT environment.

Some modelling and development work is generally needed to use complex libraries from Backbone, in order to make explicit the more advanced structures hidden inside implementation code. In our example we had to provide the `SimpleTabPanel` component, via resemblance from `TabPanel`, in order to allow tabbed interfaces. This type of adaptation can be accomplished without perturbing the libraries, and it allowed us to use connectors and parts instead of Java code to configure the requisite object-oriented structures.

We are not advocating Backbone as a superior method of creating user interfaces, although it could conceivably complement the more traditional use of a direct manipulation user interface builder. As opposed to Backbone, builders typically cannot model the entire application (or even entire user interface) [JS03], and have trouble representing dynamically created structures [LW08].

7.4   Using Backbone to Extend a Mature Application

The premise of Backbone is that if a system is structured as a hierarchical component architecture, then it can be extended by adding, deleting or replacing components (and their constituents) at the appropriate level of abstraction.

Underlying this approach is an assumption that the leaf components (i.e. those at the bottom of the hierarchy) will be relatively fine-grained in order to accommodate small changes in a convenient manner. If the leaves are too coarse-grained, then an extension that requires their replacement may need to duplicate much of their functionality to make the requisite changes. For modest extensions, the effort may be disproportionate to the change required.

Fine-grained leaves imply a deep compositional structure. This is feasible for an application that is developed from the start using the Backbone approach and tools which are designed to support this. Existing, mature object-oriented applications will rarely be structured in such a way, however. Many of these applications could still benefit from an extensibility architecture, even though it is not feasible to completely rearchitect them in the Backbone style.

To investigate restructuring a mature application using Backbone, we apply our approach to the LTSA behavioural analysis tool [MK06]. We focus particularly on gaining benefits quickly and with a modest amount of effort, without the need to completely remake the architecture. We further indicate how an existing architecture can evolve over time to support successively finer-grained components, eventually realising the full benefits of the approach.

7.4.1   The Labelled Transition System Analyser (LTSA)

LTSA is a tool that allows the behaviour of concurrent programs to be specified using a process algebra called Finite State Processes (FSP). A specification can then be analysed for certain properties. An FSP expression is translated into a labelled transition system (LTS), which can be analysed using model checking. LTSA can verify that a model satisfies certain desirable properties such as safety and liveness, whilst avoiding other undesirable properties such as deadlock. Figure 7.29 shows the LTSA tool with the "draw" window indicating the transitions of an LTS.

LTSA is a mature system that has spawned a number of variants. Of these the most prominent are the Ames[7] variant, which adds two extra safety checks, and the MTSA (Modal Transition System Analyser) variant [DFCU08] which allows the behaviour of a system to be analysed even if it is only partially specified.

LTSA was developed in Java in an object-oriented style. The codebase is around 16k physical lines of code[8] and consists of over 200 classes. To understand the overall architecture, we started by visualising the Java package structure using two commercial analysis tools [Ody09, Hea09]. Figure 7.30 shows

---

[7]The Ames variant originated from the NASA Ames Research centre.
[8]Physical lines of code measured using SLOCCount [Whe09]
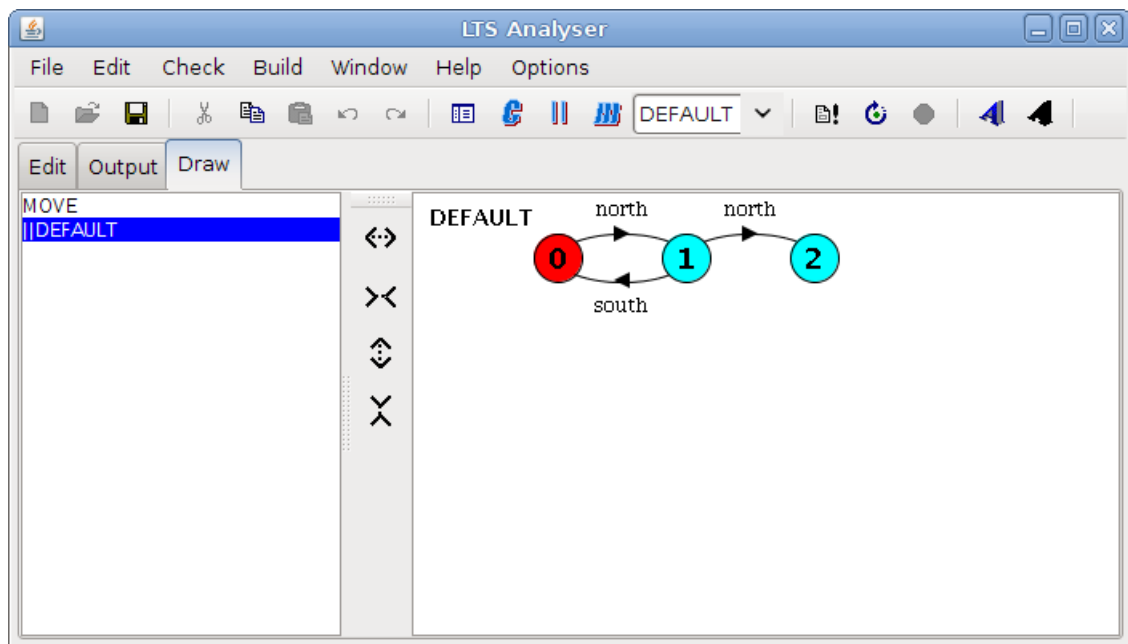
Figure 7.29: LTSA showing the draw window

the dependency relationships between the packages. Note in particular that the `lts` and `lts.ltl` packages are mutually dependent due to their classes referencing each other.
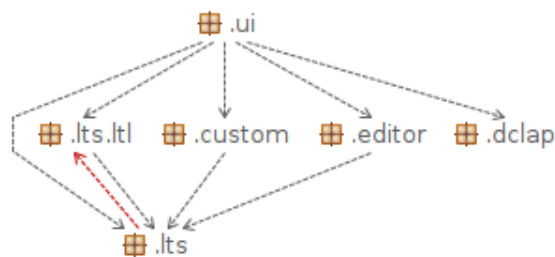
Figure 7.30: The LTSA package dependencies

The `ui` package contains the presentation classes. Graphing the classes in this package and their dependencies in figure 7.31, we see that the (large) `HPWindow` class coordinates and controls the application. It is tightly coupled to the definition of the different window types and adding an extra type would involve changes to this class.
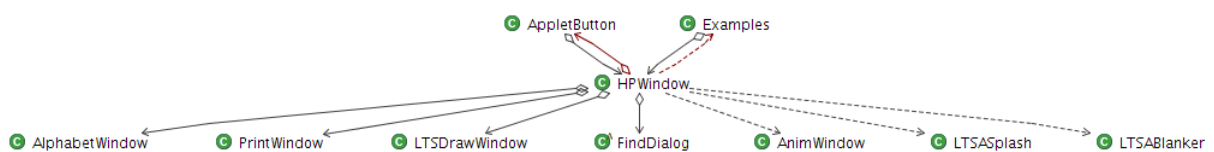
Figure 7.31: The major classes in the `ui` package

Most of the underlying logic of the application is contained inside classes in the `lts` and `lts.ltl` packages. These packages are too complex to show in a single diagram. A subset of the classes from

the `lts` package is shown in figure 7.32. The dependencies in this diagram are shown by arrows, and the thick box marked "Tangle of 3" partitions the classes which have cyclic dependencies. The `LTSCompiler` class (not shown), parses and compiles an FSP expression into an LTS and produces an instance of `CompositeState`. The `Analyser` class, which can model check an LTS for certain properties, is closely coupled to the `CompositeState` class.
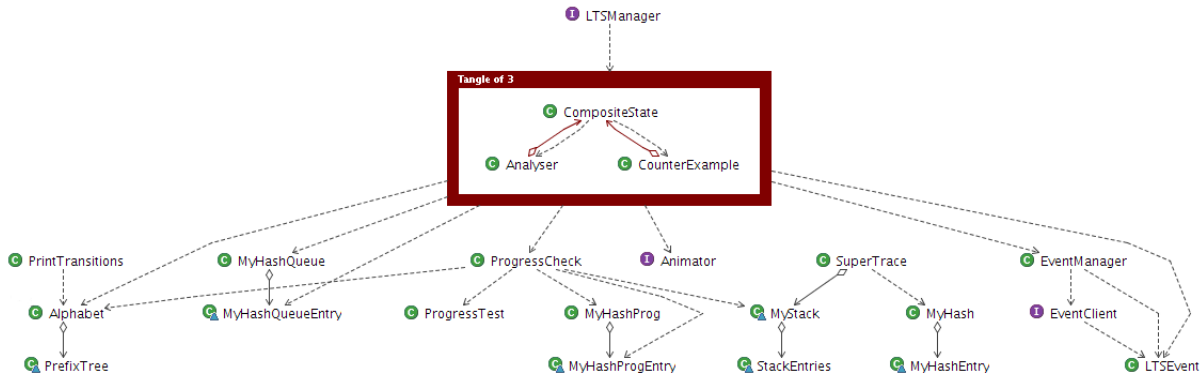
Figure 7.32: Some of the classes from the `lts` package

In general the system is reasonably structured, although a number of complex dependency cycles exist. Further, as the system has evolved, some classes have become large and logic has become centralised. In our experience, neither of these traits are unusual in complex systems, although they do complicate analysis and subsequent architectural reclamation.

It is worth noting that the system contains significant and involved functionality. The cycles in the architecture also make componentisation difficult, as this makes the boundaries between component candidates less obvious. As such, LTSA forms a good testbed for the realistic introduction of Backbone into an architecture.

7.4.2   Restructuring LTSA Using Backbone

The methodology used to introduce Backbone was to perform a top-down hierarchical decomposition of the LTSA architecture into coarse-grained components. This allowed us to represent the entire application at a shallow level of composition in Evolve. As we show later, further decomposition could then occur in a controlled manner.

Once the architecture was structured as components, it was able to be extended using the resemblance and replacement constructs, subject to the limitations of the granularity of the components.

A Starting Point

To start the decomposition process, we first made the `HPWindow` class into a Backbone component, as per figure 7.33, and phrased the LTSA application as a composite which contained an `HPWindow`

part. Clearly this was not a useful architecture for extension, but merely served as a starting point to allows LTSA to be run from inside the Evolve modelling tool.
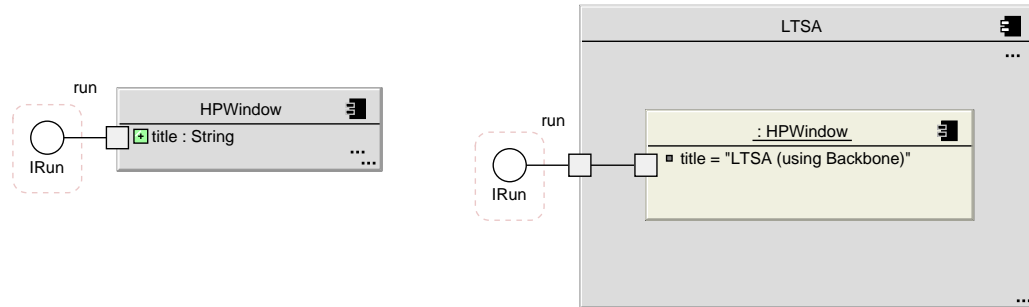
Figure 7.33: The start of LTSA componentisation

The `HPWindow` component had its implementation class set to `ui.HPWindow`. We then added a `run` port, providing the `IRun` interface in order to allow Backbone to invoke the program.

On the source code side, the changes required to the `HPWindow` class were minimal. We added the code generation markers, as described in section 5.2.10, and then regenerated the skeleton from Evolve. After generation, the class looked as shown in the following listing. The `IRunRunImpl` nested class contained the implementation of the `IRun` interface, as provided by `HPWindow`'s run port.

```
public class HPWindow {
// start generated code
// attributes
  private Attribute<java.lang.String> title;
// provided ports
  private IRunRunImpl run_IRunProvided = new IRunRunImpl();
// end generated code
  ... existing code ...
}
```

Top-Level Decomposition

Once we had LTSA running inside the Evolve tool, the next step was to look for obvious component candidates at the top level of the compositional hierarchy. This process was aided by a number of prominent interfaces in the LTSA architecture, such as `LTSInput`, `LTSOutput`, `Automata` and `Animator`. Classes which implemented or required these interfaces tended to be good component candidates as they were somewhat insulated from their environment via those interfaces.

One of the more obvious candidates was the event manager. This accepted events (`events` port), and broadcasted them to interested listeners (`clients` port). As the need to connect to this was clearly going to be pervasive throughout the levels of the architecture, we used start hyperports as per figure 7.34.
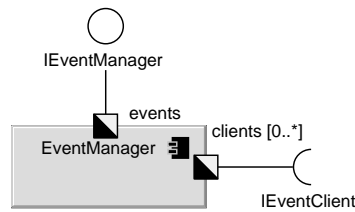
Figure 7.34: The event manager component

For the next candidates, we settled on the `AlphabetWindow`, `PrintWindow` and `LTSDrawWindow` classes of figure 7.31. These contained the logic for the alphabet, transition and draw windows in LTSA respectively. Although the `HPWindow` class was heavily reliant on the specific window class implementations, it was a relatively simple matter to decouple this by the introduction of a new interface called `IWindow`, as shown below. This represented all that `HPWindow` needed to know to control the other windows.

```
public interface IWindow {
   String getName();
   boolean isActive();
   void activate(CompositeState cs);
   JComponent getComponent();
   void copy();
   void saveFile(String directory);
   void deactivate(); }
```

Each window had a number of options displayed as checkbox menu items inside the `HPWindow` frame. To model this, we imported the Java Swing user interface toolkit into Evolve and created the `BooleanOption` component, shown in figure 7.35. This wrapped a single instance of `JCheckBoxMenuItem` (a check box menu widget from the Swing toolkit), which could be shared between the `HPWindow` component (which needed to display it) and any window which needed to read or be notified of changes to its state.
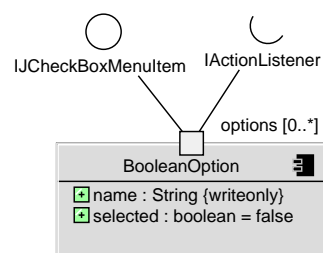
Figure 7.35: The BooleanOption component

We then turned the `AlphabetWindow` class into a leaf component, as per figure 7.36. Note that `events` was created as an end hyperport, which automatically connected to the `clients` port of the

event manager. The `AlphabetWindowFactory` component was created to dynamically instantiate and destroy the window.
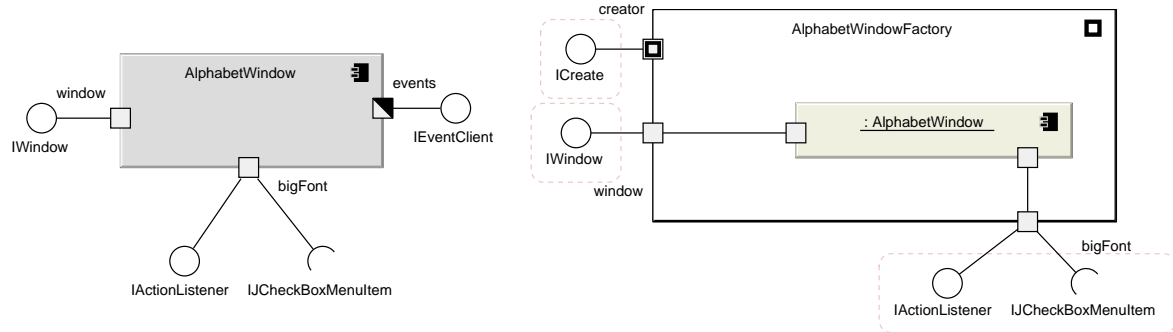
Figure 7.36: The alphabet window component

Finally, by adding extra ports to HPWindow we were able to include the alphabet window in the architecture as per figure 7.37. Note that `WindowManager` was a small convenience component created to hold a small amount of glue logic for creating the window instance. It delegated any `IWindow` calls onto the actual instantiated window, and was reused for the other window types. The alphabet window was connected up to index `[a]` of the `window` port of the `HPWindow` part.
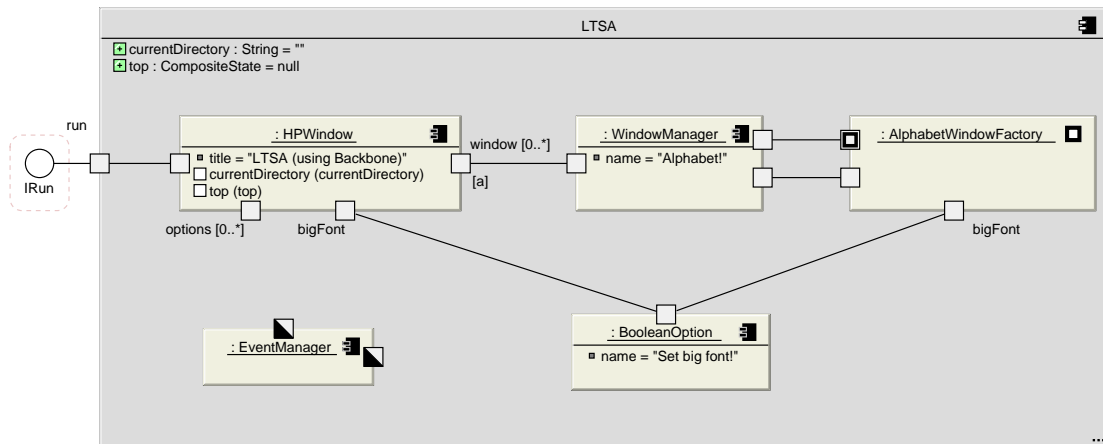
Figure 7.37: Adding the alphabet window to LTSA

The `BooleanOption` part allowed the user to select whether a big font was used for the display, and it was connected to the `bigFont` port of `HPWindow` and the alphabet window. Both the `HPWindow` and `AlphabetWindow` parts were thus notified of any state changes to this option. The `options` port allowed an arbitrary option to be connected, but we did not use this for the big font option as `HPWindow` specifically needed to know if a big font should be used for its input and output text windows.

The transitions and draw windows were turned into components using a similar approach. After the three windows were componentised, the compositional hierarchy of LTSA looked like that shown in figure 7.38. This type of compositional view is the same form as that in section 3.1, but rotated for ease of presentation.
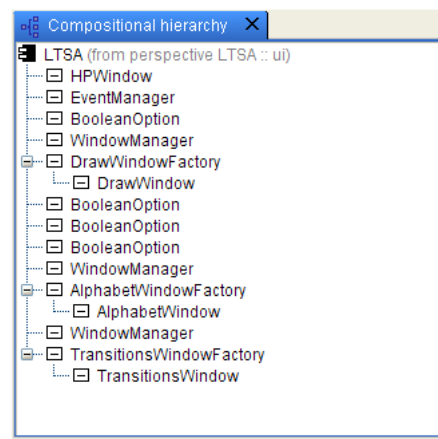
Figure 7.38: The compositional hierarchy after turning the window classes into components

Removing Object Instantiation from Code

In turning a class into a component, it is vital that its instantiation is achieved via Backbone parts and factories, rather than via the `new` operator in implementation code. This allows Backbone to control which actual implementation class is instantiated: replacement of a component in an extension can affect the actual class that will be instantiated, and this cannot easily be expressed in code.

In the original version of LTSA, the `HPWindow` code instantiated the `AlphabetWindow` class directly.

```
private void newAlphabetWindow(boolean disp)
{
  ...
  alphabet = new AlphabetWindow(current, eman);
```

In the Backbone version of LTSA, the instantiation was instead performed via the `create` port of the factory that the `WindowManager` part was connected to.

Removing object instantiation from code can have a cascade effect – turning one class into a component can force the componentisation of any classes that need to instantiate that class. Turning these related classes also into components is the preferred way to give them access to the relevant create port in order to perform instantiation. A more expedient approach is to simply pass around a reference to the create port between classes. The use of this particular technique should be seen as an interim measure until the relevant components can be extracted from the architecture.

A further point worth noting is that it sometimes makes sense to treat a class as a primitive type rather than a component. This is a temporary measure that can introduce a class into a Backbone architecture, and allow instances to be shared between components, without the effort required to turn it into a full component. We used this technique for the `CompositeState` instance referred to by the `top` attribute in figure 7.37.

Using Strata to Organise the Architecture

After working with the architecture for a couple of days, the underlying organisational structure became clear. We created a strata graph and moved each component to the appropriate stratum. For instance, the `LTSA` composite and `HPWindow` leaf were placed in the `ui` stratum and the window components were placed in to the `windows` stratum. The strata diagram is shown in figure 7.39.
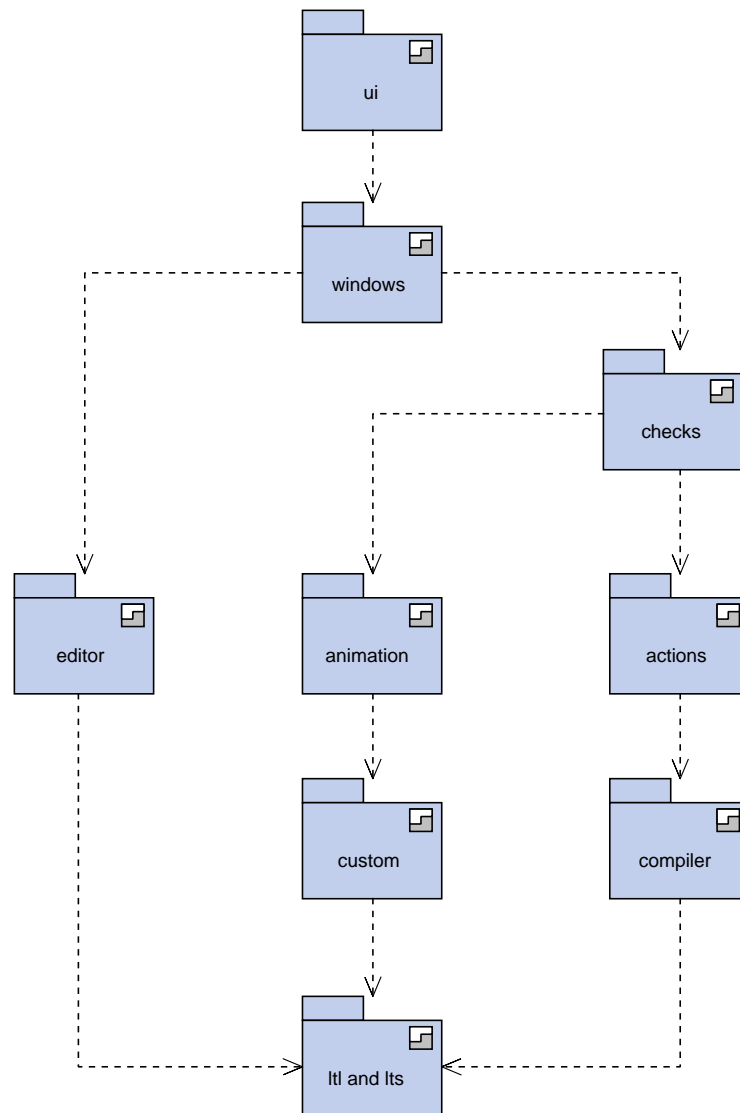
Figure 7.39: The LTSA strata organisation

Strata provide an organisational structure to a system, and ensure appropriate places for the definition of any new elements.

Turning Actions and Checks into Components

Examining the architecture further, it became apparent that the classes that performed actions (parse, compile, compose, minimise) and property checks (safety, liveness, progress) could also be made into

components with modest effort. We created the `IAction` interface, and an `Action` placeholder to define the general shape of an action or check. The compose action component is shown in figure 7.40. Note that a compiler action (via the `compiler` port) is used in order to ensure that the FSP expression is compiled before it is composed.
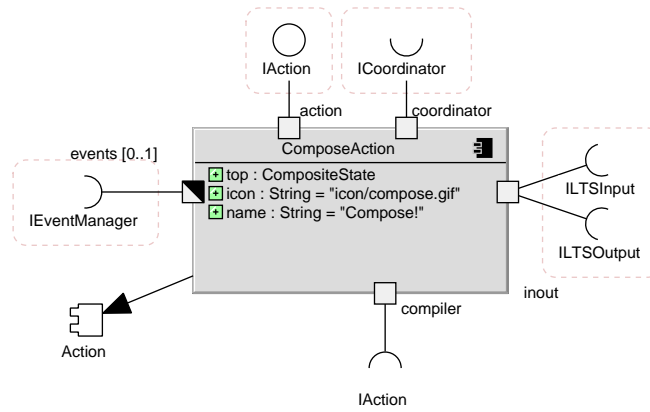
Figure 7.40: The compose action component

We then connected this up to `HPWindow` in the context of the `LTSA` composite, as per figure 7.41. This is a partial view, as denoted by the ellipsis in the bottom right corner. Evolve allows multiple, partial views of a component, which enables complex components to be specified more conveniently.
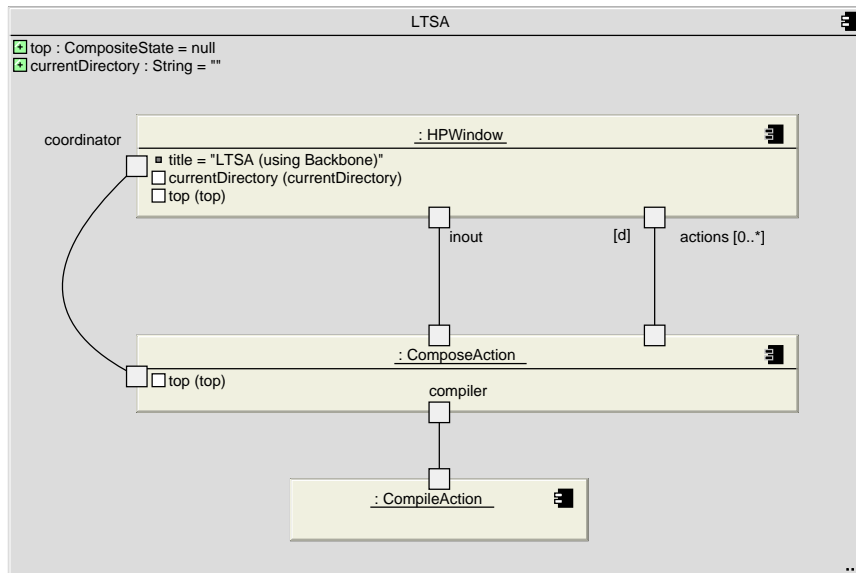
Figure 7.41: Adding the compose action to LTSA

The other actions and checks were turned into components using a similar process. The parse and compile action definitions were more complex, and the reader is referred to appendix E for instructions on downloading the full model used in the example.

Various other prominent candidates were also turned into components, including the `LTSCompiler`, `Analyser` and animator window classes. Figure 7.42 shows the `Analyser` and `AnalyserFactory` components.
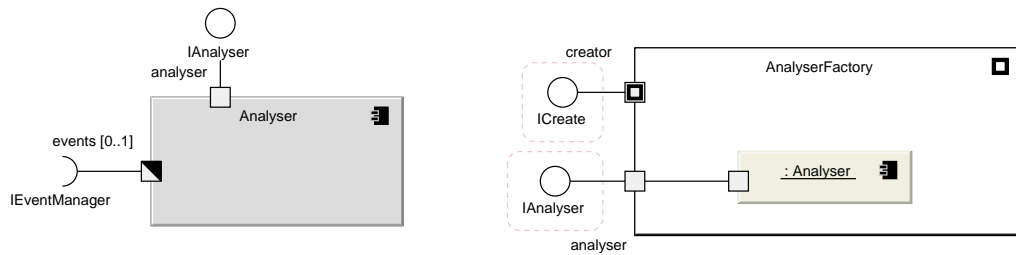
Figure 7.42: The Analyser and AnalyserFactory components

The Compositional Hierarchy After the First Pass

At this point, we had a good first pass component architecture for LTSA. Executing the LTSA system from within Evolve produced the screenshot shown in figure 7.43. Any window, menu or toolbar icon with an exclamation mark after it was controlled via Backbone. This included the check, build, window and options menus.
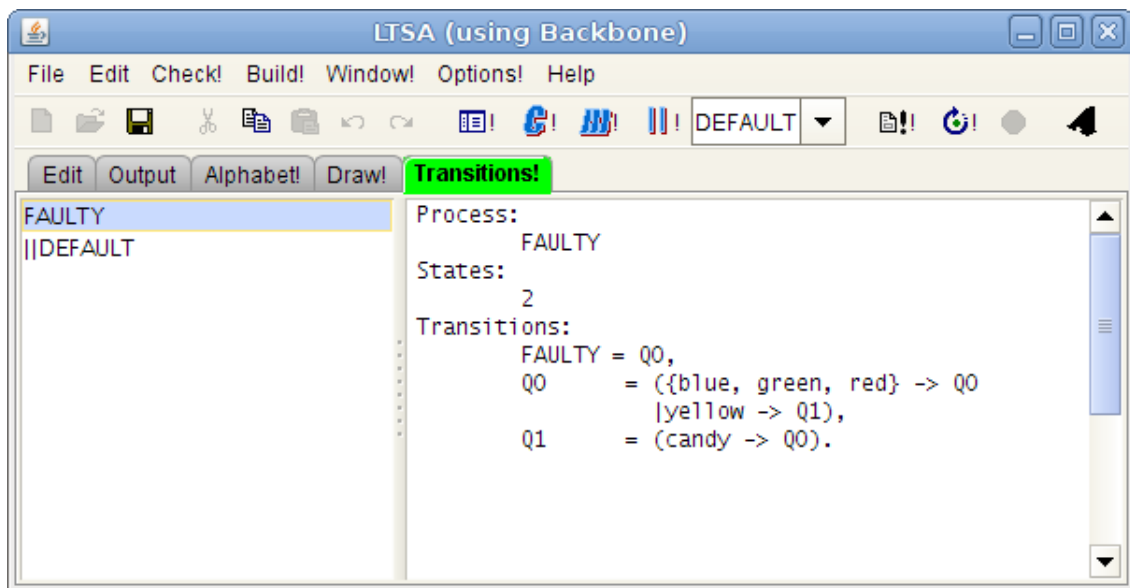
Figure 7.43: An screenshot of LTSA running inside Evolve

The entire effort required was under a week and would have been less had we been more familiar with the application architecture at the start of the process. The compositional hierarchy (figure 7.44) was shallow and wide at this point.

7.4.3 A Modest Extension: Adding a New Window Type

As LTSA was now expressed as a component architecture, it was possible to extend it using the Backbone constructs. We decided to create a "dual window" extension which would place both the alphabet and transitions windows in a single tab. Firstly, we defined a trivial `WindowCombiner` leaf component, as per figure 7.45.
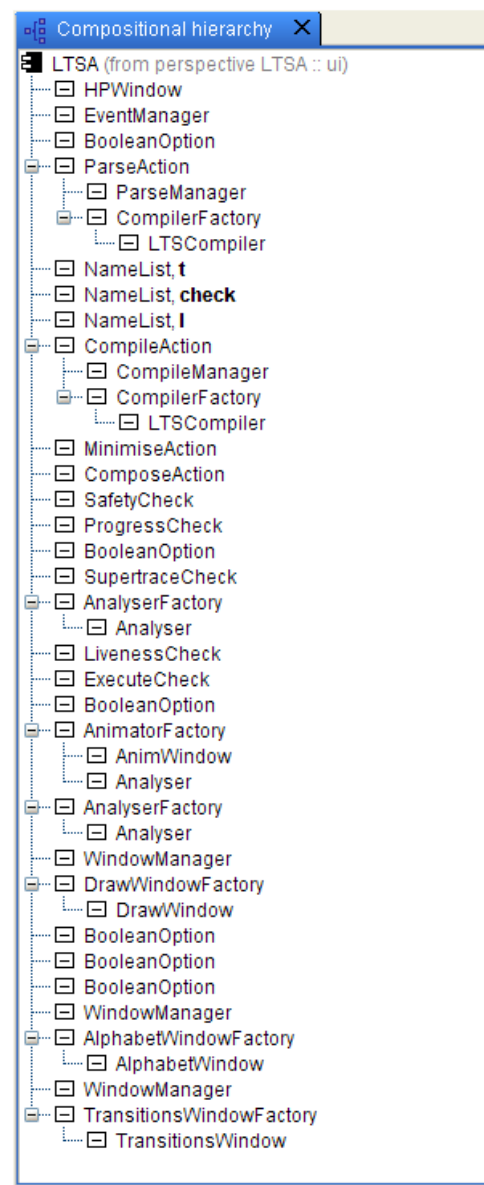
Figure 7.44: The compositional hierarchy of the initial component architecture
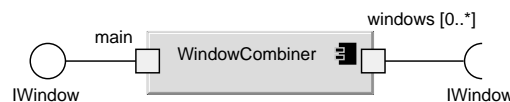
Figure 7.45: A component to combine windows

This provided a single `IWindow` interface and and hence could act as an LTSA window. Any method called on this interface simply resulted in a delegated call to the required `IWindow` interfaces of the `windows` port. This gave a fan-out effect that allowed multiple windows to act as a single window.

We then defined the `DualWindow` composite and its factory, as shown in figure 7.46. This connected a `WindowCombiner` part to the transitions and alphabet windows. We could have used any other

combination of window parts desired.
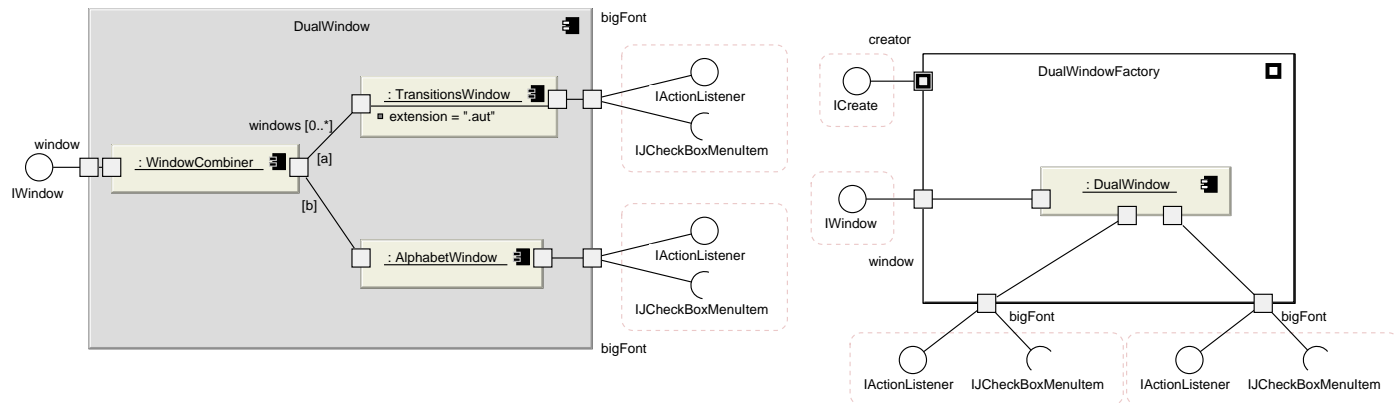
Figure 7.46: The DualWindow component combined the transitions and alphabet windows

Finally, we evolved the `LTSA` composite and connected the window into the architecture, shown in the partial view of figure 7.47. From the textual view, it can be seen that we also removed the existing alphabet and transitions window managers, factories and connectors, which were no longer required.
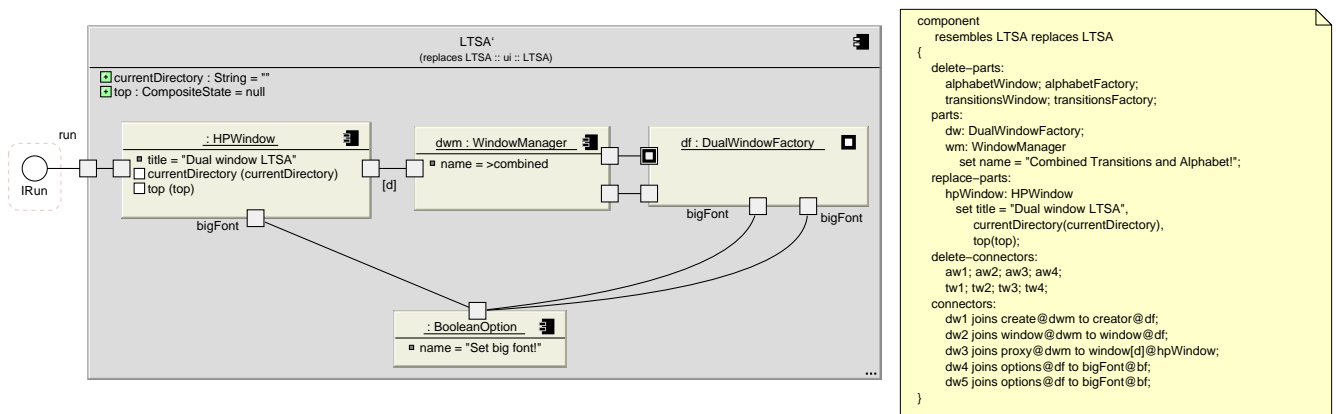
Figure 7.47: Evolving LTSA to add in the dual window

Running this in Evolve gave the screenshot of figure 7.48. The dual window positioned the transitions window at the top and the alphabet window under this, respecting the indices used in figure 7.46.
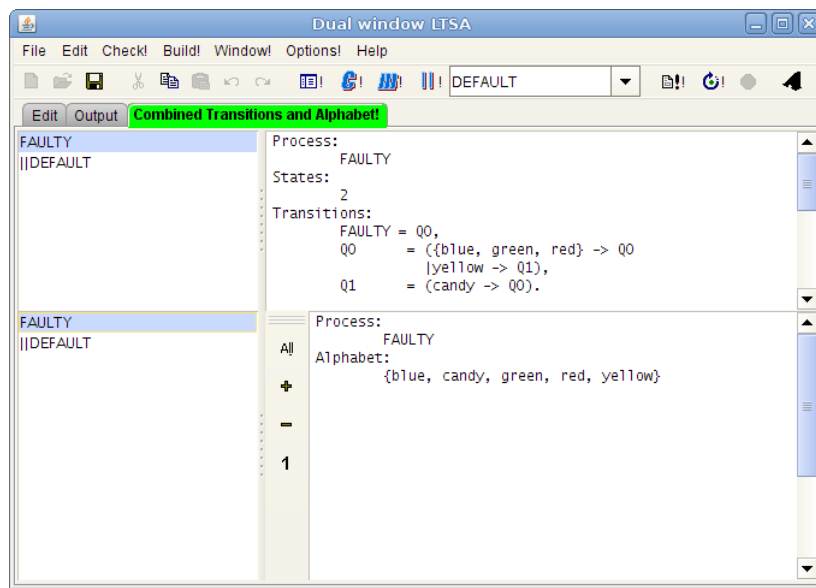
Figure 7.48: LTSA showing the dual window

We placed all of the strata of the base LTSA application (figure 7.39) inside the `LTSA` stratum, and placed the extension components and evolutions inside the `dualwindow` stratum. This is shown in figure 7.49.
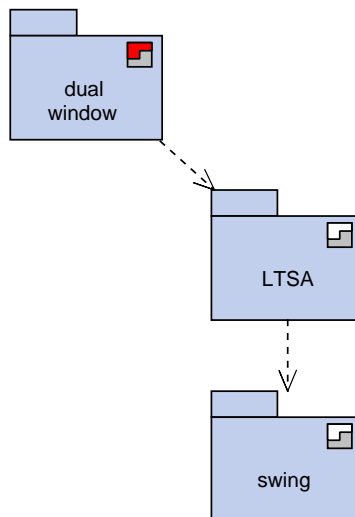
Figure 7.49: Placing the extension in a stratum

Adding the dual window was a relatively modest extension, in that we worked with the architecture at the existing, coarse-grained level of composition. If we had wanted both windows in the dual view to always show the same process name, then we would have had to decompose the `TransitionsWindow` and `AlphabetWindow` further to make this explicit in the architecture. Even that variation could have been performed without further decomposition, but this would have entailed completely replacing the implementation of either of the window components.

This extension demonstrates that with a relatively small investment, even a complex architecture can be expressed as Backbone components, and hence extended via the resemblance and replacement constructs. The full facilities of our approach and environment were made available in this way to the LTSA architecture.

7.4.4 Extending Deeply: Modelling the Ames LTSA Variant in Backbone

The Ames LTSA variant adds two enhanced safety checks. The first checks for safety whilst producing multiple counterexamples. The second checks for safety but ignores deadlock. This variant was produced by researchers copying and modifying the source code of the original LTSA application, resulting in two separate codebases.

Our goal was to express these differences as a Backbone extension to the LTSA system. We started by comparing the source code of the original to the variant and discarding changes which were inconsequential to the added functionality. After a day of analysis, we determined that most of the substantial changes were made in the logic of the `Analyser` class. For instance, two new parameters (`deadlock`, `multiCE`) and extra logic were added to the `newState_analyse()` method deep in the heart of the analyser's logic. Extra state was added for holding counterexamples, and existing logic was adjusted to utilise this.

Enabling Fine-Grained Change Via Decomposition

The Ames changes to the `Analyser` component represented a modest proportion of its overall implementation. Unfortunately, the size of these changes did not match up well with the coarse granularity of the components in the architecture at this point in time.

The ideal way to accommodate this type of situation is to hierarchically decompose existing components until the leaves are at a level where the change can be made easily. For instance, we could have further decomposed `Analyser` into subcomponents allowing us to replace one of these components in the extension. This activity fits in well with the normal process of architectural elaboration, and would happen naturally over time as more of the architecture was turned into components. As we explicate the architecture, so we add the potential for greater extensibility and vice versa. These two activities are closely related in the Backbone approach.

An alternative to decomposition would have been to replace the entire `Analyser` component with an Ames specific one. As replacing the component outright in this case would involve disproportionate effort relative to the size of the change, this was not an attractive option.

We chose to take a slightly different approach to both of the above. As we did not have the domain knowledge to decompose `Analyser` into subcomponents, we instead used the decorator pattern [GHJJ95] to allow the Ames-specific analyser to intercept methods before selectively delegating them to the existing analyser. The definition of the `AmesAnalyserDecorator` is shown in figure 7.50.
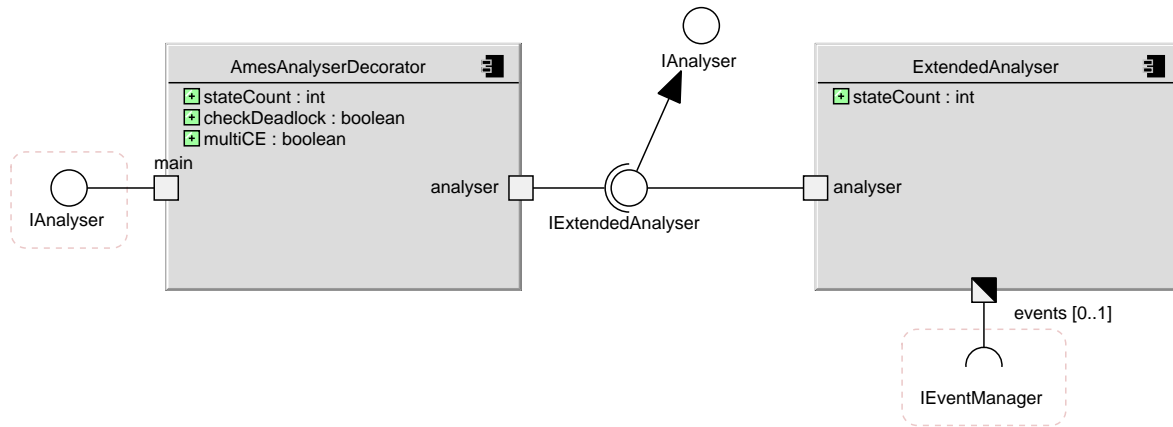
Figure 7.50: The extended analyser and Ames decorator

We then moved the existing analyser logic into the `ExtendedAnalyser` component. This was done by assigning `lts.Analyser` as the implementation class, and did not involve any source code changes at this point. However, the decorator required access to some of the previously private methods of the analyser, so the interface to this component was "widened" by declaring a subinterface called `IExtendedAnalyser`. This did require source code changes and is discussed in further detail below.

Having created these components, were were now able to evolve the `Analyser` component into a composite, allowing it to offer the Ames-specific functionality. This evolution is shown in figure 7.51.
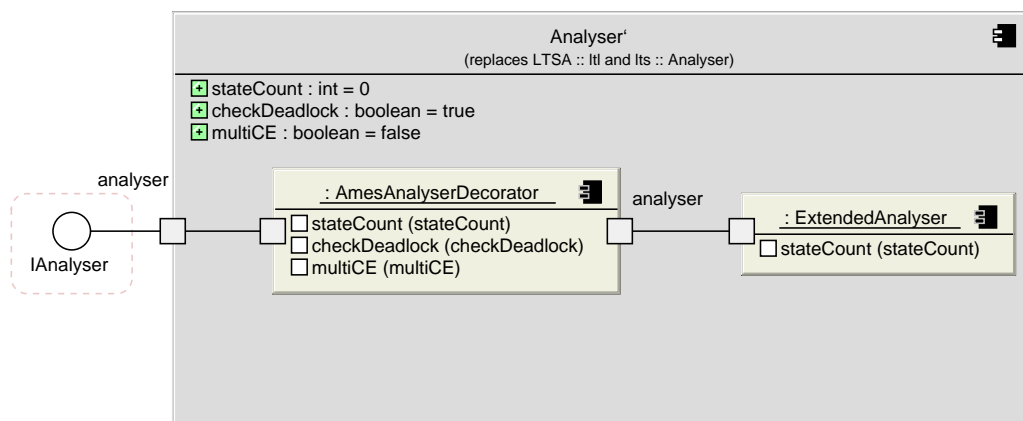
Figure 7.51: Evolving the analyser to add the Ames decorator

Note that two extra attributes were added to `Analyser`: `checkDeadlock` and `multiCE`. These controlled whether or not the analyser checked for deadlocks and produced multiple counterexamples. The attribute defaults allowed the component to be used as it was before.

We then added the extra safety checks to LTSA. Figure 7.52 shows a partial view where a `SafetyCheck` part has been connected to an `AnalyserFactory` with `checkDeadlock` set to false. This provided a safety check that did not detect deadlock. The other check was connected in a similar way.

As discussed earlier, we had to widen the `IAnalyser` interface to accommodate the decorator – the Ames logic required access to a number of the methods that were previously private to the `Analyser`
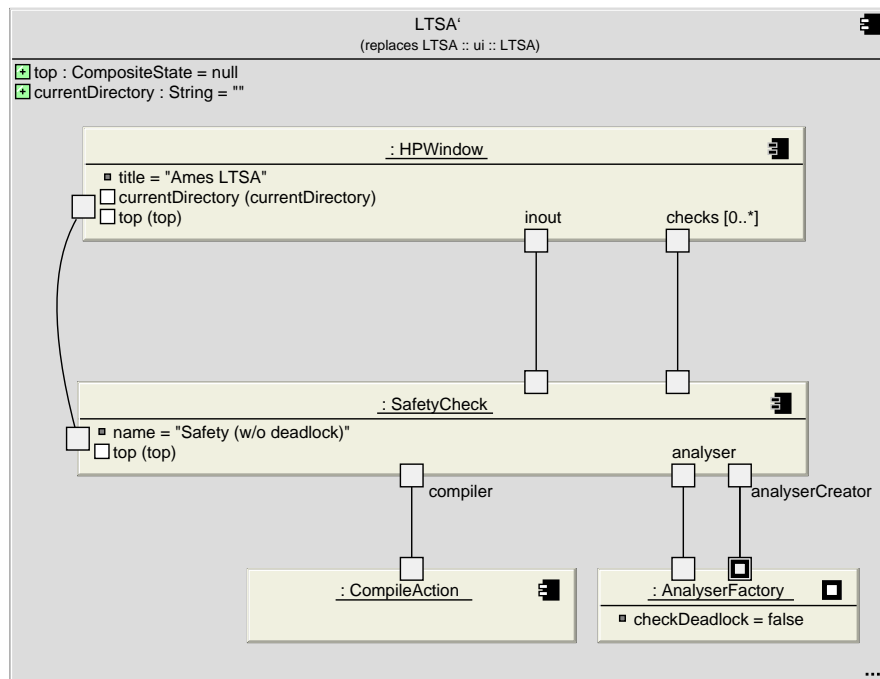
Figure 7.52: Evolving LTSA to include the new safety without deadlock check

component. This involved small changes to the underlying source code of the `Analyser` class, which violated our NoSource requirement. This is the cost of not having a fully elaborated architecture – until the leaf components in the architecture become relatively fine-grained, then unplanned extensions possibly require source code changes to the coarse-grained components in the base.

To test the Ames extension, we used the FSP expression `A = (a -> STOP)` which is guaranteed to result in deadlock. Figure 7.53 shows the original LTSA running a safety check and finding the deadlock, and the Ames variant running the "safety without deadlock check" and not detecting the deadlock.

(a) the original safety check detecting deadlock   (b) the Ames safety check without deadlock
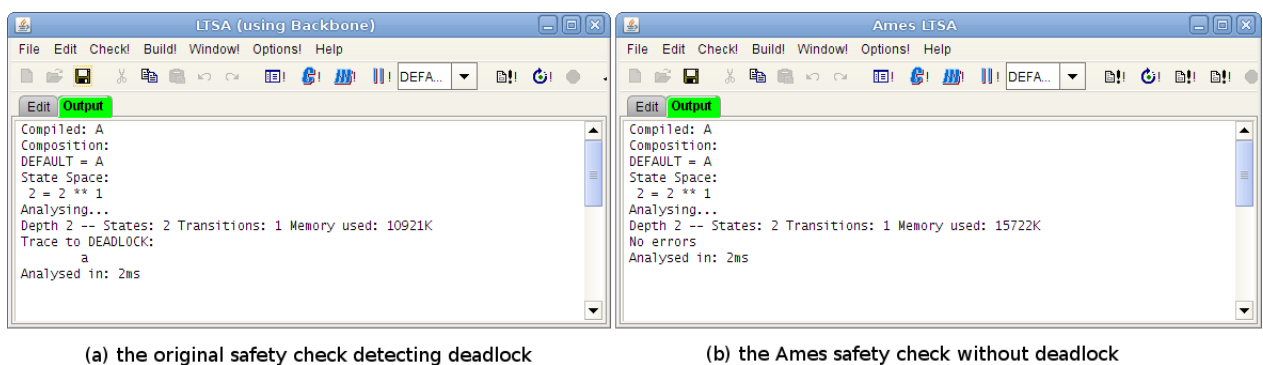
Figure 7.53: The original and Ames safety checks

7.4.5   Combining the Extensions

After packaging the Ames extension in an appropriate extension stratum (`Ames`), we then combined both extensions using a further stratum (`combined`). Figure 7.54 shows the strata diagram. Note that the `combined` stratum depends on both extensions. Running a full check of the system revealed no errors – there were no structural conflicts. As such, we were able to form a system with both the Ames and dual window extensions.
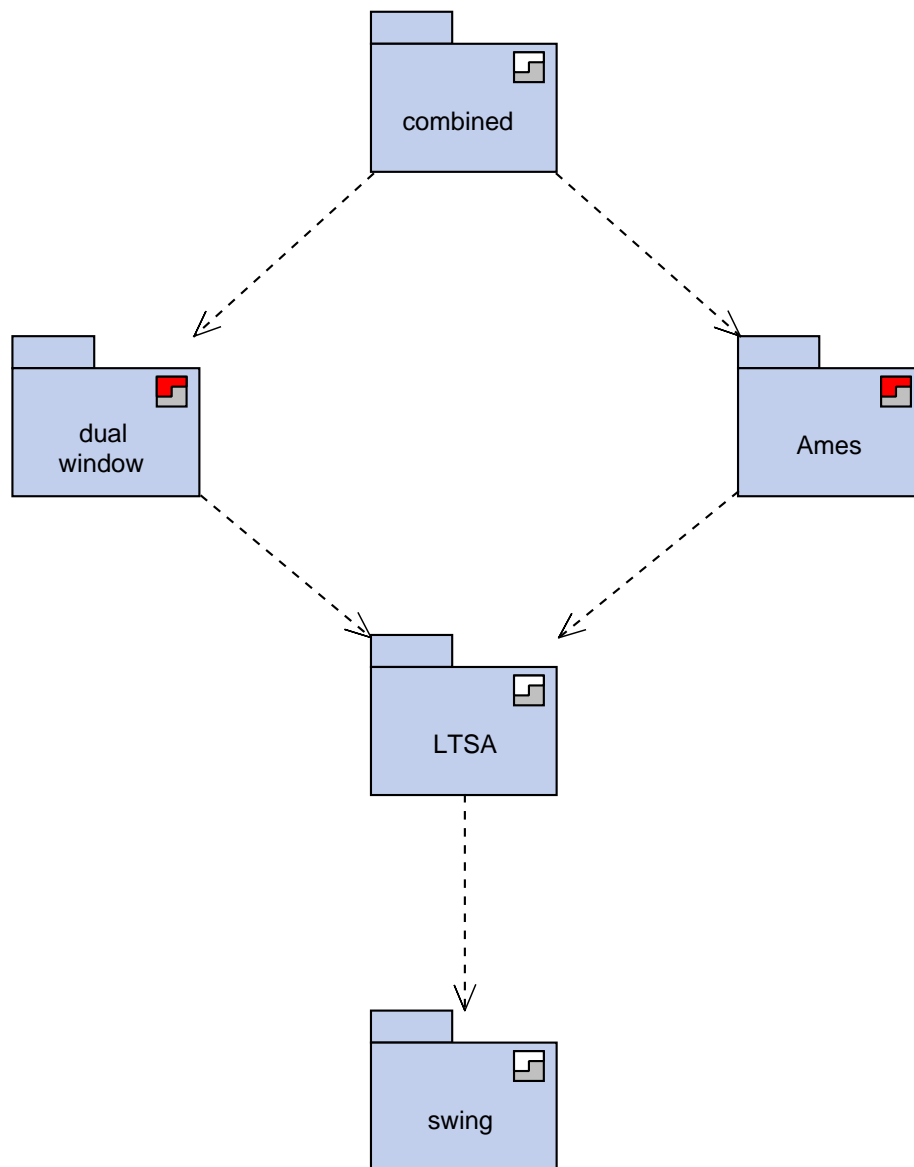
Figure 7.54: The combined strata depends on both extensions

We could then examine the compositional history of the LTSA system over time, as per figure 7.55. In (a), we have the original system after the windows and actions were componentised. Applying the dual window extension resulted in the compositional hierarchy of (b). Applying the AMES extension gave (c). Finally, combining both extensions gave (d).
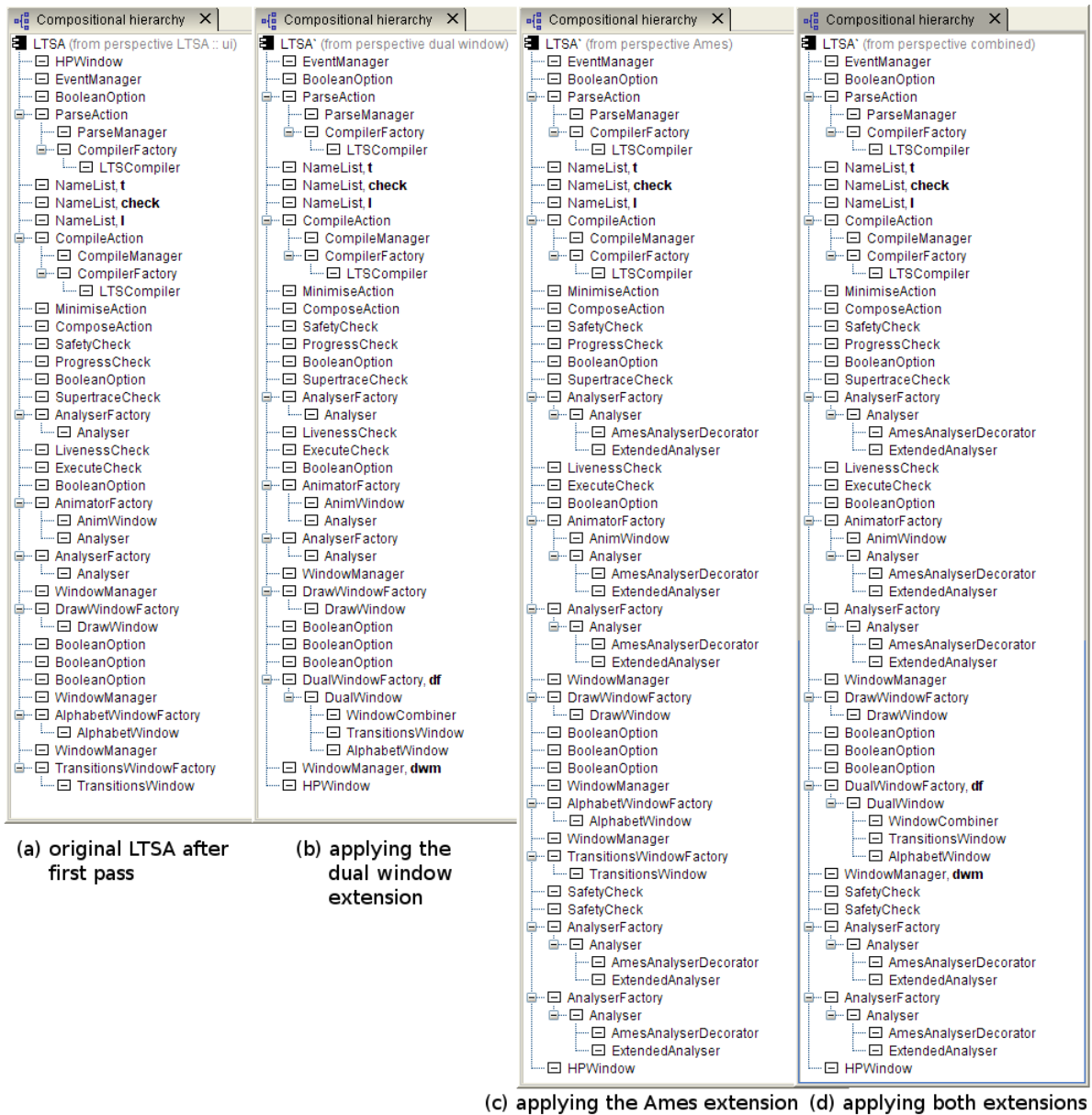
Figure 7.55: The compositional history of the LTSA extensions

7.4.6 Summary: Applying Backbone to LTSA

LTSA is a complex, mature system that provides significant functionality. It has been developed over a number of years in an object-oriented style, and several significant variants of this analysis tool exist in completely separate codebases. Our goal was to express and extend this system using the Backbone approach, in order to demonstrate Backbone's applicability to existing systems that are not already structured as deep component hierarchies.

After a first pass, involving modest effort, we were able to extract components for windows, actions and checks from the LTSA architecture. This enabled us to visualise the architecture in Evolve and run LTSA from within this environment. The compositional hierarchy at this point was shallow and

wide, reflecting the coarse-grained nature of the components.

For the first extension, we replaced the alphabet and transition windows with a window showing both views simultaneously. This was a conservative extension, as it worked with the existing, coarse-grained components without requiring further decomposition. This extension was able to be accomplished without access to the source code, by using the resemblance and replacement constructs.

For the second extension, we modelled the Ames variant of LTSA which introduced two extra safety checks. This variant was created several years ago when researchers forked the source code of the original application. In order to phrase this as an extension we first analysed the differences between the two codebases. It transpired that the extension relied on replacing part of the logic of the analyser component. To accommodate this change we further hierarchically decomposed the component so that the granularity of the architecture matched up with the size of the change required. At this point the compositional hierarchy became deeper, reflecting that some of the components had evolved from leaves to composites.

The two extensions were then able to be combined without structural conflict, resulting in a system that had both a dual alphabet/transitions window and the Ames safety checks.

The Ames extension gave an interesting insight into the applicability of our approach. Implicitly, Backbone assumes that an architecture will be structured as a compositional hierarchy with relatively fine-grained leaf components at the base. The motivation is that this allows small changes to the system to be reflected by a commensurate replacement of a small leaf component. Although this level of granularity is encouraged by the Evolve toolset, existing systems are rarely structured in this way. For these applications, a gradual process of architectural elaboration is more practical, focusing on the decomposition in expected areas of extension. Until a more complete architectural structure is achieved, some changes cannot be accommodated easily without recourse to changing the source code of various base components.

Some unplanned extensions can be handled, however, with a coarse-grained architecture. The first pass LTSA component architecture allowed the dual window extension, which involved addition and removal of parts in the hierarchy. In this respect, Backbone formed an adequate extensibility architecture for the mature application. A benefit of Backbone in this situation, over conventional approaches, is that the ability to handle unplanned change accrues increasingly as the architecture is decomposed more thoroughly.

Our approach to further decomposition is similar to that taken when evolving a legacy system into fine-grained components [MH02]. In this context, system features that have historically exhibited large maintenance costs or have already been shown to vary significantly are focused on and decomposed into a number of small components, facilitating greater reuse between variants.

7.5   Summary

In this chapter, we compared and contrasted Backbone to two commonly used approaches for creating and extending systems: a plugin approach and a compositional technique for describing product lines.

In the first evaluation, we chose to model the addition of columns to the task view in the Eclipse development tool. We examined how the Eclipse plugin architecture would handle this extension requirement. The result was that the change was not foreseen by the designers of this part of Eclipse, and accordingly this necessitated the replacement of a large plugin. This replacement in turn interacted badly with the plugin version system, in that an updated version of the plugin from the Eclipse Foundation would force the later removal of our changes. Effecting the extension turned out to be impractical. In contrast, the Backbone architecture for the same extension allowed the columns to be added via an evolution of the view component.

The Backbone approach separates out the notion of unit of extension (stratum) from the unit of replacement (component or interface). Further, it allows hierarchical decomposition of an architecture which resolves the tension between component size and architectural manageability. The plugin approach, however, fuses the notions of extension and replacement together in the plugin concept and biases plugins towards a coarse granularity for ease of architectural management. This increases the potential for conflict between plugins, resulting in plugins which can never be used together. In contrast, Backbone deals with conflict between strata by allowing another strata to correct the structural issues via component evolution. Conflicting strata therefore do not become mutually exclusive.

For the product line approach, we chose to model the audio desk scenario (section 1.2) using the AHEAD toolkit. This used constants, representing base programs, and allowed functions to refine these to produce system variants. Unfortunately, if two functions or constants conflicted in certain ways, they could not be used together and there was no guarantee that a further function could correct the situation. AHEAD had trouble modelling the mixer upgrade scenario from section 3.3.6, leading to a situation where the extensions could not be combined or corrected. Backbone does not suffer from this limitation, as structural conflicts can always be remedied using evolution in a further extension.

Following on from these evaluations, we presented two case studies examining the applicability of the Backbone approach to other environments, and also to mature systems.

We examined the applicability of Backbone to an unusual environment: the Google Web Toolkit. GWT works by translating Java programs into JavaScript, allowing them to run on the client side in a web browser. We showed that Backbone was able to generate GWT programs, utilising the toolkit library with a small amount of work. The intention was not to show that Backbone provides a superior way of building user interfaces, but to demonstrate that the approach is flexible enough to work in exotic and constrained environments, with existing libraries. The full extensibility benefits of Backbone apply to this environment.

The underlying limitation behind the Backbone approach is that it assumes that the compositional hierarchy of a system will be deep enough to apply any required changes at the appropriate level

of abstraction. For small changes, this implies that leaf components should be fine-grained. Existing systems are not generally structured in this way, although they could often benefit from an extensibility architecture.

As a case study for applying our extensibility approach to a mature system, we restructured LTSA using Backbone. Even with a coarse-grained component architecture, we were able to demonstrate extension of the application. A more complex extension required further component decomposition leading to the insight that Backbone requires a relatively fine-grained hierarchical architecture to accommodate some unplanned changes. If this decomposition is not present, a level of reimplementation effort will be required when creating an extension. Ideally, gradual hierarchical decomposition will occur as part of the natural elaboration of the architecture, and the ability to handle unplanned extension will start to accrue accordingly.

Chapter 8

Conclusions

The thesis presents an architectural approach for extensible applications, allowing a system to be extended and customised without direct access to its implementation code. This chapter summarises the contributions of our approach and outlines future work.

8.1 Contributions

8.1.1 A Formally Specified, Structural Technique for Application Extensibility

The primary contribution of this work is to describe and formally specify a design-time technique for extending an application by manipulating the hierarchical structure of its architecture. This technique accommodates both planned extension, where the original application developers had foreseen the type of changes required, and unplanned extension where they had not.

A key point of differentiation from other approaches is that extensibility does not need to be explicitly factored into an architecture, but instead accrues as the architecture is naturally elaborated into deeper hierarchical structures.

Extensibility is achieved whilst satisfying or ameliorating a number of requirements which relate to the impact of any extension changes upon other developers. These requirements capture the forces and constraints between developers involved in common extension scenarios. Existing techniques for creating variants, such as plugin architectures, product lines and aspect-oriented approaches, do not address these issues in a satisfactory way.

Architectural restructuring and manipulation is enabled by adding three concepts to a conventional UML2-based hierarchical ADL. Resemblance allows a new component to be defined in terms of structural similarity to other components. Replacement substitutes one component definition for another, allowing for structural change at multiple places in a compositional hierarchy. A stratum groups related definitions, forming a unit of extension. Together, these facilities allow the compositional hierarchy of

193

an application to be restructured incrementally to accommodate architectural changes for new requirements. The same constructs can be used to correct any well-formedness conflicts that are detected when combining independently developed extensions that share a common base.

As the approach works by treating a system as a hierarchically structured component, it is equally applicable to the reuse of components within an architecture. As well as providing for extensibility, therefore, the constructs are useful for general software construction.

The three extensibility concepts are formally specified using the Alloy logic language, without reference to the component model which is specified on top of this foundation. As such, replacement and resemblance are general concepts that can be applied to elements other than components. In the approach, we also allow interfaces to be replaced and resembled to describe the evolution and substitutability of services in an architecture.

8.1.2   A Modelling Environment for Application Creation and Extension

A secondary contribution is the provision of a modelling environment and runtime execution platform based directly on the formal specification. The use of the formal specification has proven to be invaluable in avoiding the corner cases and inconsistencies associated with previous implementations.

The modelling environment integrates the change model of our approach with UML2 composite structure diagrams, providing a familiar and intuitive workbench for application creation and extension. Strata import and export facilitate the sharing that occurs in an extension setting. Unlike many other techniques for handling architectural variation, the link between architectural components and implementation classes is captured naturally as part of the wider approach.

The runtime platform uses the mapping from leaf components to underlying implementation classes to instantiate and execute an application.

8.2   Critical Review Against Requirements

In section 1.1, we established the research question for this area of work.

> Can we devise an architectural approach to software that naturally builds extensibility into
> a system as it is constructed, which also respects the underlying forces and constraints
> between extension and base application developers?

To answer this question, we examined an extensibility scenario and established seven requirements that an ideal extensibility approach should satisfy. We now review our approach against these.

ALTER The compositional structure of an application can be arbitrarily adjusted using replacement and resemblance, allowing alterations to be made for both planned and unplanned changes. The

key factor determining whether the architectural alterations required are proportionate to the size of the functionality added, replaced or removed is the granularity of component decomposition.

NoSource Implementation source code of the base application is not required for extension. Source code does not have to be changed to handle replacement as the architectural description and run-time platform control component instantiation. If components are too coarse-grained, however, replacement can incur a cost penalty relative to the size of the added or changed functionality.

NoImpact No base or extension developer is affected by an extension, unless they choose to include it in their model. As the approach can handle unplanned extension, an extension developer is able to make changes without placing the architectural burden on the base application developers.

Combine Alterations made using replacement and resemblance are held as deltas against base components, allowing extensions to be combined. Combining extensions merges the deltas for each component using the strata dependency graph to control application order.

Upgrade An upgrade can be phrased as an extension, and combined with other extensions. Conflict can be detected in the same way, and corrected using a further extension. Alternatively, an upgrade can be performed by destructively modifying the base architecture. In this case the use of UUIDs to preserve the logical identity of components and their internal constituents reduces the disruption to existing extensions.

DetectAndCorrect Structural conflict between extensions occurs when two independently developed extensions make incompatible changes to a common base architecture. This can be detected by a set of structural rules, and corrected by evolutions in a further extension. The use of UUIDs for logical identity removes the possibility of naming conflicts between extensions.

AppropriateLevel The use of a compositional hierarchy resolves the tension between having fine-grained components for ease of extension, and having coarse-grained components for ease of architectural management. An extension can be phrased at the appropriate level of composition, whilst permitting access to deeper levels when required.

The answer to the research question is in the affirmative. It is possible to align the forces of application creation and extension in an architectural setting, such that each activity benefits the other. In the Backbone approach, natural points of extension arise as hierarchical component construction of the base application takes place, and extensions can take advantage of this to remake the architecture as required. Conversely, the constructs added to allow extensibility are useful in their own right for general software construction.

A key insight to emerge from the case studies and evaluation is that an architecture must be decomposed to a relatively fine-grained level in order to accommodate small extension changes without incurring a disproportionate cost due to replacement. In practice, the granularity of components will be expressed at a level which is natural for a given system, and can never be perfectly aligned to every possible change. This invariably leads to some development overhead when creating an extension.

In mature systems that are not structured as deep component hierarchies, gradual decomposition of the architecture is recommended as a way of easing the approach into the architecture. This requires modification of the base source until the decomposition is fine-grained enough, but this is an architectural elaboration activity rather than a predictive exercise in adding extension points.

8.3   Future Work

This section examines future work in the area of extensible applications, building on the structural foundation provided by our approach.

8.3.1   Behavioural Checking of Extensions

Our work so far has focused on a structural approach to extensibility, including the detection and correction of conflict between extensions. We plan to build on this foundation, by incorporating existing research on the behavioural analysis of component systems. The aim is to allow the behaviour of extensions to be specified, and their interaction with the base application and other extensions to be analysed for certain properties. This will also allow concurrent component behaviour to be modelled.

The motivation is that whilst resolving structural conflict is necessary for extensions to work together, it is not sufficient to guarantee correct behaviour of the extended system. As shown in [HA00], behavioural conflicts can result at both a protocol and overall goal level after functionality has been added, even if a system is notionally structurally correct.

Component-based behavioural analysis has a rich research history and been considered in depth in an architectural context [AP04, AP05, MK06, Med99]. Protocol checking in extensible systems has been considered in a plugin context in [CEK+04], although this does not cover hierarchical composition and does not provide a particularly intuitive way to specify behaviour.

In keeping with a practical, engineering approach, we plan to allow protocols to be specified at the leaf component level using UML2 sequence diagrams [Obj09e]. A sequence chart can have looping, parallel, alternate and optional regions in the style of [PV02], which can then be used to form a conservative "upper bound" of worst-case component behaviour. The protocols of composite components can be formed automatically from the leaf protocols using composition and hiding [MK06]. In this way, the behavioural properties of a system can be analysed automatically by composing protocols after an extension has been applied.

We have done some initial work towards an automated translation from sequence diagrams into FSP expressions. The interfaces provided and required by the ports of a component are modelled as separate actors, as is the component itself, allowing interactions between ports to be represented. Figure 8.1 shows the protocol of an alarm component modelled using an earlier incarnation of Evolve, and the resultant LTS after this was composed with a timer leaf component to form an alarm clock system.

We have not yet determined a suitable form for expressing the behavioural goals of an extension.
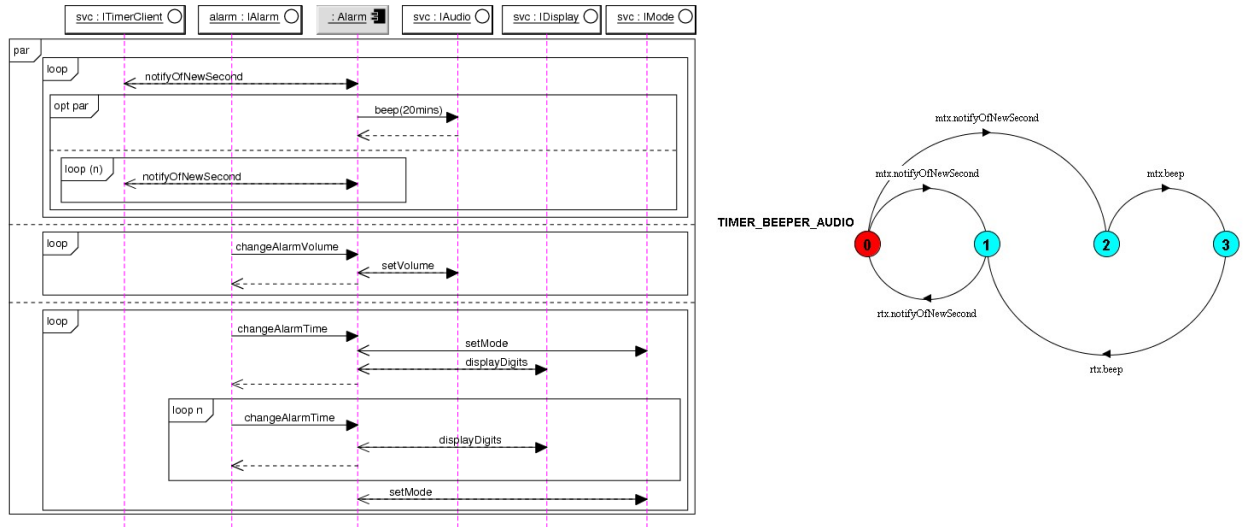
Figure 8.1: A component protocol and LTS after composition

8.3.2 Extensible Feature Modelling in Backbone

Currently there is no consideration in Backbone for the link between requirements and the elements of an architecture. To rectify this we plan to incorporate feature modelling into our approach, where an extension can define features which refer to associated components and interfaces. Features can be composed of other features, allowing feature trees to be built in a similar way to feature-oriented domain analysis techniques [KCH+90].

In keeping with our extensibility approach, we plan to represent features as elements which can therefore be replaced and resembled. This will allow the compositional structure of the feature tree to be adjusted via extensions. We anticipate that this will ameliorate the current need to specify a feature tree with optional and mandatory elements for an entire domain up front, allowing a more gradual approach to feature modelling

8.3.3 Baselining For Delta Compression

Backbone allows extensions to be built on top of other extensions, making the concept of base application a relative one. This can create a situation where the original base is buried under many layers of deltas. To address this, we plan to specify and develop a way to merge the original base and a number of extensions into a new base application with no deltas. This is known as a baseline.

Investigation into a baselining operation revealed an intriguing possibility: by merging an application and an extension and placing the result into what was previously the extension stratum, we can use the original base stratum as an extension which reverts the upgraded system back to its original structure. This allows the architecture to move forward, but retain the previous incarnation of the system as an extension for maintenance and backwards compatibility purposes.

(a) a base with extension   (b) baselining adjusts the previous extension   (c) shows how a new extension, built on system2.0,
                                            stratum to be the new base, and the old base   can be applied to system1.0 via the combine stratum
                                            to be an extension reverting back to v1.0
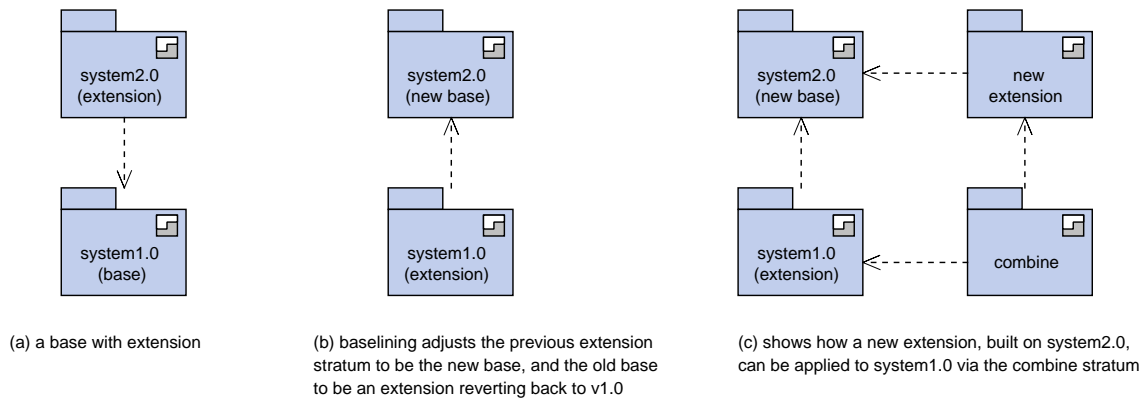
Figure 8.2: Baselining reverses the relationship between a base and an extension

The effect of this on a strata graph is shown in figure 8.2. In (a) the original `system1.0` base is extended by applying the deltas from `system2.0`. In (b), after baselining, the extension and the base have been merged, and the new application without deltas is stored in the `system2.0` stratum. That stratum is the new base and the `system1.0` stratum now holds an automatically created extension containing deltas that revert the new base back to the original 1.0 system. Note that the dependency has been reversed. The `combine` extension in (c) applies a new extension built on the new base, to the old base. Any structural issues that result can be corrected in the `combine` extension stratum.

8.3.4   Integration with Existing Module Systems

Conflict in Backbone occurs when independently developed strata, operating on the same base, evolve a base element in incompatible ways. Our approach deals with this conflict by allowing a further stratum to correct errors, resulting in an application-wide consensus on the final structure of all components.

Existing module systems used in industry, such as OSGi [OSG03], cannot correct structure in this way and instead permit multiple versions of a component to run in parallel under certain circumstances. Whilst this causes obvious problems if a single instance of a component must be shared widely throughout an application, it is useful for handling different versions of library infrastructure such as XML parsers.

We plan to integrate our system with OSGi in the future, and are investigating a way to allow Backbone to represent parallel versions where required. A "frozen" stereotype has been considered for constituent parts of a component in order to prevent further extensions being applied to the part's type. This facility could be added to or removed from a part by an evolution in an extension. Further work is needed to determine if this will allow Backbone and OSGi to interoperate effectively.

OSGi allows bundles (modules) to be dynamically loaded and unloaded whilst a program is running. As part of the integration, we plan to investigate whether Backbone can be used for runtime architectural changes in addition to its current handling of design-time changes.

8.3.5   Applying Backbone to Evolve and the Runtime Platform

Although our extensibility ideas arose from experience creating and extending earlier versions of Evolve, neither the modelling tool nor the runtime are currently structured as Backbone architectures. This was the result of a bootstrapping dilemma: we required an existing tool to develop and refine the approach, but the approach was not yet in existence when the applications were developed.

We plan to restructure both the modelling tool and runtime platform using Backbone. The benefit to the modelling tool is that Backbone can then serve as the extension architecture. The benefit of expressing the runtime using Backbone is that it will let applications extend and customise the runtime behaviour of the platform. We further plan to allow user-defined stereotypes with expansion rules and visual customisations.

8.4   Closing Remarks

Creating an extensible system can be a complex and difficult endeavour. Current techniques require a great deal of insight into possible future changes, and this type of predictive work results in a large amount of development overhead. There is also no guarantee that the predictions will be accurate, and future requirements can be difficult to realise even in an extensible system that notionally provides much of the functionality already. These limitations are a concern, because in several other respects application extensibility has proven to be an effective technique for delivering a system which can be quickly adapted for new needs.

We have described an approach where extensibility is naturally built into a system as it is constructed. The same elements that form the building blocks of development also form the units of replacement, allowing alterations to be performed on an architecture to restructure it in ways which were not necessarily planned for. This is a robust, architectural approach that avoids the need to predict future requirements, thereby simplifying the task of creating extensible systems.

Bibliography

[ACN02]   J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture
            to Implementation. In *ICSE '02: Proceedings of the 24th International Conference
            on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[AG94]   R. Allen and D. Garlan. Beyond Definition/Use: Architectural Interconnection. *Pro-
            ceedings of the workshop on Interface Definition Languages*, 29(8):35–45, 1994.

[Agh86]   G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT
            Press, Cambridge, MA, USA, 1986.

[Ald08]   J. Aldrich. Using Types to Enforce Architectural Structure. In *WICSA '08: Pro-
            ceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture
            (WICSA 2008)*, pages 211–220, Washington, DC, USA, 2008. IEEE Computer Soci-
            ety.

[AP03]   M. Alanen and I. Porres. Difference and Union of Models. In P. Stevens, Whittle.J,
            and J. Booch, editors, *UML2003*. Springer-Verlag, 2003.

[AP04]   J. Adamek and F. Plasil. Partial Bindings of Components - Any Harm? In *Proceedings
            of the 11th Asia-Pacific Software Engineering Conference (APSEC'04) - Volume 00*,
            pages 632–639. IEEE Computer Society, 2004.

[AP05]   J. Adamek and F. Plasil. Component Composition Errors and Update Atomicity:
            Static Analysis. *Journal of Software Maintenance and Evolution: Research and Prac-
            tice*, 17(5):363–377, 2005.

[Asp03]   AspectJ Team. *The AspectJ Programming Guide*, chapter 1. Getting Started with
            AspectJ. Xerox Corporation, 2002-2003.

[AZB06]   P. Avgeriou, U. Zdun, and I. Borne. Architecture-Centric Evolution: New Issues and
            Trends. In *ECOOP Workshops*, pages 97–105, 2006.

[Bar09]   J. Barnabe. Avoiding Extension Conflicts in Firefox.
            http://blog.userstyles.org/2007/02/06/avoiding-extension-conflicts/, last accessed
            August 2009.

[Bat05]   D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pages 7–20, 2005.

[Bat06]   D. Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2006.

[BBG05]   S. Bullen, R. Bovey, and J. Green. *Professional Excel Development: The Definitive Guide to Developing Applications Using Microsoft Excel and VBA*, chapter 5: Function, General and Application-Specific Add-Ins, pages 109–142. Addison Wesley, 2005.

[BBHL04]   K. Bennett, D. Budgen, T. Hoare, and P. Layzell. Software Evolution. In *Grand Challenges for Computing Research Conference Announcement*, University of Newcastle upon Tyne, 2004.

[BC90]   G. Bracha and W. Cook. Mixin-Based Inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[BCD+04]   O. Barais, E. Cariou, L. Duchien, N. Pessemier, and L. Seinturier. TranSAT: A Framework for the Specification of Software Architecture Evolution. In *ECOOP First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04)*, Oslo, Norway, jun 2004.

[Bea09]   W. Beaton. Eclipse Hints, Tips, and Random Musings. http://dev.eclipse.org/blogs/wayne/2005/10/03/fun-with-combinatorics/, last accessed August 2009.

[Bed00]   B. B. Bederson. Fisheye Menus. In *UIST '00: Proceedings of the 13th annual ACM Symposium on User Interface Software and Technology*, pages 217–225, New York, NY, USA, 2000. ACM.

[BGG04]   H. Bauerdick, M. Gogolla, and F. Gutsche. Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. In T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 188–196. Springer, 2004.

[Bir05]   D. Birsan. On Plug-Ins and Extensible Architectures. *ACM Queue*, 3(2):40–46, 2005.

[BJMvH02]   D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[Bla09]   J. Blandy. *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*, chapter 11: GNU Emacs: Creeping Featurism Is a Strength, pages 263–278. O'Reilly Media, Inc., 2009.

[BLHM02] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating Product-Lines of Product-Families. In *ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, page 81, Washington, DC, USA, 2002. IEEE Computer Society.

[BLMDL06] O. Barais, A.-F. Le Meur, L. Duchien, and J. Lawall. Safe Integration of New Concerns in a Software Architecture. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 52–64, Washington, DC, USA, 2006. IEEE Computer Society.

[BLS03] D. Batory, J. Liu, and J. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *Proceedings of the 9th European Software Engineering Conference*, pages 48–57, Helsinki, Finland, 2003. ACM Press.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, and P. Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons; 1st edition (August 8, 1996), 1996.

[Bos99] J. Bosch. Superimposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5):257–273, 25 March 1999.

[Box97] D. Box. *Essential COM*. Addison-Wesley Professional, 1997.

[BPV98] K. B. Bruce, L. Petersen, and J. Vanderwaart. Modules in LOOM: Classes are Not Enough (extended abstract). Technical report, Williams College, 1998.

[BSR03] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[Bud03] D. Budgen. *Software Design*. Addison Wesley, 2nd edition, July 2003.

[BW99] M. Buchi and W. Weck. The Greybox Approach: When Blackbox Specifications Hide Too Much. Technical report, Turku Centre for Computer Science, 1999.

[CCG$^+$04] P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, and A. van der Hoek. Differencing and Merging Within an Evolving Product Line Architecture. *Software Product-Family Engineering*, 3014:269–281, 2004.

[CDF$^+$95] S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M. Sloman, and K. Twidle. Configuration Management for Distributed Software Services. In *Proceedings of the Fourth International Symposium on Integrated Network Management*, pages 29–42, London, UK, UK, 1995. Chapman & Hall, Ltd.

[CDHSV97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM*, 40(10):70–77, 1997.

[CEK⁺04]   R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel. Predictable Dynamic Plugin Systems. In *FASE 2004 : Fundamental Approaches to Software Engineering*, pages 129–143, 2004.

[CEM03]   R. Chatley, S. Eisenbach, and J. Magee. Modelling a Framework for Plugins. In *Specification and Verification of Component-Based Systems*, 2003.

[CEM04]   R. Chatley, S. Eisenbach, and J. Magee. MagicBeans: A Platform for Deploying Plugin Components. *Component Deployment*, 3083:97–112, 2004.

[CFL03]   M. Cortes, M. Fontoura, and C. Lucena. Using Refactoring and Unification Rules to Assist Framework Evolution. *ACM SIGSOFT Software Engineering Notes*, 28(6):1–1, 2003.

[CH01]   B. Councill and G. T. Heineman. Definition of a Software Component and its Elements. In *Component-Based Software Engineering: Putting the Pieces Together*, pages 5–19, Boston, MA, USA, 2001. Addison-Wesley Longman Publishing Co., Inc.

[Che06]   R. Chern. Refactoring with Difference-Based Modules: An Experience Report. In *CPSC 511 Mini Conference*, 2006.

[CN91]   B. Cox and A. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[CN01]   P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[DAH⁺07]   E. Dashofy, H. Asuncion, S. Hendrickson, G. Suryanarayana, J. Georgas, and R. Taylor. ArchStudio 4: An Architecture-Based Meta-Modeling Environment. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 67–68, Washington, DC, USA, 2007. IEEE Computer Society.

[Das09]   E. M. Dashofy. Variants Extension for xADL 2.0. http://www.isr.uci.edu/projects/xarchuci/ext-overview.html, last accessed August 2009.

[Des09]   DesktopLinux.com. Firefox 1.5 Upgrade Brings Extension Headaches. http://www.desktoplinux.com/news/NS2432314568.html, last accessed 2009.

[DFCU08]   N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. MTSA: The Modal Transition System Analyser. In *IEEE/ACM Automated Software Engineering*, pages 475–476. IEEE, 2008.

[DK09]   L. DeMichiel and M. Keith. JSR-000220 Enterprise JavaBeans 3.0. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, last accessed August 2009.

[DSE97]   DSE, Imperial College. The Darwin Language, Version 3d. Technical report, Imperial
             College, 1997.

[EBB06]   M. Eriksson, J. Börstler, and K. Borg. Software Product Line Modeling Made Prac-
             tical. *Communications of the ACM*, 49(12):49–54, 2006.

[Ecl09a]   Eclipse Foundation. Platform Plug-in Developer Guide: OSGi Bundle Manifest Head-
             ers. http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.platform.doc.isv/
             reference/misc/bundle_manifest.html, last accessed August 2009.

[Ecl09b]   Eclipse   Foundation.   Rich   client   platform   (RCP)   applications.
             http://www.eclipse.org/community/rcp.php, last accessed August 2009.

[EH07]   C. Escoffier and R. S. Hall. Dynamically Adaptable Applications with iPOJO Service
             Components. In *6th International Symposium on Software Composition (SC 2007)*,
             2007.

[EHTE97]   G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A View-Oriented Approach to
             System Modelling Based on Graph Transformation. In *ESEC '97/FSE-5: Proceedings
             of the 6th European Conference*, pages 327–343, New York, NY, USA, 1997. Springer-
             Verlag New York, Inc.

[EK99]   A. Evans and S. Kent. Meta-Modelling Semantics of UML: The pUML Approach.
             In *2nd International Conference on the Unified Modeling Language*, pages 140–155,
             January 1999.

[Erd98]   H. Erdogmus. Representing Architectural Evolution. In *Proceedings of the 1998
             Conference of the Centre for Advanced Studies on Collaborative Research*, pages 11–,
             Toronto, Ontario, Canada, 1998. IBM Press.

[FCS$^+$08] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia,
             S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving Software
             Product Lines with Aspects: An Empirical Study on Design Stability. In *ICSE '08:
             Proceedings of the 30th International Conference on Software Engineering*, pages 261–
             270, New York, NY, USA, 2008. ACM.

[FF98]   R. B. Findler and M. Flatt. Modular Object-Oriented Programming with Units
             and Mixins. In *ICFP '98: Proceedings of the third ACM SIGPLAN International
             Conference on Functional Programming*, pages 94–104, New York, NY, USA, 1998.
             ACM.

[FHLS97]   G. Froehlich, J. Hoover, L. Liu, and P. Sorenson. Hooking into Object-Oriented
             Application Frameworks. In *Proceedings of the 19th International Conference on
             Software Engineering*, pages 491–501, Boston, Massachusetts, United States, 1997.
             ACM Press.

[FKN$^+$92]   A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, 1992.

[Fow09]   M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern. http://www.martinfowler.com/articles/injection.html, last accessed August 2009.

[FS97]   M. Fayad and D. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[GA03]   M. Goulo and F. Abreu. Bridging the Gap between Acme and UML 2.0 for CBD. In *Specification and Verification of Component-Based Systems (SAVCBS 2003)*, 2003.

[GB03]   E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[GBR03]   M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL Models by Automatic Snapshot Generation. In *UML*, pages 265–279, 2003.

[GBR07]   M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[GHJJ95]   E. Gamma, R. Helm, R. Johnson, and V. J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GMW97]   D. Garlan, R. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *CASCON '97: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[Goo09a]   Google. Google Web Toolkit. http://code.google.com/webtoolkit/, last accessed August 2009.

[Goo09b]   Google. Google Web Toolkit Application Gallery – Google Code. http://gwtgallery.appspot.com/, last accessed August 2009.

[GSC$^+$04]   J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley; 1st edition (August 16, 2004), 2004.

[HA00]   J. Hay and J. Atlee. Composing Features and Resolving Interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 110–119, New York, NY, USA, 2000. ACM Press.

[Har87]   D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hea09]   Headway Software Technologies Ltd. Structure101 - Software Architecture Analysis for Java, C/C++ and Anything. http://www.headwaysoftware.com/index.php, last accessed August 2009.

[HH01]   D. Hou and J. Hoover. Towards Specifying Constraints for Object-Oriented Frameworks. In *Proceedings of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research*, page 5, Toronto, Ontario, Canada, 2001. IBM Press.

[Hol93]   U. Holzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 36–56. Springer-Verlag, 1993.

[HSGP06]   B. Henderson-Sellers and C. Gonzalez-Perez. Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0. In *Model Driven Engineering Languages and Systems*, pages 16–26. Springer, 2006.

[HvdH07]   S. A. Hendrickson and A. van der Hoek. Modeling Product Line Architectures Through Change Sets and Relationships. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[HWL+02]   M. Hoy, D. Wood, M. Loy, J. Elliot, and R. Eckstein. *Java Swing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[IBM09a]   IBM. Rational Application Developer for WebSphere Software. http://www-01.ibm.com/software/awdtools/developer/application/, last accessed August 2009.

[IBM09b]   IBM. Rational Software Architect for WebSphere Software. http://www-01.ibm.com/software/awdtools/swarchitect/websphere/, last accessed August 2009.

[Ich02]   Y. Ichisugi. Layered Class Diagram –The Way to Express Class Structures of MixJuice Programs. Technical report, National Institute of Advanced Industrial Science and Technology (AIST), 2002.

[IT02]   Y. Ichisugi and A. Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 62–88, London, UK, 2002. Springer-Verlag.

[Jac02]   D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[Jac06]   D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, April 2006.

[Jac09]   D. Jackson. Alloy Home Page. http://alloy.mit.edu/, last accessed August 2009.

[JHA+05]   R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and C. Sampaleanu. *Professional Java Development with the Spring Framework*. Hungry Minds Inc, U.S., 2005.

[Joh97a]   R. Johnson. Components, Frameworks, Patterns. In *Proceedings of the 1997 Symposium on Software Reusability*, pages 10–17, Boston, Massachusetts, United States, 1997. ACM Press.

[Joh97b]   R. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[Jor04]   B. Jorgensen. Language Support for Incremental Integration of Independently Developed Components in Java. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1316–1322, New York, NY, USA, 2004. ACM Press.

[JS03]   J. A. Jacko and A. Sears, editors. *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 2003.

[KAG$^+$06]   U. Kulesza, V. Alves, A. Garcia, C. de Lucena, and P. Borba. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In *Lecture Notes in Computer Science*, pages 231–245. Springer, 2006.

[KAJ$^+$93]   G. Kiczales, J. M. Ashley, L. H. R. Jr., A. Vahdat, and D. G. Bobrow. *Object-Oriented Programming: The CLOS Perspective*, chapter 4: Metaobject Protocols: Why We Want Them and What Else Can They Do, pages 101–131. MIT Press, 1993.

[KAK08]   C. Kastner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 311–320, New York, NY, USA, 2008. ACM.

[KCH$^+$90]   K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[KdRB91]   G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[KF98]   S. Krishnamurthi and M. Felleisen. Toward a Formal Theory of Extensible Software. In *In SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 88–98. ACM Press, 1998.

[KGO$^+$01]   R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. Taylor. xADL: Enabling Architecture-Centric Tool Integration with XML. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*, page 9053, Washington, DC, USA, 2001. IEEE Computer Society.

[KL92]   G. Kiczales and J. Lamping. Issues in the Design and Specification of Class Libraries. In *ACM SIGPLAN Notices*, pages 435–451, Vancouver, British Columbia, Canada, 1992. ACM Press.

[KLM⁺97]   G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KM90]   J. Kramer and J. Magee. The Evolving Philosophers Problem - Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[KME95]   J. Kramer, J. Magee, and S. Eisenbach. Modelling Darwin in pi-Calculus. In *Theory and Practice in Distributed Systems*, Lecture Notes in Computer Science, pages 133–152. Springer-Verlag, July 1995.

[KMND00]   K. Kramer, J. Magee, K. N., and N. Dulay. *Software Architecture for Product Families: Principles and Practice*, chapter 2. Software Architecture Description, pages 31–65. Addison Wesley, 2000.

[KMS89]   J. Kramer, J. Magee, and M. Sloman. Configuration Support for System Description, Construction and Evolution. In *Proceedings of the 5th International Workshop on Software Specification and Design*, pages 28–33, Pittsburgh, Pennsylvania, United States, 1989. ACM Press.

[Kru06]   C. W. Krueger. New methods in Software Product Line Practice. *Communications of the ACM*, 49(12):37–40, 2006.

[LMMPN93]   O. Lehrmann Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Association for Computing Machinery, 1993.

[LMW00]   A. Lai, G. Murphy, and R. Walker. Separating Concerns with Hyper/J: An Experience Report. In *Multi-Dimensional Separation of Concerns in Software Engineering*, pages 79–91, 2000.

[LW08]   P. Li and E. Wohlstadter. View-Based Maintenance of Graphical User Interfaces. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, pages 156–167, New York, NY, USA, 2008. ACM.

[Mar00]   R. C. Martin. *More C++ Gems*, chapter The Open-Closed principle, pages 97–112. Cambridge University Press, New York, NY, USA, 2000.

[MB97]   M. Mattsson and J. Bosch. Framework Composition: Problems, Causes and Solutions. In *Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems*, pages 203–. IEEE Computer Society, 1997.

[McI68]   M. McIlroy. *Mass Produced Software Components: NATO Science Committee Report*. NATO, 1968.

[McV09]   A. McVeigh.   The Rich Engineering Heritage Behind Dependency Injection.
            http://www.javalobby.org/articles/di-heritage, last accessed August 2009.

[MDEK95]   J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software
            Architectures. In W. Schafer and P. Botella, editors, *Proceedings of the 5th European
            Software Engineering Conference*, volume 989, pages 137–153, Sitges, Spain, 1995.
            Springer-Verlag, Berlin.

[Med96]   N. Medvidovic. ADLs and Dynamic Architecture Changes. In *Joint Proceedings of
            the Second international Software Architecture Workshop (ISAW-2) and International
            Workshop on Multiple Perspectives in Software Development (Viewpoints '96)*, pages
            24–27, San Francisco, California, United States, 1996. ACM Press.

[Med99]   N. Medvidovic. *Architecture-Based Specification-Time Software Evolution*. PhD the-
            sis, University of California, Irvine, 1999. Taylor, Richard N.

[Mey97]   B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle
            River, NJ, USA, 1997.

[MH02]   A. Mehta and G. Heineman. Evolving Legacy System Features into Fine-Grained
            Components. In *24th International Conference on Software Engineering*, 2002.

[Mic09a]   Microsoft.   COM: Component   Object   Model   Technologies.
            http://www.microsoft.com/com/default.mspx, last accessed August 2009.

[Mic09b]   Microsoft.   Microsoft   Office   Online: Excel   2003   Home   Page.
            http://office.microsoft.com/en-gb/FX010858001033.aspx,   last   accessed   August
            2009.

[Mil99]   R. Milner. *Communicating and Mobile Systems: the pi-Calculus*. Cambridge Univer-
            sity Press, 1999.

[MK96]   J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings
            of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages
            3–14, San Francisco, California, United States, 1996. ACM Press.

[MK06]   J. Magee and J. Kramer. *Concurrency (State Models and Java Programs)*. John
            Wiley and Sons Ltd, 2nd edition, 2006.

[MMK06]   A. McVeigh, J. Magee, and J. Kramer. Using Resemblance to Support Component
            Reuse and Evolution. In *Specification and Verification of Component Based Systems
            Workshop*, 2006.

[MMS03]   J. Mayer, I. Melzer, and F. Schweiggert. Lightweight Plug-In-Based Application
            Development. In *NODe '02: Revised Papers from the International Conference Ne-
            tObjectDays on Objects, Components, Architectures, Services, and Applications for a
            Networked World*, pages 87–102, London, UK, 2003. Springer-Verlag.

[MORT96]   N. Medvidovic, P. Oreizy, J. Robbins, and R. Taylor. Using Object-Oriented Typing
               to Support Architectural Design in the C2 Style. In *Proceedings of the 4th ACM
               SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32, San
               Francisco, California, United States, 1996. ACM Press.

[Moz09a]   Mozilla   Developer   Center.   Firefox   Extensions.
               http://developer.mozilla.org/en/docs/Extensions, last accessed August 2009.

[Moz09b]   Mozilla Developer Center. Firefox Plugins. http://developer.mozilla.org/en/docs/
               Plugins, last accessed August 2009.

[MRT99]   N. Medvidovic, D. Rosenblum, and R. Taylor. A Language and Environment for
               Architecture-Based Software Development and Evolution. In *ICSE '99: Proceed-
               ings of the 21st International Conference on Software Engineering*, pages 44–53, Los
               Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[MS02]   E. Meijer and C. Szyperski. Overcoming Independent Extensibility Challenges. *Com-
               munications of the ACM*, 45(10):41–44, 2002.

[MT00]   N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Soft-
               ware Architecture Description Languages. *IEEE Transactions on Software Engineer-
               ing*, 26(1):70–93, January 2000.

[MTR05]   T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts
               Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science*,
               127(3):113–128, April 2005.

[NCM04]   N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheri-
               tance. *ACM SIGPLAN Notices*, 39(10):99–115, 2004.

[NEHvdH05]   E. C. Nistor, J. R. Erenkrantz, S. A. Hendrickson, and A. van der Hoek. ArchEvol:
               Versioning Architectural-Implementation Relationships. In *SCM '05: Proceedings
               of the 12th International Workshop on Software Configuration Management*, pages
               99–111, New York, NY, USA, 2005. ACM.

[Nel03]   T. P. Nelson. *Formal Verification of Projection-Based Software Systems*. PhD thesis,
               University of Waterloo, Ontario, Canada, 2003.

[NK96]   M. J. Ng K., Kramer J. A CASE Tool for Software Architecture Design. *Automated
               Software Engineering*, 3:261–284, 1996.

[NKMD95]   K. Ng, J. Kramer, J. Magee, and N. Dulay. The Software Architect's Assistant-A
               Visual Environment for Distributed Programming. In *HICSS '95: Proceedings of
               the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 254,
               Washington, DC, USA, 1995. IEEE Computer Society.

[Obj09a]   Object Management Group. Catalog of OMG CORBA and IIOP Specifications. http://www.omg.org/technology/documents/corba_spec_catalog.htm, last accessed August 2009.

[Obj09b]   Object Management Group. CORBA Component Model Specification, v4.0. http://www.omg.org/technology/documents/formal/components.htm, last accessed August 2009.

[Obj09c]   Object Management Group. Model Driven Architecture. http://www.omg.org/mda/, last accessed August 2009.

[Obj09d]   Object Management Group. MOF 2.0 / XMI Mapping Specification, v2.1.1. http://www.omg.org/docs/formal/07-12-01.pdf, last accessed August 2009.

[Obj09e]   Object   Management   Group.   UML   2.0   Specification. http://www.omg.org/spec/UML/2.0/, last accessed August 2009.

[Obj09f]   Object Technology International Inc. Eclipse Platform Technical Overview. http://www.eclipse.org/whitepapers/eclipse-overview.pdf, last access August 2009.

[Obj09g]   Object Technology International Inc. Eclipse Platform. http://www.eclipse.org, last accessed August 2009.

[Obj09h]   Object   Technology   International   Inc.   Eclipse   Plugin   Central. http://www.eclipseplugincentral.com/, last accessed August 2009.

[Ody09]   Odysseus Software GmbH. STAN: Structural Analysis for Java. http://stan4j.com/, last accessed August 2009.

[OL06]   I. Oliver and V. Luukala. On UML's Composite Structure Diagram. In *Fifth Workshop on System Analysis and Modelling*, Kaiserslautern, Germany, 2006.

[OMT98]   P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *ICSE '98: Proceedings of the 20th International Conference on Software Engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[O'N98]   J. O'Neill. *JavaBeans from the Ground Up*. Osborne/McGraw-Hill, 1998.

[O'S09]   B. O'Sullivan. Mercurial: The Definitive Guide. http://hgbook.red-bean.com/read/, last accessed August 2009.

[OSG03]   OSGi Alliance. *OSGi Service Platform*. IOS Press, 2003.

[OSG09]   OSGi   Alliance.   About   the   OSGi   Service   Platform. http://www.osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf,   last accessed August 2009.

[OT01]   H. Ossher and P. Tarr. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, 2001.

[Ous98]   J. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998.

[Par78]   D. L. Parnas. Designing Software for Ease of Extension and Contraction. In *ICSE '78: Proceedings of the 3rd International Conference on Software Engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[PV02]   F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[Rat09]   Rational Corporation, IBM. Rational Rose Technical Developer Website. http://www.ibm.com/developerworks/rational/products/rosetechnicaldeveloper/, last accessed August 2009.

[Ree09]   T. Reenskaug. The Original MVC Reports (1979). http://heim.ifi.uio.no/trygver/2007/MVC_Originals.pdf, last accessed August 2009.

[RJB04]   J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[Rob07]   D. Robinson. *Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers*. Morgan Kaufmann, 2007.

[Rou05]   D. Roundy. Darcs: Distributed Version Management in Haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 1–4, New York, NY, USA, 2005. ACM.

[RRS⁺05]   S. Robert, A. Radermacher, V. Seignole, S. Gérard, V. Watine, and F. Terrier. The CORBA Connector Model. In *SEM '05: Proceedings of the Fifth International Workshop on Software Engineering and Middleware*, pages 76–82, New York, NY, USA, 2005. ACM Press.

[Rum96]   J. Rumbaugh. *OMT Insights: Perspectives on Modeling from the Journal of Object-Oriented Programming*. Cambridge University Press, New York, NY, USA, 1996.

[RVDHMRM04]   R. Roshandel, A. Van Der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae—A System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering*, 13(2):240–276, 2004.

[SB99]   M. Svahnberg and J. Bosch. Characterizing Evolution in Product Line Architectures. In *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*, 1999.

[SE04]   C. S. S. Eisenbach, D. Kayhan. Keeping Control of Reusable Components. In *2nd International Working Conference on Component Deployment*, pages 144 – 158. e-science Institute, Springer-Verlag, 2004. Edinburgh, Scotland, 2004.

[Sei96]   C. Seiwald. Inter-File Branching - A Practical Method for Representing Variants. In *ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management*, pages 67–75, London, UK, 1996. Springer-Verlag.

[Sel03]   B. Selic. Tutorial D: An Overview of UML 2.0. UML 2003, 2003.

[SGW94a]   B. Selic, G. Gullekson, and P. Ward. Inheritance. In *Real-Time Object-Oriented Modeling*, number 9, pages 255–285. Wiley, 1st edition, 1994.

[SGW94b]   B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[SN99]   J.-G. Schneider and O. Nierstrasz. Components, Scripts and Glue. In *Software Architectures: Advances and Applications*, pages 13–25. Springer, 1999.

[Spi92]   J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1992.

[Stu05]   A. Stuckenholz. Component Evolution and Versioning State of the Art. *SIGSOFT Software Engineering Notes*, 30(1):7–, 2005.

[SUN09a]   SUN Developer Network. Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/, last accessed August 2009.

[SUN09b]   SUN Developer Network. Java SE Desktop Technologies - JavaBeans. http://java.sun.com/products/javabeans/, last accessed August 2009.

[SW98]   A. Schürr and A. J. Winter. Formal Definition and Refinement of UML's Module/Package Concept. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 211–215, London, UK, 1998. Springer-Verlag.

[Szy02]   C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Szy06]   C. Szyperski. Independently Extensible Systems – Software Engineering Potential and Challenges. In *In Proceedings of the 19th Australasian Computer Science Conference*, Melbourne, Australia, 2006.

[Tem09]   D. Templin. Escape DLL Hell: Simplify App Deployment with Click-Once and Registration-Free COM. http://msdn2.microsoft.com/en-us/magazine/cc188708.aspx, last accessed August 2009.

[TMA+95]   R. Taylor, N. Medvidovic, M. Anderson, E. Whithead Jr., and J. Robbins. A Component- and Message-based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304, Seattle, Washington, United States, 1995. ACM Press.

[TMD10]   R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley & Sons, 2010.

[TOS02]   P. Tarr, H. Ossher, and S. M. Sutton, Jr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 689–690, New York, NY, USA, 2002. ACM.

[TVJ⁺01]   E. Truyen, B. Vanhaute, B. Jorgensen, W. Joosen, and P. Verbaeton. Dynamic and Selective Combination of Extensions in Component-Based Applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 233–242, Washington, DC, USA, 2001. IEEE Computer Society.

[vdHMRRM01] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. Taming Architectural Evolution. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–10, Vienna, Austria, 2001. ACM Press.

[vGB01]   J. van Gurp and J. Bosch. Design, Implementation and Evolution of Object Oriented Frameworks: Concepts and Guidelines. *Software - Practice And Experience In the Second International Workshop on Component-Oriented Programming*, 31(3):277–300, 2001.

[VL89]   J. M. Vlissides and M. A. Linton. Unidraw: a Framework for Building Domain-Specific Graphical Editors. In *UIST '89: Proceedings of the 2nd annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 158–167, New York, NY, USA, 1989. ACM.

[vO00]   R. van Ommering. Mechanisms for Handling Diversity in a Product Population. In *ISAW-4: The Fourth International Software Architecture Workshop*, 2000.

[vO02]   R. van Ommering. Building Product Populations with Software Components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, New York, NY, USA, 2002. ACM Press.

[vO03]   R. van Ommering. Horizontal Communication: A Style to Compose Control Software. *Software - Practice And Experience*, 33(12):1117–1150, 2003.

[Vol99]   M. Volter. Pluggable Component - A Pattern for Interactive System Configuration. In *The Fourth European Conference of Pattern Languages of Programming and Computing, EuroPLoP '99*, Bad Irsee, Germany, 1999.

[Whe09]   D. A. Wheeler. SLOCCount: A set of tools for counting physical source lines of code. http://www.dwheeler.com/sloccount/, last accessed August 2009.

[Wor09]   Wordpress. Wordpress Plugins. http://wordpress.org/extend/plugins/, last accessed August 2009.

[Yeg09]   S. Yegge. When Poymorphism Fails. http://steve.yegge.googlepages.com/when-polymorphism-fails, last accessed August 2009.

[ZA06]   D. J. Zito A, Diskin Z. Package Merge in UML 2: Practice vs. Theory? *Model Driven Engineering Languages and Systems*, 9:185–199, 2006.

[ZO04]   M. Zenger and M. Odersky. Independently Extensible Solutions to the Expression Problem. Technical report, Ecole Polytehnique Federale de Lausanne, 2004.

Appendix A

The Backbone Specification in Detail

This appendix augments the outline of the formal specification presented in chapter 4, and explains some of the logic from appendices B and C in more detail.

We start by describing the logic contained within the `Deltas` signature. The formal model is then used to generate an extension conflict and its subsequent resolution using a further extension, and the result is imported into Backbone for visualisation.

The port type inference logic is also explained, and some interesting test cases are examined.

Finally, the set of rules which govern the correct structure of an architecture are presented. A subset of these are known as well-formedness rules: they are structural validations that can be violated when combining extensions. Any issues must be addressed by a further extension in order to produce a structurally correct architecture.

A.1   The Deltas Logic

The `Deltas` signature is specified by 160 lines of Alloy code in the file `bb_deltas.als`, in appendix B. This section explains this in more detail.

As described in section 4.2.4, an element is composed of constituents. A constituent can be added, deleted or replaced in an element by using a delta alteration. Each `Deltas` instance holds a number of these alterations for a single constituent type.

Alterations in a `Deltas` instance are held in the `addObjects`, `deleteObjects` and `replaceObjects` fields as shown in the listing below. These inputs correspond to the alterations that a developer makes when adjusting the inherited structure of a component or interface in Evolve.

```
newIDs:              set ID,
addObjects:          newIDs one -> one addedObjects,
deleteObjects:       set ID,
```

```
replaceObjects:        ID one -> lone replacedObjects,
```

<div align="center">Listing A.1: base_deltas.als, lines 13-17</div>

The `newIDs` field holds any new UUIDs which have been allocated for added or replacing constituents. These identifiers are forced to be globally unique. Further, any new objects from adds or replaces are owned by a single `Delta` instance and cannot be shared.

```
all o: Object |
  one newObjects.o
all n: ID |
  one newIDs.n
```

<div align="center">Listing A.2: base_deltas.als, lines 46-49</div>

A `Deltas` instance cannot delete any object that it is simultaneously replacing. A constituent that is being deleted or replaced must be present lower down in the resemblance graph, when looking from the owning element's home perspective.

```
let
  deleteIDs = this.deleteObjects,
  replaceIDs = dom[this.replaceObjects]
{
  no deleteIDs & replaceIDs
  deleteIDs + replaceIDs in dom[this.originalOldObjects_e[owner]]
}
```

<div align="center">Listing A.3: base_deltas.als, lines 74-85</div>

A.1.1   Merging and Applying Changes

The core logic for `Deltas` is contained in two predicates. The aim of these predicates is to apply the delta alterations of an element, from a given stratum perspective, in order to construct the fully expanded form of the element. For each perspective, we then have a mapping of UUID (`ID`) to constituent (`Object`) for the given stratum. This is held in the `objects_e` field, and represents the final output of the delta application logic.

```
objects_e: Stratum -> ID -> Object,
```

<div align="center">Listing A.4: base_deltas.als, lines 19-19</div>

The two predicates are described below.

Applying Changes for Resemblance: **mergeAndApplyChangesForResemblance**

This predicate uses the expanded resemblance graph, from a given perspective, to apply the deltas and form the expanded constituents. Components that replace others are treated as part of this graph, as

per B' in figure 4.5.

The parameters to the predicate are listed below. The s parameter is the stratum perspective, c is the element we are expanding and iResembleDeltas_e contains the deltas from the elements under this one in the expanded element graph. For the last parameter, if we were expanding A in figure 4.5, the value would hold the deltas for both extension1::B' and extension2::B'.

```
pred Deltas::mergeAndApplyChangesForResemblance(
  s: Stratum,
  c: Element,
  iResembleDeltas_e: set Deltas)
```

Listing A.5: base_deltas.als, lines 104-108

We cannot use the objects_e field to hold the results of this predicate, as that field is used when we copy the expanded results from any replacements back into the base element. As such, the following fields are used to hold these interim results.

```
originalObjects_e:            Stratum -> ID -> Object,
originalDeletedObjects_e:     Stratum -> ID,
originalReplacedObjects_e:    Stratum -> ID -> Object
```

Listing A.6: base_deltas.als, lines 33-35

The originalDeletedObjects_e field holds all of the constituents that have been deleted for the expanded graph so far. It is necessary to keep track of all deleted constituents, and reapply them to handle the situation where the resemblance graph merges a delete and non-delete of the same constituent from parallel evolutions. In this case, we do not wish to add back the constituent. Note that a parallel application of delete and replace will always favour replace.

The originalObjects_e field holds the expanded constituents for the stratum perspective. It coalesces the original constituents from elements it resembles in the expanded graph, removes any deleted constituents and then applies the adds and replaces that it holds locally.

```
this.originalObjects_e[s] =
  (((iResembleDeltas_e.originalObjects_e[s] - this.originalDeletedObjects_e[s]->
      Object)
    ++ this.originalReplacedObjects_e[s]) + this.addObjects)
      ++ this.replaceObjects
```

Listing A.7: base_deltas.als, lines 125-128

Applying Changes for Replacement: **mergeAndApplyChangesForElementReplacement**

This predicate copies over the expanded constituents from replacements into the original base component. This removes the need to redirect existing references from the base to an evolution or replacement, as explained at the end of section 4.2.4.

The key expression is listed below. This first part of the expression copies over the constituents from any replacements, for the given perspective. The second part applies the deletes and replaces to again ensure that parallel evolutions are handled correctly.

```
this.objects_e[s] =
  (iResembleDeltas_e.originalObjects_e[s] - this.deletedObjects_e[s]->Object)
    ++ this.replacedObjects_e[s]
```

Listing A.8: base_deltas.als, lines 157-159

A.1.2   Structural Conflict at the Deltas Level

Structural conflict at the `Deltas` level occurs when there is more than one constituent for a given `ID`, after the alterations have been fully applied. Freedom from delta conflict is checked by the following predicate. This ensures that there is only at most one constituent per `ID`.

```
pred Deltas::oneObjectPerID(s: Stratum)
{
  let objects = this.objects_e[s] |
    function[objects, dom[objects]]
}
```

Listing A.9: base_deltas.als, lines 53-57

Delta conflict occurs when independently developed extensions, that both replace the same constituent of an element, are combined. This is not explicitly checked in the base level, but results in a series of checks for each constituent type in the ADL level.

A.2   Generating Conflict and Resolution from the Specification

We can use the formal specification and the Alloy analyser to generate a witness showing conflict from combining independently developed extensions, and subsequent resolution in a further extension.

This scenario can be generated by the following Alloy expression.

```
run conflictAndResolution for 3 but
  exactly 5 Stratum,
  exactly 6 Element,
  exactly 5 Component,
  exactly 1 Interface,
  exactly 5 bb/full/Ports/Deltas,
  exactly 4 bb/full/Parts/Deltas,
  exactly 4 bb/full/Connectors/Deltas,
  exactly 0 bb/full/Operations/Deltas,
  exactly 4 Part,
```

```
  5 LinkEnd
// generates a conflict & resolves it: types of replaced parts will be the same.
// takes 9737s on core2duo machine
pred conflictAndResolution
{
  no parent
  no links
  some isInvalid_e
  some disj s1, s2, s3, s4, s5: Stratum
  {
    s1.dependsOn = s2
    s2.dependsOn = s3 + s4
    s3.dependsOn = s5
    s4.dependsOn = s5
    no s2.ownedElements
    Model::errorsOnlyAllowedIn[s2]
  }
}
```

Listing A.10: unittests_conflict.als, lines 18-45

The use of the `exactly` keyword limits the state space for checking, allowing a witness to be found more quickly at the cost of sometimes missing a solution if the state space is too small. The "some isInvalid_e" statement will force a conflict. As each stratum has to be well-formed when it is defined, this can only occur by combining two stratum that are mutually independent. Further, to get conflict, the extensions must replace or evolve an element in a common base strata.

Backbone can import witnesses from the formal specification. An example witness generated by the above is shown in figure A.1.

The source of the conflict is that `Component1` and `Component2` both replace the same part (`PartID0`) in independent evolutions of `Component4`. The conflict is resolved by a further evolution in `Stratum4` which replaces `PartID0` again.

As the state space is restricted to five components, there cannot be enough instances to allow each evolution to replace the part with one of a different type. The conflict occurs even though the replacing parts are all of the same type, as they are logically different instances of that type.

Another interesting witness generated in the same way featured a base leaf component that was separately evolved into a composite component by one of the independent extension strata.

A.3   Port Type Inference

Port type inference is used to automatically determine the interfaces provided and required by ports of a composite component. This facility is important for extensibility in a hierarchical component model,
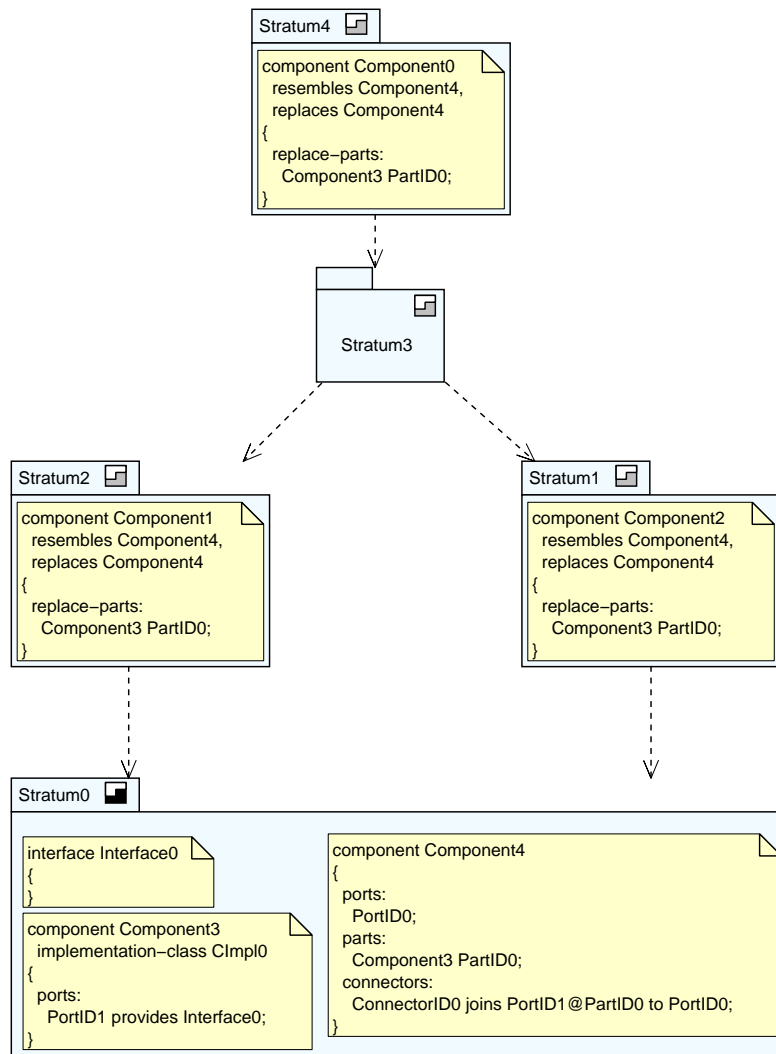
Figure A.1: A witness showing conflict and subsequent resolution

as it removes the need to respecify all the ports of composites when a leaf is evolved and its ports are altered. Without this, a single evolution deep in the base of the compositional hierarchy could require a cascade of many other evolutions, all the way up to the top of the compositional tree.

A.3.1   Inference Logic

The aim of the inference logic is to describe how interface type constraints propagate between the ports of a component, reflecting the connectors and port links in the compositional hierarchy.

In essence the inference logic states that if more (i.e. a subinterface) is provided to a required interface on a component port then, depending on the connectors and ports links, more may also be provided by another port on the component.

Consider figure A.2 as an example. Component `X` has a port `rt` which provides interface `A`. This port also has a port link to port `lt` which requires interface `A`. Port type information can propagate along
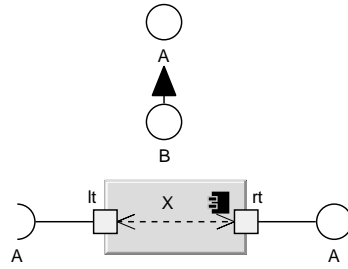
Figure A.2: A leaf component with a port link
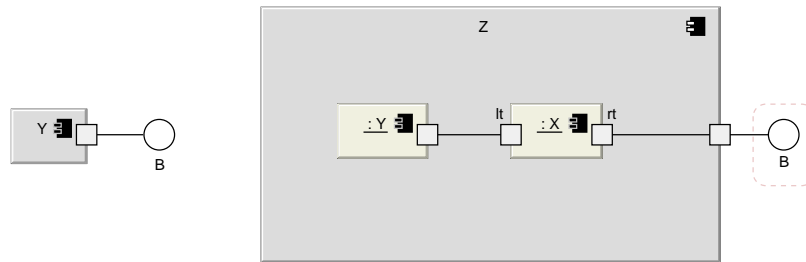
Figure A.3: Type propagation of a provided subinterface

this link to `rt` if more is provided to `lt`.

We can see this propagation in figure A.3, where interface `B` (a subinterface of `A`) has been provided to the `lt` port of the part of type `X`. The type information has propagated through to `rt`, which also now provides `B`.

To infer the provided interfaces of a port, the inference logic traverses to any terminal ports of parts, or ports of the component which are reachable from that port by following connectors and port links. A terminal port is a leaf component port that does have a port link. In our previous example, the port of `Y` was terminal, but not ports `lt` or `rt`. The provided interface for port `Z` was then determined by traversing the connectors and port links back to the provided port of the `Y` part. The logic ensures that the interfaces along the traversal routes are compatible.

To infer the required interfaces of a port, the logic also traverses to any ports of parts that can be reached from that port by following connectors or port links. Note that unlike for provided interfaces, this does not consider other ports of the component.

Alloy provides the reflexive-transitive operator ($\star$), and this is used to succinctly form the full set of navigation paths between ports and parts in a component. Some of this logic is shown below.

```
fromPortToPart = portToPart.*(partInternal.partToPart),
fromPartToPort = ~fromPortToPart,
fromPartToPart = partToPart.*(partInternal.partToPart),
```

Listing A.11: bb_port_inference.als, lines 62-64

A.3.2   Links and Connectors in the Compositional Hierarchy

In theory, the inference logic involves traversing all possible links and connectors in the entire component hierarchy. To simplify this, each component instead summarises the relationships between its ports and stores this in the `inferredLinks` field of the `Component` signature. This is analogous to the port link information provided by a leaf. This field can then be used by the next level up in the compositional hierarchy as an alternative to navigating further down into the structures. The field needs to be determined for each component from each relevant stratum perspective, as it can change as extensions are applied.

The predicate for determining the inferred links for a composite is `setupLinks`, which takes the following parameters.

---
pred setupCompositeLinks(s: Stratum, c: Component)

<div align="center">Listing A.12: bb_port_inference.als, lines 42-42</div>

This contains logic to traverse the connectors of the composite to infer the port interfaces, using the `inferredLinks` of each part type. The `inferredLinks` field can also be cached in the implementation, avoiding the need to traverse deeply into the same part types many times when determining port interfaces in a large system

A.3.3   Two Interesting Test Cases

We now present two of the more interesting test cases used to evaluate the inference logic.

The first involves component `Case1`, shown in figure A.4, where a port and internal part separately constrain the propagation of a subinterface. Interface `C` resembles interface `B` which resembles interface `A`. The `BProv` part provides interface `B`. The component `AThrough` requires `A` via the `lt` port and provides `A` via the `rt` port. `AThrough` has a port link between the two ports.

In this case, the inference logic has determined that port `right` of `Case1` can only provide interface `A` despite the `BProv` part providing `B`. This is because when the traversals from the `right` port are considered we end up navigating to the port of `BProv` (provides `B`) and the `left` port (requires `A`). We can then only propagate the lowest common superinterface[1] of `B` and `A`, which is `A`.

If a part of type `Case1` was to configured so that `left` was connected to something providing interface `B`, then this would propagate through to `right`. However, if something provided interface `C` to `left`, then only interface `B` would propagate through as `BProv`'s provided interface constrains the end result to be the lowest superinterface of `B` and `C`, which is `B`.

The second test case examines how required interfaces affect type propagation. Consider the composite component `Case2` in figure A.5, where the two parts require interfaces which are not subtypes. In this example, interfaces `X` and `Y` both resemble `Z`, and interface `Q` resembles both `X` and `Y`. Component

---

[1]Lowest and highest refer to positions in an inheritance graph where supertypes are placed above subtypes.
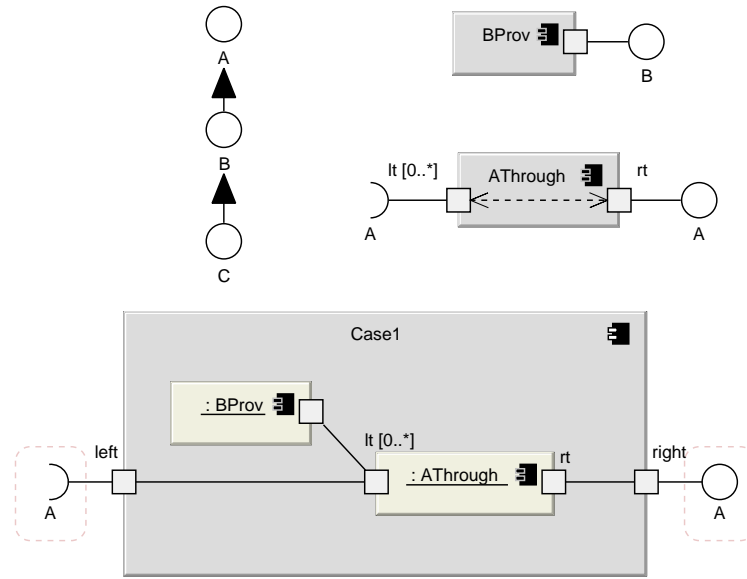
Figure A.4: Internal provisions can constrain port type propagation

ZThrough requires Z through the lt port and provides Z through the rt port. There is a port link between the two ports of ZThrough.
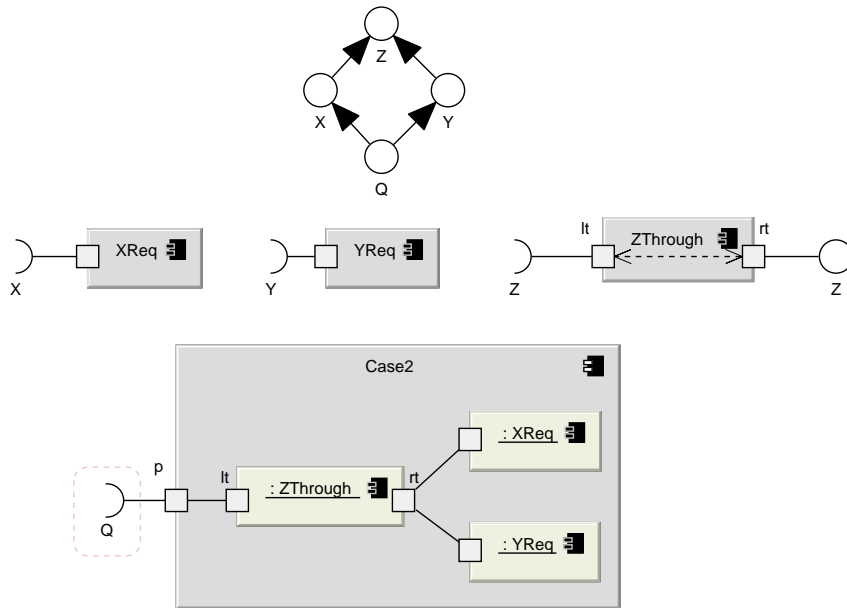
Figure A.5: Internal interface requirements can result in the need for a common subinterface

As we can see, the inference logic has determined that port p of Case2 requires interface Q. According to the traversal logic described previously, by navigating from the port of Case2 we reach the ports of XReq and YReq. We can only propagate an interface which satisfies both of these required interfaces, which is Q. If there was no interface that resembled both X and Y, then the inference logic would determine that the port was in error. If multiple interfaces resembled X and Y then an error would also result, unless there was a single highest superinterface of all candidates that also resembled X and Y.

A.4   Structural Rules of the Specification

The formal specification contains a number of structural rules. These verify that an architecture is structurally correct.

As previously discussed, it is possible for an element to be in error from a stratum perspective even if it has no errors from its home perspective. This can occur when two independently developed (and correct) strata that modify a common base are combined. To allow these errors to be represented in the specification (rather than just outlawed from generated witnesses), we have devised a set of well-formedness rules to model the possible errors that can occur. These are a subset of the structural rules.

Well-formedness rules for an element must be checked from every relevant perspective, unlike the other structural rules that can just be checked for an element in its home stratum. A relevant perspective is the element's home stratum or any strata that can transitively reach this through dependencies. A component, for instance, can change from a leaf in its home stratum to a composite in an extension. The well-formedness rules will have to be run on this component for both perspectives; in the first case treating it as a leaf and in the second case treating it as a composite.

The rest of this section describes the rules, and also references the relevant Alloy listings. Note that the well-formedness rules have a WF_ prefix.

A.4.1   Stratum Rules

- STRATUM_ACYCLIC
  (`base_facts.als`, line 30)
  No strata dependency cycles are permitted. As a stratum automatically depends on nested strata, a stratum cannot depend on its parent without breaking this rule.

- STRATUM_ELEMENT_REPLACEMENT
  (`base_facts.als`, line 45)
  A stratum can only contain at most one replacement for any given element. The replaced element cannot be owned by the same stratum as the replacement.

A.4.2   Element Rules

- ELEMENT_HOME
  (`base_structure.als`, line 75)
  Each element has a single home stratum, which owns it.

- ELEMENT_VISIBILITY
  (`base_facts.als`, line 36)
  An element only has visibility of definitions in other strata if its home stratum can see these

strata via its dependency relationships. For rules on what a stratum can see, refer to section 4.2.1. This visibility rule constrains what an element can resemble or replace, the part types that can be used, the attribute types that are visible and the interfaces that a port can refer to.

- ELEMENT_OK_AT_HOME
  (`bb.als`, line 250 for components)
  An element must conform to all the structural rules (including the well-formedness rules) from its home perspective.

- ELEMENT_RESEMBLANCE
  (`bb_structure.als`, line 37 for components)
  An element can resemble one or more elements of the same type. For instance, a component can resemble other components, but cannot resemble interfaces.

- ELEMENT_REPLACEMENT
  (`bb_structure.als`, line 37 for components)
  An element can replace at most one other element of the same type. The element being replaced must not be in the same stratum as the element doing the replacing.

- ELEMENT_REPLACEMENT_NOT_REFERENCED
  (`bb.als`, line 128 for components)
  An element cannot refer directly to a replacement or an evolution. As described in section 4.2.4, all references must be made to the original element that was replaced rather than its replacement.

- WF_ELEMENT_EXPANDED_RESEMBLANCE_ACYCLIC
  (`base_facts.als`, line 108)
  The expanded resemblance graph must always be acyclic. When this rule is checked from the home perspective of an element, this also verifies that the normal resemblance graph is acyclic.

A.4.3 Delta Rules

- DELTA_DELETE
  (`base_deltas.als`, line 82)
  A delta deletion must refer to an inherited constituent, when viewed from the perspective of the owning element's home stratum.

- DELTA_REPLACE
  (`base_deltas.als`, line 82)
  In the same fashion, a delta replacement must also refer to an inherited constituent.

A.4.4 Interface Rules

- WF_INTERFACE_ONE_IMPLEMENTATION
  (`bb_well_formed.als`, line 14)
  An interface must have a single implementation interface name constituent.

- WF_INTERFACE_OPERATION_PER_UUID
  (`bb_well_formed.als`, line 10)
  An interface must have at most one operation constituent per UUID.

A.4.5 Primitive Type Rules

- WF_PRIMITIVE_IMPLEMENTATION
  (`bb_well_formed.als`, line 21)
  A primitive type must have a single implementation class name constituent.

A.4.6 Component Rules

- WF_COMPONENT_NO_SELF_COMPOSITION
  (`bb_well_formed.als`, line 37)
  A fully expanded component cannot contain an instance of itself. This rule covers possibly inheriting an instance via resemblance or when replacements are applied, or having parts which indirectly contain an instance.

- WF_COMPONENT_PORTS
  (`bb_well_formed.als`, line 40)
  A component must have at least one port.

- WF_COMPONENT_LINKS_OK
  (`bb_well_formed.als`, line 54)
  A composite component cannot have port link constituents. There must be no duplication between port links. For instance, two port links cannot connect between the same ports.

- WF_COMPONENT_LEAF_IMPLEMENTATION
  (`bb_well_formed.als`, line 71)
  A leaf must must have a single implementation class name constituent. Composite components must have no implementation class name constituents.

- WF_COMPONENT_PORT_PER_UUID
  (`bb_well_formed.als`, line 43)
  A component must have at most one port constituent per UUID.

- WF_COMPONENT_PART_PER_UUID
  (`bb_well_formed.als`, line 57)
  A component must have at most one part constituent per UUID.

- WF_COMPONENT_ATTRIBUTE_PER_UUID
  (`bb_well_formed.als`, line 45)
  A component must have at most one attribute constituent per UUID.

- WF_COMPONENT_CONNECTOR_PER_UUID
  (`bb_well_formed.als`, line 59)
  A component must have at most one connector constituent per UUID.

- WF_COMPONENT_PORTLINK_PER_UUID
  (`bb_well_formed.als`, line 78)
  A component must have at most one port link constituent per UUID.

A.4.7 Part Rules

- PART_TYPE_VISIBLE
  (`bb.als`, line 146)
  A part's type must be visible from the home stratum of the element that owns it.

- WF_PART_TYPE
  (`bb_structure.als`, line 109)
  A part must have a type.

- WF_PART_SLOT_LITERAL
  (`bb_well_formed.als`, line 124)
  If a part slot contains a literal value, it must be of the correct type for the attribute. This is a well-formedness rule because although a slot cannot change with perspective, the attribute that it refers to can.

- WF_PART_SLOT_ALIAS_COPY
  (`bb_well_formed.als`, line 128)
  If a part slot is aliased or copied, then the attribute must exist in the expanded definition of the enclosing component.

- WF_PART_SLOT_DEFAULT
  (`bb_well_formed.als`, line 137)
  A slot must exist for each attribute of the part's type that has no default value. The slot must have a literal value, or alternatively copy or alias an attribute.

- WF_PART_NO_ISLANDS
  (`bb_well_formed.als`, line 106)
  By following connectors between the ports of a component and its parts, we must be able to reach all parts of a component. It is not allowable to have a "part island" which is not anchored back to the component eventually via connectors.

- WF_COMPONENT_LEAF_PORTS
  (`bb_port_inference.als`, line 32)
  If a component is a leaf, the interfaces that the ports provide and require must be explicitly specified. Leaf component ports do not use type inference to determine their interfaces. Composite

component ports may also specify their interfaces explicitly, but this is only used or checked from the home perspective. Type inference is used in all other perspectives.

A.4.8 Connector and Port Link Rules

- WF_CONNECTOR_NO_PORT_TO_PORT
  (`bb_port_inference.als`, line 170)
  Connectors cannot join one port of a composite to another. They must connect a port to a part (via the part's port) or a part to a part.

- WF_CONNECTOR_OPTIONAL
  (`bb_well_formed.als`, line 184)
  If one end of a connector binds to an optional index of a component port, then the other end must also bind to an optional index of a part's port.

- WF_CONNECTOR_SAME
  (`bb_well_formed.als`, line 190)
  A connector between two parts must have both ends binding to the same type of port index; either both optional or both mandatory.

- WF_CONNECTOR_INDEX
  (`bb_well_formed.als`, line 217)
  Each index of a connector end must be within the range of the port multiplicity.

- WF_CONNECTOR_ONE_TO_ONE
  (`bb_port_inference.als`, line 109 and other locations)
  There must be an unambiguous, one-to-one mapping between the provided and required interfaces of one connector end, and the required and provided interfaces of the other end. This is described in more detail in section 4.3.2.

- WF_LINK_COMPLEMENTARY
  (`bb_port_inference.als`, line 24)
  Two ports that are joined by a port link must have complementary (i.e. reversed) provided and required interfaces.

- WF_CONNECTOR_PROVIDES_ENOUGH
  (`bb_port_inference.als`, line 148)
  Each end of a connector must provide the same interfaces (or subtypes) to satisfy the required interfaces of the ports on the other end.

A.4.9 Port Rules

- PORT_MULTIPLICITY
  (`bb_structure.als`, line 149)

The upper bound of the port multiplicity must be greater than or equal to the lower bound. The lower bound must be greater than or equal to 0.

- PORT_INTERFACE_VISIBILITY
  (`bb.als`, line 148)
  If the interfaces that are required and provided by a port are explicitly declared, these must be visible to the home stratum of the component that adds the port.

- WF_PORT_SOME_INTERFACES
  (`bb_port_inference.als`, line 191)
  A port must provide or require some interfaces, either by explicitly declaring these or alternatively via port type inference.

A.4.10 Attribute Rules

- ATTRIBUTE_TYPE_VISIBILITY
  (`bb.als`, line 150)
  The primitive type of the attribute must be visible to the home stratum of the component that adds the attribute.

- WF_ATTRIBUTE_TYPE
  (`bb_structure.als`, line 193)
  An attribute must have a type.

- WF_ATTRIBUTE_DEFAULT
  (`bb_structure.als`, line 198)
  If an attribute specifies a default value, then the literal or attribute reference must be of the correct type.

Appendix B

The Backbone Formal Specification: Stratum, Element and Deltas

B.1 Stratum and Element Signatures: **base_structure.als**

Listing B.1: base_structure.als

```
1   module base_structure
2
3   open util/boolean as boolean
4   open util/relation as relation
5
6
7   one sig Model
8   {
9     -- normally this should be set to none
10    errorsAllowed: set Stratum,
11    providesIsOptional: Bool
12  }
13
14  sig Stratum
15  {
16    parent: lone Stratum,
17
18    -- external strata that this depends directly on
19    dependsOn: set Stratum,
20    -- any child packages that are explicitly depended on
21    dependsOnNested: set Stratum,
22    -- nested stratum are any with this as the direct parent
23    nestedStrata: set Stratum,
24
25    -- does this export stratum it depends on?
26    isRelaxed: Bool,
27    ownedElements: set Element,
28
29    -- derived state -- export strata includes this and canSee
30    exportsStrata: set Stratum,
```

```
31    canSee: set Stratum,
32
33    -- simple is all that we directly depend on
34    -- taking away what any children depend on
35    simpleDependsOn: set Stratum,
36
37    -- a single top exists which binds directly
38    -- any independent stratum
39    isTop: Bool,
40    -- this is every stratum that can be seen from here down
41    transitive: set Stratum,
42    -- elements that replace others
43    replacing: set Element,
44    canSeePlusMe: set Stratum,
45    transitivePlusMe: set Stratum,
46
47    -- components that are new definitions
48    defining: set Element
49  }
50  {
51    defining = ownedElements - replacing
52    canSeePlusMe = canSee + this
53    transitivePlusMe = transitive + this
54    nestedStrata = {n: Stratum | n.@parent = this}
55  }
56
57  ----------------------------------------------------
58  -- handle the basics of resemblance and replacement
59  ----------------------------------------------------
60
61  abstract sig Element
62  {
63    home: Stratum,
64    replaces: lone Element,
65    resembles: set Element,
66
67    -- for a given stratum, a component resembles other components in a given
           stratum view
68    resembles_e: Element -> Stratum,
69    -- does this act as a non-primed for a particular stratum
70    actsAs_e: Element -> Stratum,
71    -- is this element valid for a given stratum?
72    isInvalid_e: set Stratum
73  }
74  {
75    -- rule: ELEMENT_HOME -- element is owned by a single stratum
76    home = ownedElements.this
77  }
```

B.2   Stratum and Element Facts: `base_facts.als`

<div align="center">Listing B.2: base_facts.als</div>

```
1   module facts
2
3   open base_structure
4
5
6   fact StratumFacts
7   {
8     one isTop.True
9     all s: Stratum |
10    -- a stratum depends on any external it explicitly declares, plus any nested
          strata
11    let fullDependsOn = dependsOn + nestedStrata + ^parent.dependsOn {
12      -- dependsOn cannot be nested packages
13      all d : s.dependsOn | s not in d.*parent
14
15      -- dependsOnNested must be nested packages.  these can be deeply nested also
          .
16      all n : s.dependsOnNested | s in n.^parent
17
18      -- for relaxed, export what it depends on and their exports
19      -- for strict and relaxed, export any direct nested dependencies
20      isTrue[s.isRelaxed] =>
21        s.exportsStrata = s + s.(dependsOn + dependsOnNested).exportsStrata
22      else
23        s.exportsStrata = s + s.dependsOnNested.exportsStrata
24
25      -- used for partial ordering
26      -- contains nothing the children already depend on
27      s.simpleDependsOn = s.fullDependsOn - s.fullDependsOn.transitive
28
29      -- no cycles
30      -- rule: STRATUM_ACYCLIC
31      s not in s.transitive
32
33      -- can only see what others export
34      -- NOTE: if a nested stratum only exports a public package, then this is
35      --       all that the parent can see
36      -- rule: ELEMENT_VISIBILITY -- defined what can be seen.  augmented by other
          checks
37      s.canSee = s.fullDependsOn.exportsStrata
38
39      -- the strata we can see using the dependency graph
40      s.transitive = s.^fullDependsOn
41
42      -- a stratum is called top if no stratum depends on it
43      isTrue[s.isTop] <=> no simpleDependsOn.s
44
45      -- rule: STRATUM_ELEMENT_REPLACEMENT -- have max of one replacement of an
          element per stratum
46      all e: Element |
47        lone s.ownedElements & replaces.e
```

```
48        -- ties up replacing and replaces
49        s.replacing = s.ownedElements & dom[replaces]
50    }
51 }
52
53
54 pred independent[a, b: Stratum] {
55    a not in b + b.transitive and b not in a + a.transitive
56 }
57 pred independentOnCommonBase[a, b: Stratum] {
58    independent[a, b] and some a.transitive & b.transitive
59 }
60
61 fun stratumPerspective[stratum: Stratum]: set Stratum
62 {
63    stratum.*dependsOn
64 }
65
66 ----------------------------------------------------
67 -- handle the basics of resemblance and replacement
68 ----------------------------------------------------
69
70 fact ElementFacts
71 {
72    -- nothing can resemble a replacement -- check to see that the things we
            resemble don't replace also
73    no resembles.replaces
74
75    all
76      e: Element |
77    let
78      owner = e.home,
79      -- strata that can see the component
80      resemblingOwningStratum = e.resembles.home,
81      replacingOwningStratum = e.replaces.home
82    {
83      -- no circularities in resemblance or replacement, and must be visible
84      e not in (e.^resembles + e.replaces)
85      resemblingOwningStratum in owner.canSeePlusMe
86      replacingOwningStratum in owner.canSee
87
88      -- tie up the owning stratum and the elements owned by that stratum
89      e.home = ownedElements.e
90
91      -- we only need to form a definition for stratum that can see us
92      all s: Stratum |
93      let
94        -- who should I resemble
95        -- (taking replacement into account)
96        iResemble = e.resembles_e.s,
97        -- if we resemble what we are replacing,
98        -- look for the original under here
99        topmostOfReplaced = getTopmost[
100           owner.simpleDependsOn,
101           e.replaces & e.resembles],
102        -- look for any other resembled components
103        -- from here down
```

```
104        topmostOfResemblances = getTopmost[
105          s,
106          e.resembles - e.replaces]
107      {
108        -- rule: WF_ELEMENT_EXPANDED_RESEMBLANCE_ACYCLIC
109        -- expanded resemblance graph must also be acyclic
110        e not in e.^(resembles_e.s)
111
112        owner not in s.transitivePlusMe =>
113        {
114          no iResemble
115          no e.actsAs_e.s
116        }
117        else
118        {
119          -- rewrite the resemblance graph to handle replacement
120          iResemble = topmostOfReplaced + topmostOfResemblances
121          -- who do we act as in this stratum?
122          e.actsAs_e.s =
123            { real: Element | no real.replaces and e in getTopmost[s, real] }
124        }
125      }
126    }
127 }
128
129 fun getTopmost(s: set Stratum, e: Element): set Element
130 {
131   let replaced = replaces.e & s.transitivePlusMe.replacing,
132     topmostReplaced = replaced - replaced.resembles_e.s
133       { some topmostReplaced => topmostReplaced else e }
134 }
```

B.3   Deltas Signature: **base_deltas.als**

Listing B.3: base_deltas.als

```
1  module base_deltas[ID, Object]
2
3  open base_structure
4
5  sig Deltas
6  {
7    -- newObjects is any objects added or replaced.  these fields allow new object
           creation to be controlled
8    newObjects:           set Object,
9    addedObjects:         set newObjects,
10   replacedObjects:      set Object,
11
12   -- newIDs are any new IDs added
13   newIDs:               set ID,
14   -- the deltas that are to be applied.  these 3 fields are the input to the
         merge
15   addObjects:           newIDs one -> one addedObjects,
```

```
16    deleteObjects:           set ID,
17    replaceObjects:          ID one -> lone replacedObjects,
18
19    objects_e: Stratum -> ID -> Object,
20    -- the expanded objects for this stratum.  these 2 fields are the output of
         the merge!
21    objects:                 Object -> Stratum,
22    -- old objects is what was what was there before any replacing was done
23    oldObjects_e:         Stratum -> ID -> Object,
24    originalOldObjects_e:    Stratum -> ID -> Object,
25
26    -- working variables to track the expansion of objects, and allow it to happen
          cumulatively
27    -- note: we need to keep track of what has been deleted and replaced to handle
          the cumulative effects
28    -- e.g. delete in one stream, not in the other.
29    -- NOTE: original is taking only resemblance_e into account, non-original is
          the full definition
30    --        taking element replacement into account also
31    deletedObjects_e:            Stratum -> ID,
32    replacedObjects_e:           Stratum -> ID -> Object,
33    originalObjects_e:           Stratum -> ID -> Object,
34    originalDeletedObjects_e:    Stratum -> ID,
35    originalReplacedObjects_e:   Stratum -> ID -> Object
36 }
37 {
38    -- cannot delete and replace
39    no dom[replaceObjects] & deleteObjects
40    replacedObjects = newObjects - addedObjects
41 }
42
43 -- indicate that any new part/ID can only be introduced by one component
44 fact Owned
45 {
46    all o: Object |
47      one newObjects.o
48    all n: ID |
49      one newIDs.n
50 }
51
52
53 pred Deltas::oneObjectPerID(s: Stratum)
54 {
55    let objects = this.objects_e[s] |
56      function[objects, dom[objects]]
57 }
58
59 pred Deltas::nothing(s: Stratum)
60 {
61    no this.objects.s
62    no this.objects_e[s]
63    no this.oldObjects_e[s]
64    no this.deletedObjects_e[s]
65    no this.replacedObjects_e[s]
66    no this.originalObjects_e[s]
67    no this.originalDeletedObjects_e[s]
68    no this.originalReplacedObjects_e[s]
```

```
69  }
70
71  -- ensure that deletes and replaces makes sense from the perspective of the
        original stratum
72  pred Deltas::deltasIsWellFormed(owner: Stratum)
73  {
74    -- no overlap between deleted and replaced IDs
75    let
76      deleteIDs = this.deleteObjects,
77      replaceIDs = dom[this.replaceObjects]
78    {
79      -- no overlap between deleted and replaced
80      no deleteIDs & replaceIDs
81      -- anything we delete or replace must be there already
82      -- rule: DELTA_DELETE -- deleted object must be present
83      -- rule: DELTA_REPLACE -- replaced object must be present
84      deleteIDs + replaceIDs in dom[this.originalOldObjects_e[owner]]
85    }
86  }
87
88  -- ensures that this delta removes everything
89  pred Deltas::cleanSlate(owner: Stratum)
90  {
91    this.deleteObjects = dom[this.oldObjects_e[owner]]
92  }
93
94  -- ensures that we only have adds, no deletes or replaces
95  pred Deltas::onlyAdds(owner: Stratum)
96  {
97    no this.deleteObjects
98    no this.replaceObjects
99  }
100
101
102 -- the predicate to merge any underlying resembled entities and apply current
        changes
103 -- this is driven off the newly computed resemblance graph for each component in
        each stratum
104 pred Deltas::mergeAndApplyChangesForResemblance(
105   s: Stratum,
106   c: Element,
107   -- who should I resemble, taking element replacement into account
108   iResembleDeltas_e: set Deltas)
109 {
110   -- handle add, delete etc as if we are only taking resemblance into account
111   -- nothing will ever resemble itself
112   this.originalOldObjects_e[s] =
113     (iResembleDeltas_e.originalObjects_e[s]
114       - iResembleDeltas_e.originalDeletedObjects_e[s]->Object)
115         ++ iResembleDeltas_e.originalReplacedObjects_e[s]
116
117   this.originalDeletedObjects_e[s] =
118     iResembleDeltas_e.originalDeletedObjects_e[s]
119       - dom[iResembleDeltas_e.originalReplacedObjects_e[s]] + this.deleteObjects
120
121   this.originalReplacedObjects_e[s] =
122     (iResembleDeltas_e.originalReplacedObjects_e[s] - this.deleteObjects->Object
```

```
           )
123          ++ this.replaceObjects
124
125     this.originalObjects_e[s] =
126        (((iResembleDeltas_e.originalObjects_e[s] - this.originalDeletedObjects_e[s
                ]->Object)
127           ++ this.originalReplacedObjects_e[s]) + this.addObjects)
128             ++ this.replaceObjects
129   }
130
131   pred Deltas::mergeAndApplyChangesForElementReplacement(
132     s: Stratum,
133     c: Element,
134     topmost: set Element,
135     -- who should I resemble, taking element replacement into account
136     iResembleDeltas_e: set Deltas)
137   {
138     -- expand out into a easier form for expressing well-formedness rule, where
            IDs don't count
139     this.objects = {p: Object, s: Stratum |
140        some n: ID | s->n->p in this.objects_e}
141
142     -- handle add, delete etc as if we are only taking resemblance into account
143     topmost = c =>
144        this.oldObjects_e[s] = iResembleDeltas_e.originalOldObjects_e[s]
145     else
146        this.oldObjects_e[s] =
147           (iResembleDeltas_e.originalObjects_e[s]
148            - iResembleDeltas_e.originalDeletedObjects_e[s]->Object)
149             ++ iResembleDeltas_e.originalReplacedObjects_e[s]
150
151     this.deletedObjects_e[s] =
152        iResembleDeltas_e.originalDeletedObjects_e[s]
153         - dom[iResembleDeltas_e.originalReplacedObjects_e[s]]
154
155     this.replacedObjects_e[s] = iResembleDeltas_e.originalReplacedObjects_e[s]
156
157     this.objects_e[s] =
158        (iResembleDeltas_e.originalObjects_e[s] - this.deletedObjects_e[s]->Object)
159          ++ this.replacedObjects_e[s]
160   }
```

Appendix C

The Backbone Formal Specification: Component Model

C.1 Component Model Signatures: `bb_structure.als`

```
1  module bb_structure
2
3  open base_deltas[PartID, Part] as Parts
4  open base_deltas[PortID, Port] as Ports
5  open base_deltas[ConnectorID, Connector] as Connectors
6  open base_deltas[AttributeID, Attribute] as Attributes
7  open base_deltas[OperationID, Operation] as Operations
8  open base_deltas[InterfaceImplementationID, InterfaceImplementation] as
       InterfaceImplementation
9  open base_deltas[ComponentImplementationID, ComponentImplementation] as
       ComponentImplementation
10 open base_deltas[PrimitiveTypeImplementationID, PrimitiveTypeImplementation] as
       PrimitiveTypeImplementation
11 open base_deltas[LinkID, Link] as Links
12
13 sig Component extends Element
14 {
15   myParts: lone Parts/Deltas,
16   myPorts: lone Ports/Deltas,
17   myConnectors: lone Connectors/Deltas,
18   myAttributes: lone Attributes/Deltas,
19   myCImplementation: lone ComponentImplementation/Deltas,
20   myLinks: lone Links/Deltas,
21
22   -- the final result, after taking replacement + resemblance into account
23   iDParts: PartID -> Part -> Stratum,
24   -- composite or leaf?
25   isComposite: set Stratum,
26   parts: Part -> Stratum,
```

```
27    ports: Port -> Stratum,
28    connectors: Connector -> Stratum,
29    attributes: Attribute -> Stratum,
30    cimplementation: ComponentImplementation -> Stratum,
31    links: Link -> Stratum,
32
33    -- the internal links, used for port type inferencing
34    inferredLinks: Port -> Port -> Stratum
35 }
36 {
37    -- rule: COMPONENT_RESEMBLANCE -- for components
38    replaces + resembles in Component
39    -- propagate up the objects from the delta into the sig, to make it more
            convenient
40    parts = myParts.objects
41    ports = myPorts.objects
42    connectors = myConnectors.objects
43    attributes = myAttributes.objects
44    cimplementation = myCImplementation.objects
45    links = myLinks.objects
46
47    -- form idParts
48    iDParts = {n: PartID, p: Part, s: Stratum | s -> n -> p in myParts.objects_e}
49 }
50
51 -- ensure each delta is composed by only one component
52 fact
53 {
54    all p: Parts/Deltas | one myParts.p
55    all p: Ports/Deltas | one myPorts.p
56    all c: Connectors/Deltas | one myConnectors.c
57    all a: Attributes/Deltas | one myAttributes.a
58    all i: ComponentImplementation/Deltas | one myCImplementation.i
59    all l: Links/Deltas | one myLinks.l
60 }
61
62 sig PrimitiveType extends Element
63 {
64    myTImplementation: lone PrimitiveTypeImplementation/Deltas,
65    -- the expanded elements
66    timplementation: PrimitiveTypeImplementation -> Stratum
67 }
68 {
69    replaces + resembles in PrimitiveType
70    -- propagate up the objects from the delta into the sig, to make it more
            convenient
71    timplementation = myTImplementation.objects
72 }
73 -- ensure each delta is composed by only one primitive type
74 fact
75 {
76    all t: PrimitiveTypeImplementation/Deltas | one myTImplementation.t
77 }
78
79 sig Interface extends Element
80 {
81    -- the deltas
```

```
82    myOperations: lone Operations/Deltas,
83    myIImplementation: lone InterfaceImplementation/Deltas,
84    -- the expanded elements
85    operations: Operation -> Stratum,
86    iimplementation: InterfaceImplementation -> Stratum,
87    superTypes: Interface -> Stratum
88  }
89  {
90    -- for interfaces
91    replaces + resembles in Interface
92    -- propagate up the objects from the delta into the sig, to make it more
          convenient
93    operations = myOperations.objects
94    iimplementation = myIImplementation.objects
95  }
96  -- ensure each delta is composed by only one interface
97  fact
98  {
99    all p: Operations/Deltas | one myOperations.p
100   all i: InterfaceImplementation/Deltas | one myIImplementation.i
101 }
102
103
104 -- each artifact must have a id, so it can be replaced or deleted
105 sig PartID, PortID, ConnectorID, AttributeID, OperationID,
        InterfaceImplementationID, ComponentImplementationID,
        PrimitiveTypeImplementationID, LinkID {}
106
107 sig Part
108 {
109   -- rule: WF_PART_TYPE -- each part must have a type
110   partType: Component,
111   -- remap a port from this part onto the port of a part that we are replacing
112   -- (new port -> old, replaced port)
113   portRemap: PortID lone -> lone PortID,
114   portMap: Stratum -> PortID lone -> lone Port,
115
116   -- the values of the attributes are set in the part   (child id -> parent id)
117   -- although they don't have to be set if we want to take the default
118   attributeValues: AttributeID -> lone AttributeValue,
119   -- do we alias a parent attribute?
120   attributeAliases: AttributeID -> lone AttributeID,
121   -- or do we simply copy a parent attribute, but retain our own state?
122   attributeCopyValues: AttributeID -> lone AttributeID,
123
124   -- derived state -- the parts that the connectors link to
125   linkedToParts: Part -> Stratum -> Component,
126   -- derived state -- any componts that the connectors link to
127   linkedToOutside: Stratum -> Component
128 }
129
130 abstract sig Index {}
131 one sig Zero, One, Two, Three extends Index {}
132
133 pred isContiguousFromZero(indices: set Index)
134 {
135   indices = indices.*(Three->Two + Two->One + One->Zero)
```

```
136  }
137
138  sig Port
139  {
140    -- set values are what the user has explicitly set
141    setProvided, setRequired: set Interface,
142    -- provided and required are inferred
143    provided, required: Interface -> Stratum -> Component,
144    mandatory, optional: set Index
145  }
146  {
147    -- mandatory indices start at 0, optional start from mandatory end, no overlap
148    -- all contiguous and must have some indices
149    -- rule: PORT_MULTIPLICITY
150    isContiguousFromZero[mandatory] and
151      isContiguousFromZero[mandatory + optional]
152    no mandatory & optional      -- no overlap
153    some mandatory + optional    -- but must have some indices
154  }
155
156  sig Connector
157  {
158      -- require 2 ends
159      ends: set ConnectorEnd
160  }
161  {
162    -- ensure 2 connector ends using a trick felix taught me
163    some disj end1, end2: ConnectorEnd | ends = end1 + end2
164      all end: ends |
165          end.otherEnd = ends - end
166  }
167
168  abstract sig ConnectorEnd
169  {
170      portID: PortID,
171      port: Port -> Stratum -> Component,
172      index: Index,
173      otherEnd: ConnectorEnd
174  }
175  {
176      -- an end is owned by one connector
177      one ends.this
178  }
179
180  sig ComponentConnectorEnd extends ConnectorEnd
181  {
182  }
183
184  sig PartConnectorEnd extends ConnectorEnd
185  {
186      partID: PartID,
187      cpart: Part -> Stratum -> Component
188  }
189
190
191  sig Attribute
192  {
```

```
193    -- rule: WF_ATTRIBUTE_TYPE -- an attribute must have a type
194    attributeType: PrimitiveType,
195    defaultValue: lone AttributeValue
196  }
197  {
198    -- rule: WF_ATTRIBUTE_DEFAULT
199    some defaultValue =>
200      defaultValue.valueType = attributeType
201  }
202
203  sig AttributeValue
204  {
205    valueType: PrimitiveType
206  }
207
208  sig Operation
209  {
210    -- this identifies the impelementation id and signature
211  }
212
213  sig InterfaceImplementation
214  {
215    -- this identifies the interface implementation clas or no s.dependsOns...
216  }
217
218  sig ComponentImplementation
219  {
220    -- this identifies the component implementation class...
221  }
222
223  sig PrimitiveTypeImplementation
224  {
225      -- this indentified the implementation of a primitive type
226  }
227
228  -- links are used for port inference
229  -- a bit like a connector, but multiplicity and optionality don't count
230  sig Link
231  {
232    linkEnds: PortID -> PortID
233  }
234  {
235    lone linkEnds
236  }
237
238  abstract sig LinkEnd
239  {
240    linkPortID: PortID,
241    linkError: Stratum -> Component
242    -- the internal interfaces are the interfaces presented inside the component
            content area
243    --   for a port, it is the interfaces seen internally (opposite)
244    --   for a port instance, it is the interfaces seen externally (same)
245  }
246
247  sig ComponentLinkEnd extends LinkEnd
248  {
```

```
249  }
250
251  sig PartLinkEnd extends LinkEnd
252  {
253    linkPartID: PartID
254  }
255
256  -- translate from port id to component link end -- guaranteed to be 1 per id
257  fun getComponentLinkEnd(id: PortID): one ComponentLinkEnd
258  {
259    { end: ComponentLinkEnd | end.linkPortID = id }
260  }
261
262  -- translate from a port/part to a part link end -- guaranteed to be 1 per pair
263  fun getPartLinkEnd(portID: PortID, partID: PartID): PartLinkEnd
264  {
265    { end: PartLinkEnd | end.linkPortID = portID and end.linkPartID = partID }
266  }
267
268  fun ComponentLinkEnd::getPort(s: Stratum, c: Component): one Port
269  {
270    c.myPorts.objects_e[s][this.linkPortID]
271  }
272
273  fun PartLinkEnd::getPortInstance(s: Stratum, c: Component): Port -> Part
274  {
275    let
276      cpart = c.myParts.objects_e[s][this.linkPartID],
277      cport = cpart.partType.myPorts.objects_e[s][this.linkPortID] |
278    cport -> cpart
279  }
280
281  -- get the port of a component connector
282  fun ComponentConnectorEnd::getPort(s: Stratum, c: Component): lone Port
283  {
284      c.myPorts.objects_e[s][this.portID]
285  }
286
287  -- should return only 1 Port, unless the component is invalid. NOTE: the
       component owns the part
288  fun PartConnectorEnd::getPortInstance(s: Stratum, c: Component): Port -> Part
289  {
290      let
291          ppart = c.myParts.objects_e[s][this.partID],
292          port = ppart.portMap[s][this.portID] |
293      port -> ppart
294  }
295
296  fun PartLinkEnd::getPortInstanceRequired(s: Stratum, c: Component): set
       Interface
297  {
298    let portPart = this.getPortInstance[s, c],
299      pport = dom[portPart],
300      ppartType = ran[portPart].partType
301    {
302      pport.required.ppartType.s
303    }
```

```
304  }
305
306  fun PartLinkEnd::getPortInstanceProvided(s: Stratum, c: Component): set
        Interface
307  {
308    let portPart = this.getPortInstance[s, c],
309      pport = dom[portPart],
310      ppartType = ran[portPart].partType
311    {
312      pport.provided.ppartType.s
313    }
314  }
```

C.2   Well-Formedness Rules: `bb_well_formed.als`

Listing C.2: bb_well_formed.als

```
1   module bb_well_formed
2
3   open bb_structure
4
5
6   -- check that the interface is well formed
7   pred interfaceIsWellFormed(s: Stratum, i: Interface)
8   {
9     -- should have only 1 operation definition per id
10    -- rule: WF_INTERFACE_OPERATION_PER_UUID
11    i.myOperations.oneObjectPerID[s]
12    -- we should only have one implementation, so if we resemble something
13    -- we must replace the implementation
14    -- rule: WF_INTERFACE_ONE_IMPLEMENTATION
15    one i.iimplementation.s
16  }
17
18  -- check that the primitive type is well formed
19  pred primitiveTypeIsWellFormed(s: Stratum, t: PrimitiveType)
20  {
21    -- rule: WF_PRIMITIVE_IMPLEMENTATION
22    one t.timplementation.s
23  }
24
25  -- check that the component is well formed
26  pred componentIsWellFormed(s: Stratum, c: Component)
27  {
28    -- the original (either the thing being replaced or the original)
29    -- is not in the composition hierarchy taking resemblance into account
30    -- NOTE: if c cannot be composed if it is a replacement
31    -- NOTE: a further constraint is that we cannot be composed of the thing we
32    --       are replacing
33    let
34      resembling = resembles_e.s, partTypes = parts.s.partType,
35      original = no c.replaces => c else c.replaces
36    {
```

```
37      -- rule: WF_COMPONENT_NO_SELF_COMPOSITION
38      original not in c.*(resembling + partTypes).partTypes
39    }
40    -- rule: WF_COMPONENT_PORTS
41    some c.ports.s
42
43    -- rule: WF_COMPONENT_PORT_PER_UUID -- max of one port per ID
44    c.myPorts.oneObjectPerID[s]
45    -- rule: WF_COMPONENT_ATTRIBUTE_PER_UUID -- max of one attr per ID
46    c.myAttributes.oneObjectPerID[s]
47
48    // if this is composite, ensure the ports, parts and connectors are well
          formed
49    s in c.isComposite =>
50    {
51      -- note: we will always have parts because that is what defines a composite
52      -- no implementation allowed: must be deleted or not there to begin with to
            be a well formed composite
53      no c.cimplementation.s
54      -- rule: WF_COMPONENT_LINKS_OK -- no links allowed
55      no c.links.s
56
57      -- rule: WF_COMPONENT_PART_PER_UUID -- max of one part per ID
58      c.myParts.oneObjectPerID[s]
59      -- rule: WF_COMPONENT_CONNECTOR_PER_UUID -- max of one connector per ID
60      c.myConnectors.oneObjectPerID[s]
61
62      -- internals must be well formed
63      partsAreWellFormed[s, c]
64      connectorsAreWellFormed[s, c]
65      portAndPortInstancesAreConnected[s, c]
66    }
67    else
68    {
69      -- won't have parts, as this is the definition of a leaf
70      -- can have a maximum of one implementation, either inherited or added, or
            replaced
71      -- rule: WC_COMPONENT_LEAF_IMPLEMENTATION
72      one c.cimplementation.s
73      -- a leaf cannot have connectors
74      no c.connectors.s
75
76      -- to be well formed, we must have one element per ID
77      c.myCImplementation.oneObjectPerID[s]
78      -- rule: WF_COMPONENT_LINK_PER_UUID -- max of one link per ID
79      c.myLinks.oneObjectPerID[s]
80
81      -- to be well formed, links must have no duplication and must refer to real
            ports in the stratum
82      -- note that links are allowed to loop back, as connectors can loop back
            through parts...
83      let l = c.links.s.linkEnds
84      {
85        no ~l & l
86        dom[l] + ran[l] in dom[c.myPorts.objects_e[s]]
87      }
88    }
```

```
89  }
90
91  pred portAndPortInstancesAreConnected(s: Stratum, c: Component)
92  {
93    all port: c.ports.s |
94      portIsConnected[s, c, port]
95
96    all cpart: c.parts.s |
97      all port: cpart.partType.ports.s |
98        portInstanceIsConnected[s, c, port, cpart]
99  }
100
101
102 pred partsAreWellFormed(s: Stratum, c: Component)
103 {
104   all pPart: c.parts.s
105   {
106     -- rule: WF_PART_NO_ISLANDS
107     -- it must be possible to reach this part from a series of connections from
            the owning component
108     -- otherwise, this part will be completely internally connected -- an island
109     s -> c in pPart.*(linkedToParts.c.s).linkedToOutside
110     -- check the attributes
111     let
112       valueIDs = dom[pPart.attributeValues],
113       aliasIDs = dom[pPart.attributeAliases],
114       copyIDs = dom[pPart.attributeCopyValues],
115       parentAttrs = c.myAttributes.objects_e[s],
116       partAttrs = pPart.partType.myAttributes.objects_e[s],
117       partAttrIDs = dom[partAttrs]
118     {
119       -- should have no overlap between the different types of possibilities
120       disj[valueIDs, aliasIDs, copyIDs]
121       -- all the IDs must exist in the list of attributes
122       (valueIDs + aliasIDs + copyIDs) in partAttrIDs
123
124       -- rule: WF_PART_SLOT_LITERAL -- any new values must have the correct type
125       all ID: valueIDs |
126         pPart.attributeValues[ID].valueType = partAttrs[ID].attributeType
127
128       -- rule: WF_PART_SLOT_ALIAS_COPY
129       -- any aliased or copied attributes must exist and have the correct type
130       all ID: aliasIDs |
131         partAttrs[ID].attributeType =
132           parentAttrs[pPart.attributeAliases[ID]].attributeType
133       all ID: copyIDs |
134         partAttrs[ID].attributeType =
135           parentAttrs[pPart.attributeCopyValues[ID]].attributeType
136
137       -- rule: WF_PART_SLOT_DEFAULT
138       -- anything left over must have a default value or else the parts
            attribute is unspecified
139       all ID: partAttrIDs - (valueIDs + aliasIDs + copyIDs) |
140         one partAttrs[ID].defaultValue
141     }
142   }
143 }
```

```
144
145
146   pred setupConnectors(s: Stratum, c: Component)
147   {
148     all end: c.connectors.s.ends |
149     {
150       -- if just one end of the connector goes to the component, it must be
                mandatory
151       -- if the part end is mandatory
152       end in ComponentConnectorEnd =>
153       {
154         end.port.c.s = (end & ComponentConnectorEnd)::getPort[s, c]
155       }
156       else
157       {
158         -- this is a part connector end, make sure we connect to a single port
                instance
159         let
160           portAndPart = (end & PartConnectorEnd)::getPortInstance[s, c],
161           resolvedPort = dom[portAndPart]
162         {
163           end.port.c.s = resolvedPort
164           end.cpart.c.s = ran[portAndPart]
165         }
166     }
167   }
168 }
169
170 pred connectorsAreWellFormed(s: Stratum, c: Component)
171 {
172     all end: c.connectors.s.ends |
173     let other = end.otherEnd, aport = end.port.c.s, otherPort = other.port.c.s |
174     {
175       end.index in end.port.c.s.(mandatory + optional)
176       one end.port.c.s
177
178       -- if just one end of the connector goes to the component, it must be
                mandatory
179       -- if the part end is mandatory
180       end in ComponentConnectorEnd =>
181       {
182         -- note: other end must be a part connector end, as no component to
                component connectors are allowed
183         -- if the outside is optional, the inside cannot be mandatory...
184         -- rule: WF_CONNECTOR_OPTIONAL
185         end.index in aport.optional => other.index in otherPort.optional
186       }
187       else
188       {
189         one end.cpart.c.s
190         -- rule: WF_CONNECTOR_SAME
191         end.index in aport.optional <=> other.index in otherPort.optional
192       }
193     }
194 }
195
196
```

```
197
198    ----------- support predicates
199
200    pred portIsConnected(s: Stratum, c: Component, o: Port)
201    {
202      -- ports on the component must always be connected internally
203      all idx: o.mandatory + o.optional |
204      one end: c.connectors.s.ends & ComponentConnectorEnd |
205      {
206        end.port.c.s = o
207        idx = end.index
208      }
209    }
210
211    pred portInstanceIsConnected(s: Stratum, c: Component, o: Port, p: Part)
212    {
213      -- don't need to check any provided interfaces unless these are not optional
214      (some o.required.c.s or isFalse[Model::providesIsOptional]) =>
215      {
216        -- match up any mandatory required interfaces on the port with a single
217             connector
         -- rule: WF_CONNECTOR_INDEX
218        all idx: o.mandatory |
219          one end: c.connectors.s.ends & PartConnectorEnd |
220            end.port.c.s = o and end.cpart.c.s = p and idx = end.index
221        all idx: o.optional |
222          lone end: c.connectors.s.ends & PartConnectorEnd |
223            end.port.c.s = o and end.cpart.c.s = p and idx = end.index
224      }
225    }
```

C.3   Component Model Facts: `bb.als`

Listing C.3: bb.als

```
1    module bb
2
3    -- opening the two structure modules here is not strictly necessary, but it
          simplifies the naming
4    open base_structure as base
5    open bb_structure as full
6
7    open bb_well_formed
8    open bb_port_inference
9    open base_facts
10
11
12    --------------------------------------------------------------------
13    -- handle any extra rules for interfaces
14    --------------------------------------------------------------------
15
16    fact InterfaceFacts
17    {
```

```
18    all i: Interface |
19    let
20      owner = i.home
21    {
22      -- we only need to form a definition for stratum that can see us
23      all s: Stratum |
24      let
25        invalid = s in i.isInvalid_e,
26        visible = owner in s.transitivePlusMe
27      {
28        -- if we can see this interface, test to see if it is valid in this
              stratum
29        not visible =>
30          invalidateUnseenInterface[s, i]
31        else
32        {
33          -- ensure that the subtypes are set up correctly
34          -- this is a subtype of an interface if we can reach it transitively and
35          -- our set of operations are a super-set of the super type's
36          -- of operationID -> Operation.  i.e. if you replace an operationID you
                are breaking subtype
37          -- so the operationID is the name, and the Operation is the full spec
                which is assumed to have
38          -- changed in a replacement...
39          -- NOTE: the super types are direct -- to follow use closure
40          --       also note that we only need/want supertypes for non-primes
41          no i.replaces =>
42          {
43            i.superTypes.s =
44              { super: i.resembles |
45                super.myOperations.objects_e[s] in
46                  i.myOperations.objects_e[s] }
47          }
48          else
49            no i.superTypes.s
50
51          -- merge any parts and apply changes
52          let topmost = getTopmost[s, i] & Interface
53          {
54            i.myOperations::mergeAndApplyChangesForResemblance[
55              s, i, i.resembles_e.s.myOperations]
56            i.myOperations::mergeAndApplyChangesForElementReplacement[
57              s, i, topmost, topmost.myOperations]
58            i.myIImplementation::mergeAndApplyChangesForResemblance[
59              s, i, i.resembles_e.s.myIImplementation]
60            i.myIImplementation::mergeAndApplyChangesForElementReplacement[
61              s, i, topmost, topmost.myIImplementation]
62          }
63
64          -- the interface must be valid in the place it was defined
65          (s = owner or s not in Model.errorsAllowed) => !invalid
66          s = owner =>
67          {
68            i.myOperations.deltasIsWellFormed[s]
69            i.myIImplementation.deltasIsWellFormed[s]
70          }
71
```

```
72              -- a component is valid if it is well formed...
73              !invalid <=> interfaceIsWellFormed[s, i]
74          }
75        }
76      }
77  }
78
79
80  ----------------------------------------------------------------
81  -- handle any extra rules for primitive types
82  ----------------------------------------------------------------
83
84  fact PrimitiveTypeFacts
85  {
86    all t: PrimitiveType |
87    let
88      owner = t.home
89    {
90      -- we only need to form a definition for stratum that can see us
91      all s: Stratum |
92      let
93        invalid = s in t.isInvalid_e,
94        visible = owner in s.transitivePlusMe
95        {
96          -- if we can see this interface, test to see if it is valid in this
                 stratum
97          not visible =>
98            invalidateUnseenPrimitiveType[s, t]
99          else
100         {
101           -- merge any parts and apply changes
102           let topmost = getTopmost[s, t] & PrimitiveType
103           {
104             t.myTImplementation::mergeAndApplyChangesForResemblance[
105               s, t, t.resembles_e.s.myTImplementation]
106             t.myTImplementation::mergeAndApplyChangesForElementReplacement[
107               s, t, topmost, topmost.myTImplementation]
108           }
109
110           -- the primitive type must be valid in the place it was defined
111           (s = owner or s not in Model.errorsAllowed) => !invalid
112           s = owner =>
113             t.myTImplementation.deltasIsWellFormed[s]
114
115           -- a component is valid if it is well formed...
116           !invalid <=> primitiveTypeIsWellFormed[s, t]
117         }
118       }
119   }
120 }
121
122 ----------------------------------------------------------------
123 -- handle any extra rules for components
124 ----------------------------------------------------------------
125
126 fact ComponentFacts
127 {
```

```
128    -- rule: COMPONENT_REPLACEMENT_NOT_REFERENCED
129    -- no part or port can refer explicitly to a replaced component or interface
130    no partType.replaces
131    no (setProvided + setRequired).replaces
132
133    all c: Component |
134    let
135      owner = c.home,
136      -- strata that the component can see
137      iCanSeePlusMe = owner.canSeePlusMe,
138      types = c.myParts.newObjects.partType,
139      attrTypes = c.myAttributes.newObjects.attributeType,
140      interfaces = c.myPorts.addedObjects.(setRequired + setProvided)
141    {
142      -- resemblance has no redundancy
143      c.resembles = c.resembles - c.resembles.^resembles
144
145      -- attribute types, part types and port interfaces must be visible
146      -- rule: PART_TYPE_VISIBLE
147      types.home in iCanSeePlusMe
148      -- rule: PORT_INTERFACE_VISIBIILITY
149      interfaces.home in iCanSeePlusMe
150      -- rule: ATTRIBUTE_TYPE_VISIBILITY
151      attrTypes.home in iCanSeePlusMe
152
153      -- ensure that the port remaps are correctly formed for the stratum they are
             owned by
154      let delta = c.myParts |
155      all p: delta.newObjects | -- parts of the delta
156      let
157        partID = delta.replaceObjects.p,
158        oldPart = delta.oldObjects_e[owner][partID],
159        remap = p.portRemap,
160        newPortIDs = dom[remap],
161        oldPortIDs = ran[remap]
162      {
163        -- we can only alias ports that we actually have
164        newPortIDs in dom[p.partType.myPorts.objects_e[owner]]
165        -- we can only use port ids of the component we are replacing
166        oldPortIDs in dom[oldPart.partType.myPorts.objects_e[owner]]
167
168        -- each port we remap should have a different id, or there's no point
169        -- this is not strictly needed, but ensures nice witnesses
170        bijection[remap, newPortIDs, oldPortIDs]
171
172        -- can't map a port id onto the same id
173        no remap & iden
174      }
175
176      -- we only need to form a definition for stratum that can see us
177      all s: Stratum |
178      let
179        invalid = s in c.isInvalid_e,
180        visible = owner in s.transitivePlusMe
181      {
182        -- if we can see this component, test to see if it is valid in this
               stratum
```

```
183          not visible =>
184            invalidateUnseenComponent[s, c]
185          else
186          let topmost = getTopmost[s, c] & Component
187          {
188            -- this is a composite in this stratum if it has parts
189            s in c.isComposite <=> some c.parts.s
190
191            -- merge any parts and apply changes
192            c.myParts::mergeAndApplyChangesForResemblance[
193              s, c, c.resembles_e.s.myParts]
194            c.myParts::mergeAndApplyChangesForElementReplacement[
195              s, c, topmost, topmost.myParts]
196
197            -- merge any ports and apply changes
198            c.myPorts::mergeAndApplyChangesForResemblance[
199              s, c, c.resembles_e.s.myPorts]
200            c.myPorts::mergeAndApplyChangesForElementReplacement[
201              s, c, topmost, topmost.myPorts]
202
203            -- merge any connectors and apply changes
204            c.myConnectors::mergeAndApplyChangesForResemblance[
205              s, c, c.resembles_e.s.myConnectors]
206            c.myConnectors::mergeAndApplyChangesForElementReplacement[
207              s, c, topmost, topmost.myConnectors]
208
209            -- merge any attributes and apply changes
210            c.myAttributes::mergeAndApplyChangesForResemblance[
211              s, c, c.resembles_e.s.myAttributes]
212            c.myAttributes::mergeAndApplyChangesForElementReplacement[
213              s, c, topmost, topmost.myAttributes]
214
215            -- merge any implementations and apply changes
216            c.myCImplementation::mergeAndApplyChangesForResemblance[
217              s, c, c.resembles_e.s.myCImplementation]
218            c.myCImplementation::mergeAndApplyChangesForElementReplacement[
219              s, c, topmost, topmost.myCImplementation]
220
221            -- merge any implementations and apply changes
222            c.myLinks::mergeAndApplyChangesForResemblance[
223              s, c, c.resembles_e.s.myLinks]
224            c.myLinks::mergeAndApplyChangesForElementReplacement[
225              s, c, topmost, topmost.myLinks]
226
227            -- if we are "home", all the deltas must be well formed...
228            -- this is not necessarily the case if we are not home
229            s = owner =>
230            {
231              c.myParts.deltasIsWellFormed[s]
232              c.myPorts.deltasIsWellFormed[s]
233              c.myConnectors.deltasIsWellFormed[s]
234              c.myAttributes.deltasIsWellFormed[s]
235              c.myCImplementation.deltasIsWellFormed[s]
236              c.myLinks.deltasIsWellFormed[s]
237            }
238
239          setupParts[s, c]
```

```
240            setupConnectors[s, c]
241            s in c.isComposite =>
242              setupCompositeLinks[s, c]
243            else
244              setupLeafLinks[s, c]
245
246            -- the component must be valid in the place it was defined
247            (s = owner or s not in Model.errorsAllowed) => !invalid
248
249            -- a component is invalid iff it is not well formed
250            -- rule: COMPONENT_OK_AT_HOME
251            invalid <=>
252              (!componentIsWellFormed[s, c] or !linksAreWellFormed[s, c])
253          }
254        }
255      }
256 }
257
258 pred setupParts(s: Stratum, c: Component)
259 {
260    -- reference the parts we are linked to and link to the outside if true
261    let allParts = c.parts.s
262    {
263      no (Part - allParts).linkedToParts.c.s
264      all pPart: allParts |
265      {
266        pPart.linkedToParts.c.s =
267        { p: allParts - pPart |
268          some end: c.connectors.s.ends |
269            end.cpart.c.s = pPart and end.otherEnd.cpart.c.s = p }
270
271        -- reference if we are linked to the outside of the component
272        s -> c in pPart.linkedToOutside <=>
273        {
274          some end: c.connectors.s.ends |
275            end.cpart.c.s = pPart and end.otherEnd in ComponentConnectorEnd
276        }
277      }
278    }
279
280    -- form the full port map for this stratum, taking remap into account
281    all p: c.myParts.newObjects |
282    let
283      -- turn the remap from id -> id to id -> port
284      idToPort = p.partType.myPorts.objects_e[s],
285      newPorts = idToPort[PortID],
286      remap =
287        { newPort: newPorts, oldID: PortID |
288          idToPort.newPort -> oldID in p.portRemap }
289    {
290      -- remove the existing ID of the port before adding the new one
291      ~(p.portMap[s]) =
292        ~(p.partType.myPorts.objects_e[s]) ++ remap
293    }
294 }
295
296
```

```
297  -- if the connector is not visibile to a component in a stratum, it should be
         zeroed out to
298  -- make it easier to interpret the results and and zeroOutUnseenElement[s, e]
         cut back on the state space for performance reasons
299  fact ZeroOutUnseenConnectorsFact
300  {
301    all conn: Connector, c: Component, s: Stratum |
302    conn not in c.connectors.s =>
303    {
304      all end: conn.ends
305      {
306        no end.port.c.s
307        no end.cpart.c.s
308      }
309    }
310  }
311
312  -- if the part is not visible to a component in a stratum, it should also be
         zeroed out
313  -- for understandability and performance reasons
314  fact ZeroOutUnseenPartsFact
315  {
316    all p: Part, c: Component, s: Stratum |
317    p not in c.parts.s =>
318    {
319      no p.linkedToParts.c.s
320      s-> c not in p.linkedToOutside
321    }
322  }
323
324  -- if the part is not visible to a component in a stratum, it should also be
         zeroed out
325  -- for understandability and performance reasons
326  -- NOTE: if you move the valid setting up into the main body, it gets slow
327  pred invalidateUnseenComponent(s: Stratum, c: Component)
328  {
329    s not in c.isInvalid_e -- it isn't invalid here
330    no c.parts.s
331    no c.iDParts.s
332    no c.ports.s
333    no c.connectors.s
334    no c.attributes.s
335    no c.cimplementation.s
336    no c.inferredLinks.s
337    no c.links.s
338    s not in c.isComposite
339    c.myParts::nothing[s]
340    c.myAttributes::nothing[s]
341    c.myPorts::nothing[s]
342    c.myConnectors::nothing[s]
343    c.myCImplementation::nothing[s]
344    c.myLinks::nothing[s]
345  }
346
347  pred invalidateUnseenInterface(s: Stratum, i: Interface)
348  {
349    s not in i.isInvalid_e -- it isn't invalid here
```

```
350    no i.operations.s
351    no i.iimplementation.s
352    i.myOperations::nothing[s]
353    i.myIImplementation::nothing[s]
354    no i.superTypes.s
355  }
356
357  pred invalidateUnseenPrimitiveType(s: Stratum, t: PrimitiveType)
358  {
359    s not in t.isInvalid_e -- it isn't invalid here
360    no t.timplementation.s
361    t.myTImplementation::nothing[s]
362  }
363
364  pred invalidateUnseenPorts()
365  {
366    all s: Stratum, c: Component
367    {
368      all p: Port |
369        p not in c.ports.s =>
370          no p.(provided + required).c.s
371    }
372  }
373
374  -- some predicates to help with structuring a model
375  pred Model::providesIsOptional()
376  {
377    Model.providesIsOptional = True
378  }
379
380  pred Model::providesIsNotOptional()
381  {
382    Model.providesIsOptional = False
383  }
384
385  pred Model::noErrorsAllowed()
386  {
387    no this.errorsAllowed
388  }
389
390  pred Model::errorsAllowedInTopOnly()
391  {
392    this.errorsAllowed = isTop.True
393  }
394
395  pred Model::topDefinesNothing()
396  {
397    no isTop.True.ownedElements
398  }
399
400  pred Model::definesNothing(s: Stratum)
401  {
402    no s.ownedElements
403  }
404
405  pred Model::errorsOnlyAllowedInTopAndOthers(others: set Stratum)
406  {
```

```
407    this.errorsAllowed = others + isTop.True
408  }
409
410  pred Model::errorsOnlyAllowedIn(others: set Stratum)
411  {
412    this.errorsAllowed = others
413  }
414
415  pred Model::forceErrors(errorStrata: set Stratum)
416  {
417    all e: errorStrata |
418      some isInvalid_e.e
419  }
```

C.4   Port Type Inference Logic: **bb_port_inference.als**

Listing C.4: bb_port_inference.als

```
1   module bb_port_inference
2
3   open bb_structure
4   open bb_inference_help
5
6   pred setupLeafLinks(s: Stratum, c: Component)
7   {
8     -- make sure we have enough link ends
9     ensureLinkEndsExist[s, c]
10
11    let
12      idToPorts = c.myPorts.objects_e[s],
13      inferred =
14        { p1, p2: ran[idToPorts] |
15          idToPorts.p1 -> idToPorts.p2 in c.links.s.linkEnds }
16    {
17      -- copy over the links
18      c.inferredLinks.s = inferred
19
20      -- copy over the sets
21      all cport: c.ports.s |
22      let
23        end = getComponentLinkEnd[idToPorts.cport],
24        -- rule: WF_LINK_COMPLEMENTARY
25        errors =
26          some other: c.ports.s |
27            cport -> other in c.inferredLinks.s and
28              (cport.required.c.s != other.provided.c.s or
29               cport.provided.c.s != other.required.c.s)
30      {
31        -- propagate the set value into the inferred value
32        -- rule: WF_COMPONENT_LEAF_PORTS
33        cport.required.c.s = cport.setRequired
34        cport.provided.c.s = cport.setProvided
35
```

```
36        -- we have no errors if all linked match up exactly
37        s -> c in end.linkError <=> errors
38      }
39    }
40  }
41
42  pred setupCompositeLinks(s: Stratum, c: Component)
43  {
44    ensureLinkEndsExist[s, c]
45
46    let
47      allPorts = c.ports.s,
48      allParts = c.parts.s,
49      idToPorts = c.myPorts.objects_e[s],
50      idToParts = c.myParts.objects_e[s],
51      -- flatten everything into a LinkEnd->LinkEnd structure so we can
52      -- use transitive closure to navigate
53      portToPort =  makePortToPort[s, c],
54      partInternal = makePartInternal[s, c],
55      partToPart =  makePartToPart[s, c],
56      portToPart =  makePortToPart[s, c],
57      partToPort =  ~portToPart,
58      -- we connect by going from a port to a port,
59      -- or from a port to part to possibly the other side of the part
60      -- and then onto another part etc, until we get to a final part,
61      -- or to a final port
62      fromPortToPart = portToPart.*(partInternal.partToPart),
63      fromPartToPort = ~fromPortToPart,
64      fromPartToPart = partToPart.*(partInternal.partToPart),
65      -- harsh allows us to bounce around looking for any possibly connected other
             elements.
66      -- used to disallow inferredLinks via tainting
67      harshFromPortToAny =
68        portToPart.*(portToPart + partToPort + portToPort + partInternal +
             partToPart),
69      fromPortToPort = portToPort + fromPortToPart.partInternal.partToPort
70    {
71      -- set up the inferred links, propagating the constraints to the next level
72      propagateInferredCompositeLinks[
73        s, c, harshFromPortToAny, fromPortToPort,
74        partInternal, portToPort]
75
76      -- get the provided and required interfaces of ports
77      all cport: allPorts |
78      let
79        end = getComponentLinkEnd[idToPorts.cport],
80
81        infReq = cport.required.c.s,
82        reqEnds = end.fromPortToPart,
83        requiresFromEnds =
84          { r: Interface  |
85            some ce: reqEnds |
86              r in ce.getPortInstanceRequired[s, c] },
87      matchingRequires = extractLowestCommonSubtypes[s, requiresFromEnds],
88
89        infProv = cport.provided.c.s,
90        provEnds = end.fromPortToPort - end +
```

```
 91            { e: PartLinkEnd |
 92              e in end.fromPortToPart and no e.partInternal
 93            },
 94        providesFromEnds =
 95            { p: Interface |
 96              {
 97                (some e: provEnds & ComponentLinkEnd |
 98                  p in e.getPort[s, c].required.c.s)
 99                or
100                (some e: provEnds & PartLinkEnd |
101                  p in e.getPortInstanceProvided[s, c])
102              }
103            },
104      matchingProvides = extractHighestCommonSupertypes[s, providesFromEnds]
105      {
106        infReq = matchingRequires
107        infProv = matchingProvides
108
109        -- rule: WF_CONNECTOR_ONE_TO_ONE
110        s -> c not in end.linkError <=>
111          (oneToOneProvidedMappingExists[s, c, infProv, reqEnds] and
112           oneToOneRequiredMappingExists[s, c, infReq, reqEnds])
113      }
114
115      -- enforce the constraints for each port instance
116      all cpart: allParts,
117        cport: cpart.partType.ports.s |
118      let end = getPartLinkEnd[cpart.portMap[s].cport, idToParts.cpart]
119      {
120
121        {
122          let
123            infReq = end.getPortInstanceRequired[s, c],
124            infProv = end.getPortInstanceProvided[s, c],
125            terminalEnds = end.fromPartToPort +
126              { e: PartLinkEnd |
127                e in end.fromPartToPart and no e.partInternal
128              },
129            provFromTerminalEnds =
130              { p: Interface |
131                {
132                  (some e: terminalEnds & ComponentLinkEnd |
133                    p in e.getPort[s, c].provided.c.s)
134                  or
135                  (some e: terminalEnds & PartLinkEnd |
136                    p in e.getPortInstanceRequired[s, c])
137                }
138              },
139            allEnds = end.fromPartToPort +
140              { e: PartLinkEnd |
141                e in end.fromPartToPart
142              },
143            matchingTerminalProvides =
144              extractLowestCommonSubtypes[s, provFromTerminalEnds]
145          {
146            s -> c not in end.linkError <=>
147              {
```

```
148                  -- rule: WF_CONNECTOR_PROVIDES_ENOUGH
149                  no end.partInternal =>
150                    providesEnough[s, infProv, matchingTerminalProvides]
151                  oneToOneRequiredMappingExists[s, c, infProv, allEnds]
152                  oneToOneProvidedMappingExists[s, c, infReq, allEnds]
153                }
154              }
155            }
156          }
157        }
158  }
159
160  pred linksAreWellFormed(s: Stratum, c: Component)
161  {
162    let
163      allPorts = c.ports.s,
164      allParts = c.parts.s,
165      idToParts = c.myParts.objects_e[s],
166      portToPort =   makePortToPort[s, c]
167    {
168      -- enforce that no ports connect directly to each other
169      -- as this can lead to nondeterministic interface assigment
170      -- rule: WF_CONNECTOR_NO_PORT_TO_PORT
171      s in c.isComposite =>
172        no portToPort
173
174      -- enforce the constraints for each port
175      all cport: allPorts |
176      let
177        infProv = cport.provided.c.s,
178        infReq = cport.required.c.s,
179        idToPorts = c.myPorts.objects_e[s],
180        end = getComponentLinkEnd[idToPorts.cport],
181        amHome = c.home = s,
182        setInterfaces = cport.(setProvided + setRequired)
183      {
184        -- check any set values only if we are "home"
185        (amHome and some setInterfaces) =>
186        {
187          infProv = cport.setProvided
188          infReq = cport.setRequired
189        }
190
191        -- rule: WF_PORT_SOME_INTERFACES -- a port must have some interfaces
192        some infProv + infReq
193        s -> c not in end.linkError
194      }
195
196      -- enforce the constraints for each port instance
197      all cpart: allParts,
198        cport: cpart.partType.ports.s |
199      let
200        end = getPartLinkEnd[cpart.portMap[s].cport, idToParts.cpart],
201        infReq = end.getPortInstanceRequired[s, c],
202        infProv = end.getPortInstanceProvided[s, c]
203      {
204        -- must have some interfaces
```

```
205        some infProv + infReq
206        s -> c not in end.linkError
207      }
208    }
209 }
210
211
212 -- set up the inferred links for this component for a leaf, just use the links
213 -- for a composite, trace through from port to port, but only infer a link if
      there
214 -- is no terminal part involve anywhere
215 pred propagateInferredCompositeLinks(
216   s: Stratum,
217   c: Component,
218   harshFromPortToAny:  ComponentLinkEnd -> LinkEnd,
219   fromPortToPort:      ComponentLinkEnd -> ComponentLinkEnd,
220   partInternal:        PartLinkEnd -> PartLinkEnd,
221   portToPort:          ComponentLinkEnd -> ComponentLinkEnd)
222 {
223   let
224     idToPorts = c.myPorts.objects_e[s],
225     terminateInternallyIDs =
226     { id: dom[idToPorts] & PortID |
227       some end: PartLinkEnd |
228       let instance = end.getPortInstance[s, c],
229         cport = dom[instance],
230         cpart = ran[instance]
231       {
232         getComponentLinkEnd[id] in harshFromPortToAny.end
233         no end.partInternal
234         -- only provided terminals break linking
235         some cport.provided.(cpart.partType).s
236       }
237     }
238   {
239     -- find all port->port combinations that go through a leaf part and link up
240     -- but which don't have a termination on a provided port instance interface
241     let inferred =
242     { p1, p2: Port |
243       some end: dom[fromPortToPort] | let other = end.fromPortToPort
244       {
245         -- no connector loopbacks on port instances
246         disj[end, other]
247         p1 = idToPorts[end.linkPortID]
248         p2 = idToPorts[other.linkPortID]
249         -- if we can reach a port which links internally,
250         -- do not create an alias
251         no (end + other).*portToPort.linkPortID & terminateInternallyIDs
252       }
253     } |
254     c.inferredLinks.s = inferred
255   }
256 }
```

C.5 Port Type Inference Support: **bb_inference_help.als**

Listing C.5: bb_inference_help.als

```
1   module bb_inference_help
2
3   open bb_structure
4
5
6   pred providesEnough(s: Stratum,
7       provided: set Interface, required: set Interface)
8   {
9     all prov: provided |
10      one req: required |
11        req in prov.*(superTypes.s)
12    -- ensure that it works the other way around also
13    all req: required |
14      one prov: provided |
15        req in prov.*(superTypes.s)
16  }
17
18  pred oneToOneProvidedMappingExists[s: Stratum, c: Component, provided: set
        Interface, ends: LinkEnd]
19  {
20    all end: ends & ComponentLinkEnd |
21      oneToOneMappingExists[s, provided, end.getPort[s, c].required.c.s]
22
23    all end: ends & PartLinkEnd |
24      oneToOneMappingExists[s, provided, end.getPortInstanceProvided[s, c]]
25  }
26
27  pred oneToOneRequiredMappingExists[s: Stratum, c: Component, required: set
        Interface, ends: LinkEnd]
28  {
29    all end: ends & ComponentLinkEnd |
30      oneToOneMappingExists[s, required, end.getPort[s, c].provided.c.s]
31
32    all end: ends & PartLinkEnd |
33      oneToOneMappingExists[s, required, end.getPortInstanceRequired[s, c]]
34  }
35
36  pred oneToOneMappingExists[s: Stratum, a: set Interface, b: set Interface]
37  {
38    all aa: a |
39      one bb: b |
40        bb in expand[s, aa]
41
42    -- ensure that it works the other way around
43    all bb: b |
44      one aa: a |
45        bb in expand[s, aa]
46  }
47
48  fun extractHighestCommonSupertypes(s: Stratum, require: Interface): set
        Interface
```

```
49  {
50    let
51      -- map is an interface in require (i) with all matching interfaces in
             require (e)
52      map =
53      { i: require, e: require |
54        some expand[s, i] & expand[s, e] },
55      highestCommonSupertypes =
56      {
57        super: Interface |
58          some i: require |
59            super = highestCommonSupertype[s, map[i]]
60      }
61    {
62      highestCommonSupertypes
63    }
64  }
65
66  fun highestCommonSupertype(s: Stratum, required: set Interface): lone Interface
67  {
68    { i: Interface |
69      {
70        required in *(superTypes.s).i
71        no sub: superTypes.s.i |
72          required in *(superTypes.s).sub
73      }
74    }
75  }
76
77  fun extractLowestCommonSubtypes(s: Stratum, require: Interface): set Interface
78  {
79    let
80      -- map is an interface in require (i) with all matching interfaces in
             require (e)
81      map =
82      { i: require, e: require |
83        some expand[s, i] & expand[s, e] },
84      lowestCommonSubtypes =
85      {
86        sub: Interface |
87          some i: require |
88            sub = lowestCommonSubtype[s, map[i]]
89      }
90    {
91      lowestCommonSubtypes
92    }
93  }
94
95  fun lowestCommonSubtype(s: Stratum, required: set Interface): lone Interface
96  {
97    { i: Interface |
98      {
99        required in i.*(superTypes.s)
100       no super: i.superTypes.s |
101         required in super.*(superTypes.s)
102     }
103   }
```

```
104  }
105
106  fun expand(s: Stratum, i: Interface): set Interface
107  {
108    -- expand forms the full expanded supertype and subtype hierarchy
109    i.*(superTypes.s) +  ^(superTypes.s).i
110  }
111
112
113  -- a generator axiom to ensure that we have a unique link end per port, or port
        instance
114  pred ensureLinkEndsExist(s: Stratum, c: Component)
115  {
116    let
117      idToPorts = c.myPorts.objects_e[s],
118      idToParts = c.myParts.objects_e[s]
119    {
120      -- set up the linkends
121      -- ensure all ports have a link end
122      all portID: dom[idToPorts] |
123        one l: ComponentLinkEnd |
124          l.linkPortID = portID
125
126      -- ensure all part/ports have a link end
127      all ppart: c.parts.s |
128        let partID = idToParts.ppart |
129          all portID: ppart.portMap[s].Port |
130            one l: PartLinkEnd |
131              l.linkPortID = portID and l.linkPartID = partID
132    }
133  }
134
135  fun makePortToPort(s: Stratum, c: Component): ComponentLinkEnd ->
        ComponentLinkEnd
136  {
137    -- only links can go from port to port, connectors can't
138    s not in c.isComposite =>
139    { p1, p2: ComponentLinkEnd |
140        let actualLinks = c.links.s.linkEnds |
141      some end: dom[actualLinks], other: actualLinks[end]
142      {
143        -- these are disjoint because of a clause in bb_well_formed.als
144        end = p1.linkPortID
145        other = p2.linkPortID
146      }
147    }
148    else
149      none -> none
150  }
151
152  fun makePartInternal(s: Stratum, c: Component): PartLinkEnd -> PartLinkEnd
153  {
154    let
155      idToParts = c.myParts.objects_e[s] |
156    { p1, p2: PartLinkEnd |
157      some partID: dom[idToParts] |
158        let
```

```
159          realPart = idToParts[partID],
160          realType = realPart.partType,
161          inferredOneWay = realType.inferredLinks.s,
162          inferred = inferredOneWay + ~inferredOneWay,
163          idToPorts = realPart.portMap[s],
164          realPort = idToPorts[p1.linkPortID]
165      {
166        disj[p1, p2]
167        p1.linkPartID = partID
168        p2.linkPartID = partID
169        realPort in dom[inferred]
170        idToPorts[p2.linkPortID] = inferred[realPort]
171      }
172    }
173  }
174
175  fun makePartToPart(s: Stratum, c: Component): PartLinkEnd -> PartLinkEnd
176  {
177    { p1, p2: PartLinkEnd |
178      some end: c.connectors.s.ends | let other = end.otherEnd
179      {
180        disj[end, other]
181        end.portID = p1.linkPortID
182        end.partID = p1.linkPartID
183        other.portID = p2.linkPortID
184        other.partID = p2.linkPartID
185      }
186    }
187  }
188
189
190  fun makePortToPart(s: Stratum, c: Component): ComponentLinkEnd -> PartLinkEnd
191  {
192    { p1: ComponentLinkEnd, p2: PartLinkEnd |
193      some end: c.connectors.s.ends | let other = end.otherEnd
194      {
195        end in ComponentConnectorEnd
196        end.portID = p1.linkPortID
197        other.portID = p2.linkPortID
198        other.partID = p2.linkPartID
199      }
200    }
201  }
```

Appendix D

A Brief Introduction to Alloy

In this appendix, we provide a brief overview of Alloy in order to aid further understanding of the formal specifications presented in this thesis. The definitive source of information on Alloy modelling is [Jac06].

Alloy [Jac02, Jac09] is a first-order relational logic which is well suited for modelling and reasoning about the properties of an object-oriented system. Alloy is based on the foundations of the formal language Z [Spi92], and has been carefully designed to allow models to be comprehensively checked for counterexamples and witnesses within a finite state space. Visualisation and animation facilities are provided by the toolset. The Alloy language does not have recursive functions and therefore is not well suited to some domains. It has, however, proven to be expressive enough to model a wide range of primarily structural problems. Although first-order, Alloy allows existential qualifiers which are transformed using skolemisation.

The Alloy `sig` construct (short for signature) is used to represent a collection of objects. Signatures can also contain fields, and each field has an associated multiplicity. In this sense, signatures are analogous to classes in an object-oriented system.

Building minimally on the first example in [Jac06], an address book can be modelled as below.

```
sig Name, Addr {}
sig Book {
  owners: set Name,
  entries: some Name -> lone Addr
}
```

`Book` represents an address book, `Name` represents a person via their name, and `Addr` denotes an address. Each book can be owned by a set of people (or have no owner at all), and contains a set of tuples (`entries`) where each tuple maps from a name to an address.

To check an assertion, we phrase it as an Alloy expression and look for a counterexample using the analyser in the Alloy toolset. For instance, we can assert that it is not possible for two books to share

owners, by stating that if two books are not equal then this implies (=>) that their owners do not intersect (&).

```
assert noSharedOwners {
    all b1, b2: Book | b1 != b2 => no b1.owners & b2.owners }
```

To generate a counterexample, we must indicate the size of the space to search. This is specified in terms of number of allowed instances of each signature. To check our assertion with up to 3 instances of each signature, we can use the following.

```
check noSharedOwners for 3
```

Alloy produces a counterexample like figure D.1 showing that our assertion was incorrect: books can indeed share owners as nothing in the model currently prohibits this.
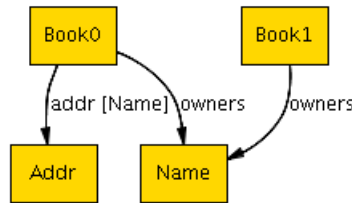

Figure D.1: A counterexample generated by the Alloy analyser

To prevent this sharing, we can define a predicate to define what we mean by two books sharing an owner.

```
pred ownersShared(b1, b2: Book) { some b1.owners & b2.owners }
```

We can then use a fact to assert that no (disjoint) books are allowed to share owners.

```
fact noSharing { all disj b1, b2: Book | !ownersShared(b1, b2) }
```

Checking our assertion along with this fact now generates no counterexamples, within the space of 3 instances for each signature. We can be sure that our assertion is valid within this finite space because we were previously able to generate a counterexample. This technique of finding a counterexample before adding new facts allows us to ensure that the state space is large enough to find meaningful structures in the first place. Clearly, checking for counterexamples is pointless if the state space is too small.

Alternatively, we could have phrased our constraint by simply indicating that each name must be associated with at most one book. The relational join of `owners.n` below matches the right hand elements of the `owners` tuple (`Book -> Name`) with n (`Name`) to produce a set of `Book` instances. We then use `lone` to express that there should be at most one book per name.

```
fact noSharing2 { all n: Name | lone owners.n }
```

As models get larger, the default graphical visualisation of witnesses and counterexamples becomes difficult to interpret. To counter this, counterexamples and witnesses from the Backbone specification can be imported into the Evolve modelling environment and viewed as composite structure diagrams from each relevant perspective.

Appendix E

Obtaining the Software and Example Models

The following location allows files and software relating to this thesis to be downloaded.

```
http://www.tanarc.com/phd
```

The following downloads are available.

- An electronic copy of this thesis.

- The Evolve modelling tool and Backbone runtime platform.

- A number of the example models used in this thesis, along with related source code where appropriate.

- The formal specification.

The site may be password protected to prevent search engine indexing. If prompted for a username and password, enter `phd` for both.

Further instructions for using Evolve to execute the example applications are contained on the site. Evolve is compatible with any platform that has a recent Java runtime. JRE1.6 or greater is preferred, although the software also works with JRE1.5.