Backbone: An Architectual Approach for Extensible Applications

Andrew McVeigh

Transfer Report
Supervisors: Professor J. Magee and Professor J. Kramer

July 12, 2007

Acknowledgements

I would like to thank my supervisors, Professor Jeff Magee and Professor Jeff Kramer for their advice and deep insights into the world of component structures and behavioural analysis.

Contents

Chapter 1

Introduction

1.1 Creating Extensible Applications

It has become increasingly common for applications to be designed so that they can be extended by third parties. The aim is to allow developers to add features to the base application, without further involvement from the original creators of the application. The application then forms an extensible platform on which a family of applications can be built, servicing a wider audience than would otherwise be possible.

Eclipse [42], Excel [68] and Firefox [13] are three prominent examples of this approach. They all support extension by different mechanisms, but the basic concept is the same: they allow third party developers to add functionality to the base application. These application have each spawned a significant set of extensions, and in some cases have created a sizeable commercial marketplace for third-party add-ons. The concept has shown itself to be of significant value to the original application creators, extension developers and end users.

Unfortunately, the approaches underlying extensible applications place unwelcome limitations on the types of extensions possible. Some approaches supply a monolithic base application and pre-planned extension points that limit the functionality that can be added. 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 extension developers.

There is a further problem. Extensible applications naturally lead to the desire to combine two or more independently-developed extensions into a single product. However, as extension developers are given greater flexibility to adapt the base application, the potential for multiple extensions to conflict when they are combined goes up proportionately. This limits the value of the approach.

Backbone is presented as an architectural approach which addresses or ameliorates many of the issues outlined above. It has its heritage is in the architectural component models of Darwin [53] and UML2 [76], allowing complex architectures to be described, managed and reasoned about. It contains a small number of additional constructs which ease the tension between architectural management, freedom of extension and the potential for conflict between combined extensions [58, 59].

1.2 Example Scenario

The AudioSoft company produces a studio desk application for controlling digital audio devices via a software mixer. This software is used by radio studios to control a set of devices for on-air transmission. Unwilling to add a software controller for every device on the market, AudioSoft develops an extensible base application so that other device controllers can be added by third party developers. The company is not prepared to release the source code for the product due to several proprietary algorithms used in the mixer. The application is complex, and contains many components.

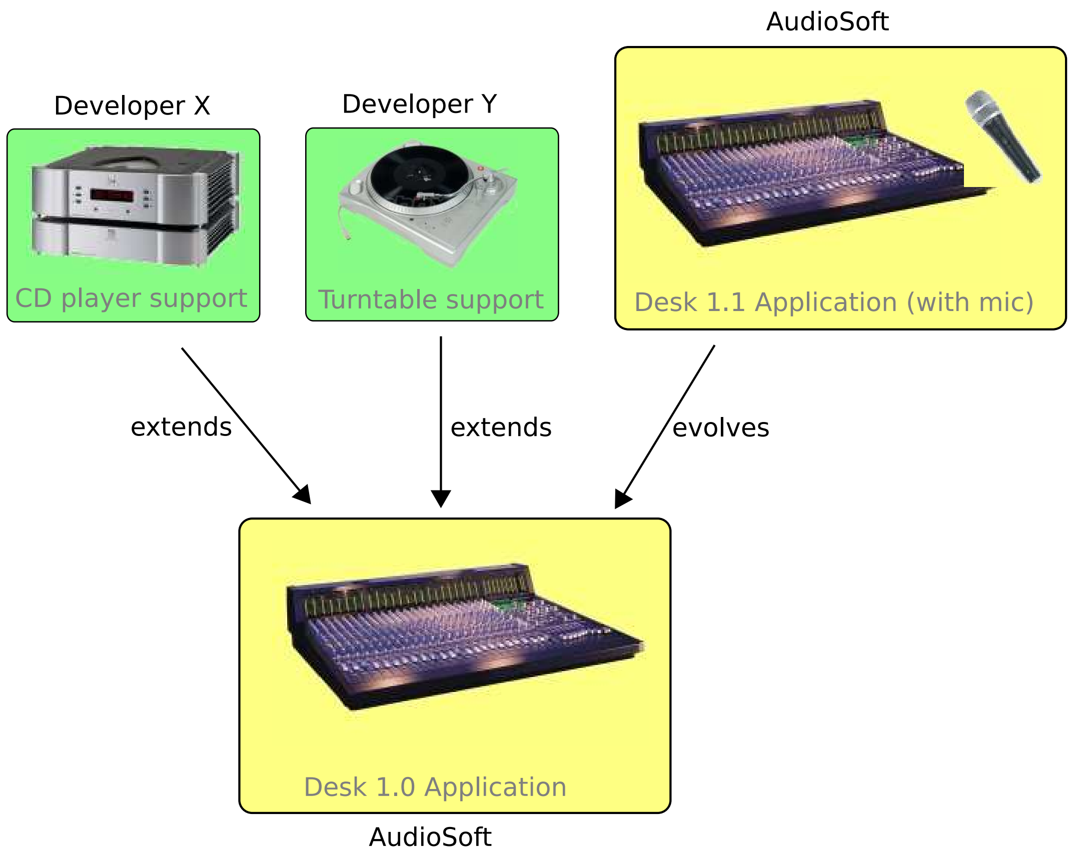


Figure 1.1: The desk application along with various extensions

Developer X is a third party developer contracted by a radio station to add support for a particular make of CD player to version 1.0 of the desk application. However, there is a problem. The CD player supports cueing, where the audio is sent to an off-air audio bus so that the start of a track can be found. The desk does not support cueing. As a consequence, X must adjust the base application to add cue support before integrating in the CD player controller.

Other developers are independently tasked with creating software controllers for a variety of other devices such as turntables, and these are separately incorporated into the 1.0 desk component as well.

In the meantime, AudioSoft upgrades the desk application to version 1.1 and add support for a microphone device. Unlike the CD player, the microphone does not require cue facilities. However, it does require support to monitor audio levels from the output, requiring a mixer upgrade.

Finally, a radio station using desk 1.0 wishes to upgrade to version 1.1 and use the CD player and all other device controllers in a single desk. The situation is shown in figure 1.1.

1.3 Requirements for Approach

Analysing the example scenario gives the following requirements for an approach to creating extensible applications:

1. MANAGE
   It should be possible to define and manage a complex architecture using the approach. Extension developers should be shielded from the full complexity of the underlying application, and only exposed to the relevant parts of the architecture at the appropriate abstraction level.
2. EXTEND
   It should be possible to extend an application to accommodate new features. The changes to the base application may be unplanned, and therefore the ability to make arbitrary changes is required. In addition, the approach must work even if the implementation source code for the application is not available. Ideally the extension developer is afforded the same power to amend the architecture as the developers of the base application.
3. NO_IMPACT
   The changes made by one extension should constrain the ongoing development of the application, or the development of any other extensions. An extension should not place the maintenance burden for any part of the added features onto the developers of the base application.
4. COMBINE
   It should be possible to combine multiple, independently developed extensions to a base application, so as to form a unified architecture.
5. UPGRADE
   It should be possible to combine extensions with an upgraded version of the application.
6. VERIFY_AND_REPAIR
   It should be possible to verify that the system has the correct structure and behaviour after combining extensions and upgrades. Any conflicts should be detected and should be reparable, without violating the other requirements.

Some of these requirements appear to be in direct conflict. For instance, EXTEND indicates that arbitrary changes must be possible implying a fine grained-decomposition of features which can be removed or substituted, whilst the application architecture must stay manageable and expressed at the appropriate level of abstraction (MANAGE). Backbone addresses this issue by providing a hierarchical component model with full support for composition and encapsulation.

Further, EXTEND implies that the application architecture can be modified, but NO_IMPACT says that this must not be visible to or constrain the base application developers. Backbone ameliorates this tension by allowing alterations to components to be expressed as deltas which are applied at runtime. A set of well-formedness rules pick up any conflicts, which can further be repaired by applying other deltas (VERIFY_AND_REPAIR). The deltas are packaged as components, giving an intuitive approach to the development of extensions.

Future work will allow components to express a behavioural protocol which will be translated into FSP [55]. Protocols will be composed reflecting the structural composition of components and safety and liveness properties will be checked to ensure that the behaviour of the system is correct.

1.4 Thesis Structure

This thesis presents the Backbone component model and discusses why it is suitable for developing and extending extensible applications. The core constructs of resemblance and redefinition are shown to address (or ameliorate) the extensibility requirements in the previous section.

The remainder of this thesis is structured as follows. Chapter 2 examines existing work in the area of extensible applications and architectures and examines how they match up against the requirements outlined.

Chapter 3 presents an expanded form of the example scenario and shows how it can be modeled using Backbone, focusing on the structural aspects and conflict resolution. How Backbone addresses the requirements is considered in the context of the example.

Appendix A presents an informal overview of all the Backbone structural constructs. Appendix B elaborates on this by presenting the highlights of the Backbone formal structure specification in Alloy. The approach for the translation of human-readable names to UUIDs is also described, allowing independently developed extensions to avoid introducing identical identities for separate components. This is an important part of the approach.

Chapter 2

Background

This chapter presents a survey of existing work that is relevant to the creation of extensible applications, focusing particularly on a component-based viewpoint. Various approaches to solving the problem of combining independently developed components are reviewed.

2.1 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 [57]. 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 parameterised components is 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 parameterised software components is proposed. The proposal outlines options such as space-versus-time tradeoffs 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 parameterisation 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.

In essence, the wider vision remains largely unfulfilled and surprisingly elusive. Fundamental issues remain where even minor syntactic mismatches prevent component integration and reuse [37].

In [20], software engineering is compared to the production of muskets in the pre-industrial era, where hand crafting of parts was routine. The lack of interchangeability and standard methods of measuring component compliance to specifications are cited as being key impediments to the development of component catalogues.

2.2 Component Technologies

A number of component technologies exist, each with unique characteristics. The various technology models are examined with a focus on supporting the specification, reuse and integration of existing components.

2.2.1 CORBA and the CORBA Component Model (CCM)

CORBA [75] provides a platform-independent communications format and a 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 via interfaces or turned into a textual form. Bindings between components are formed by passing references, and are not specified using connectors. A hierarchical model is not supported. A CORBA marketplace for reusable components did not occur, largely because the approach was focused on the integration of components implemented for heterogeneous platforms.

The CCM is a component model for CORBA [77]. 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 [81]. There is a considerable amount of similarity between the CCM and the EJB specification (section 2.2.2).

2.2.2 Java Component Models

JavaBeans [70] is the client-side technology for the creation of reusable Java components. It is primarily focused on GUI components, and supports an event model where clients are notified of changes to state attributes. Connections are modeled as references from a JavaBean to clients interested in changes in the underlying state model. This supports a limited form of hierarchical composition through object aggregation. It does not support explicit service interfaces.

Enterprise JavaBeans (EJB) is the distributed server-side Java component model [69]. Connections are formed by using a registry called JNDI (Java Naming and Directory Interface) to locate named components. Component hierarchy is not supported. As an alternative to EJB, so-called lightweight frameworks have become popular in recent years. The Spring framework [82] uses XML configuration files to explicitly connect interfaces between Java classes. This is known as dependency injection [32], and is analogous to a simple, non-hierarchical ADL with configuration programming expressed via XML. Spring also contains aspect-oriented programming facilities.

2.2.3 COM (Component Object Model)

COM is a component model and infrastructure built into the Windows operating system [9]. It forms a common extension mechanism for many applications, including the Office suite applications such as Excel [68]. 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.

The COM model does not focus on supporting or resolving extension conflicts, and the obscure and implicit nature of the configuration (via the registry) makes models difficult to evolve architecturally. Multiple versions of a COM component are supported, although this leads to the troublesome case known as "DLL Hell" [25]..

COM and its successor ActiveX [67] have successfully produced a marketplace for Windows GUI components.

2.2.4 Open Services Gateway Initiative

The Open Services Gateway Initiative (OSGi) defines a component-based environment for the deployment of network-enabled services [5]. Facilities are provided to manage the deployment and life-cycle of components. Components can either be bundles (analogous to plugins) or services, and these may be updated or removed without restarting the system. Bundles may depend on and also provide interfaces, although OSGi has not provided any form of component connection until very recently [28, 5].

Eclipse [41] 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. It also offers the ability to specify plugins using the OSGi declarative service facility [21].

OSGi provides a general model of extension, where a plugin must explicitly indicate its points of extension. These correspond loosely to provided services that can support many connections. Other plugins indicate which services they require, and the providers and requirers are matched up through some form of registry. It is interesting to note that an extension point can be more than just a provided service, and arbitrary meta-data can be associated with each extension point and each plugin.

The OSGi model does not offer support for true component composition, and any architectural hierarchy must be inferred through the logical names of the bundles which are named according to Java package conventions. This tends to bias the developers of a large architecture towards more coarse grained components, increasing the footprint of potential conflicts when components must be replaced.

2.3 Architecture Description Languages

Architecture description languages (ADLs) evolved out of the desire to explicitly describe the architecture of a software system as a set of connected components. Many ADLs exist, and the general consensus is that they must support composition of components, explicit connectors and provide an underlying formal model which can be analysed [63]. Components must specify all required and provided interfaces. Most ADLs also use a hierarchical component model.

2.3.1 Darwin

Darwin [54, 50] is an ADL used for specifying the architecture of distributed systems. It offers support for multiple views and allows information to be added to the architectural specification in order to aid analysis. The separation of the structural description from the implementation is often referred to as a configuration-based approach. Behavioural information is specified using finite state processes (FSP) [55], a process calculus. The Labeled Transition System Analyser (LTSA) tool can process FSP into labeled transition systems, allowing deadlock checking, the checking of safety and progress properties, and other types of analysis to be performed.

Although initially developed independently, the core Backbone component model is almost identical to that of Darwin. This presumably reflects the influence that Darwin and other ADLs have had on the UML2 component model (section 2.4.1) upon which Backbone is based.

Darwin supports a form of port type inference, as well as type parameterisation [24].

Dynamic reconfiguration of architectures specified in Darwin is explored in [49]. This work considers how to accommodate evolutionary change to the structure of an architecture, whilst the system is running. Configuration changes are modeled 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 component to the new one. Backbone allows a similar set of deltas to be specified against an architecture, but packages these as component definitions. Backbone also focuses on statically representing the evolution of a system, and the interpreter cannot dynamically apply extensions.

2.3.2 Koala

Koala is an ADL designed to express the architecture of software in embedded electronic devices [94]. It is loosely a variant of Darwin, but incorporates features designed to tailor it to the embedded domain and reduce the size and overhead of a running application. Further, it provides facilities to describe product families to handle the diversity of component selections in a consumer electronics product line. See section 2.5.3 for a discussion on how Koala handles extensibility.

2.3.3 C2 Family

C2 [90] is a non-hierarchical ADL designed to support the explicit requirements of GUI software, including the reuse of GUI elements. A component has a top and bottom domain. The top domain specifies the notifications that the component can accept and the requests that are issued to the rest of the architecture. In essence, the domain concept models the notions of provided interfaces for the handling of notifications, and required interfaces for issuing requests.

The bottom domain indicates the notifications that will be emitted, essentially modeling 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. Pictured visually, 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 [60] is a variant of C2 designed to support dynamic instantiation of components. It also supports upgrading components in a running system and removing unwanted components via reconfiguration. C2 SADL defines the concept of *placeholder* components [61], 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. Backbone has incorporated this concept into its approach.

Later work in this area developed C2 SADEL [62] which explicitly provides support for analysing the architectural evolution of a system. The protocol of a component is modeled via state, invariants, and pre and post-conditions. Further, this approach is used to implement the design environment and language, demonstrating its applicability.

The C2, C2 SADL and C2 SADEL models do not deal with hierarchical components, or the explicit analysis of architectural issues that arise from independent extensions.

2.3.4 ROOM

ROOM [85] is an ADL for modeling and constructing real-time software systems. Components are called actors, and must specify any required or provided interfaces via ports. Actors may be hierarchical and active. Actor 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 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 resolution are not described in the ROOM literature.

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. It has had a far reaching impact on graphical modeling techniques, and is one of the key influences that led to the vision of Model Driven Architecture (MDA) [78]. The ROOM approach and similar techniques have been successfully used in the real-time software arena for many years.

2.3.5 Balboa

Balboa is a component model and environment that is focused on providing a convenient and efficient compositional design environment for system-level application architectures [23]. Balboa supports type inference, and a developer may choose to be explicit about the “input” and “output” types of a component, or defer this to the type inference algorithm. This allows an architect to remain oblivious to the C++ implementation types which form the implementation strategy. Balboa contains incremental type inferencing algorithms, but features an NP-complete model. In contrast, Backbone places constraints relating to the specification of leaf components that reduce the problem down to a simple algorithm.

Balboa indicates links in leaf components through a textual syntax and parameterisation. Backbone has no type parameterisation, and uses explicit links instead (section 3.3.5).

2.4 Graphical Component Modeling

2.4.1 The Unified Modeling Language 2.0

The Unified Modeling Language (UML) is a graphical language for describing the structure and behaviour of object-oriented systems [76]. 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 [33]. The design of these diagram types have been heavily influenced by Darwin, ROOM and several other ADLs. Composite structure diagrams can be used to model composite and leaf components. Component diagrams are also provided, but these are essentially a syntactic variant of composite structure diagrams and are not considered further here.

Component protocols in UML2 are modeled either as sequence diagrams [84] or activity diagrams, with the latter seemingly preferred. Essentially, this amounts to modeling 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 introduces package merge which provides a way to specify extensions to a package via other packages. As pointed out in [97], 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. Further, there is currently no precise definition of package merge available.

UML2 represents an amalgam of loosely integrated techniques and graphical diagrams from many areas, showing its heritage as standard designed by a committee. The meta-model 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. This is the approach that Backbone has taken for component modeling.

2.4.2 Software Architect's Assistant (SAA)

SAA is a graphical design environment for the development of Darwin programs [71, 72]. 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 conventional 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.

jUMbLe is the Backbone graphical CASE tool. jUMbLe is related to Backbone in the same way that SAA is related to Darwin. SAA does not focus particularly on the creation of extensible systems. Backbone currently does not focus on the development of distributed systems, although this is an area of potential further work.

2.5 Approaches to Extensible Systems and Component Reuse

2.5.1 Scripting

In an influential article by Ousterhout [79], 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 also be used to produce systems which are extensible via customisation of scripts.

Many systems have opted for a scripting approach in order to provide an extensible base which supports the introduction of new features by extension developers. A prominent example of a platform with an important scripting focus is Excel [68] which uses Visual Basic for Applications as the scripting language. The scripts can call out to COM components, which can be added to the application.

Firefox uses plugins which can consist of Javascript scripts and optional implementation-level components. These scripts is used to extend the user interface and deal with any logic, and can call into the underlying application.

Scripting approaches do not address the problems to do with combining extensions, although they tend to minimise it as extensions are only given limited freedom to adjust or modify the base application. This contravenes the EXTEND requirement.

2.5.2 Frameworks

Frameworks are a reuse technique allowing a semi-complete application to be customised and extended into a complete application [29]. A framework allows objects implementing particular interfaces to be registered with it, and will invoke these objects at various points in its processing. This is known as inversion of control [45]. This offers an effective and large-scale form of reuse, but in practise a number of fundamental issues limit this approach [17]. Many of these issues stem from ownership of architectural changes, where changes to the framework must be performed by the framework developers, or else a copy of the framework's code must be copied and independently modified leading to subsequent maintenance problems. In addition, evolving a framework is difficult because the base framework must be aware of the different uses that it is being put to before it can be reliably updated. This is a violation of the NO_IMPACT requirement. Various techniques have been proposed to mitigate this situation [19, 38, 56].

In addition, frameworks can only be extended in pre-planned ways, based on variation points encoded as required interfaces. This contravenes the EXTEND requirement.

Frameworks are able to be extended without having the implementation source code, however. Many commercial frameworks are delivered in this way. As long as the specification of the interfaces is available, the framework can be delivered in binary object form.

2.5.3 Parameterisation and Product Diversity

Koala [93] is a component model based on Darwin that allows for variation in an architecture through variation points and parameterisation. Component variants can be plugged into the variation points, supporting a family of applications. The points must be decided in advance and planned into the architecture, limiting this to a technique for planned extension. This does not fully satisfy the EXTEND requirement.

Parameterisation is used in Koala to capture options supported by a component [93]. This approach only supports planned variation and can also result in a combinatorial explosion of options if the parameters of the constituent parts of a composite are also exposed.

Koala features HORCOM, a software bus which mirrors the functionality of an electronic hardware bus [95]. Components are decoupled from each other by HORCOM, and understand a standard protocol. Additional components can be plugged into the bus, extending the behaviour of the system in an additive way.

2.5.4 Plugin Architectures

Plugin architectures have one or more frameworks at the core of their architecture, but are generally packaged as an application [42]. Plugins represent components that can be added to the platform by adjusting the system configuration. Large extensions to these systems represent platforms in their own right, and can literally consist of hundreds of plugins. Plugin architectures support a constrained form of component-based extension architecture and have problems with the combination of independent extensions [7, 22].

Eclipse supports a plugin model, based on OSGi [4] where components (OSGi bundles) indicate how they are connected to the service provisions of other components via a set of manifest files [18]. The model is non-hierarchical because it does not support composition of other component instances in an architectural model. Modifying the component connections in an existing application can involve a considerable duplication of manifest entries and is not practical for large applications (MANAGE, EXTEND). Structural conflict is a problem in this model, although in practice it is restricted because replacing individual components in a large configuration is difficult (VERIFY_REPAIR).

2.5.5 Aspect-Oriented Programming

Aspect-oriented programming [48] aims to allow orthogonal features to be developed and woven together at a later stage. Aspects are specified orthogonally to the main source code base.

The main issue with this approach is that although it works well with truly separate concerns such as transactions and logging, it does not provide a convincing approach to handling features which are designed to interact. In addition, there has been little take up of this approach for the specification of general program logic. Spring [35, 82] and other Java enterprise approaches have stared using aspects for transactional demarcation and security concerns.

Aspects are limited in how they can affect the base application (EXTEND). Combining multiple, independently developed aspects is also often problematic (COMBINE, VERIFY_REPAIR).

2.5.6 Viewpoint-Oriented Approach

Viewpoints allow multiple perspectives on a single system to be constructed and combined at a later point [30]. This allows 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, function, architecture and also 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 removal of features from a system or the modification of existing features.

A more structurally-oriented approach is taken in [26]. In this model, views also encode partial knowledge about a system, based on a reference model. In practice, views are partial specifications of instance graphs. In the Backbone approach, views would be represented as redefinitions on the same underlying component configuration.

2.5.7 Architecturally-Aware CM Systems

Conventional CM (configuration management) systems have been successfully used to manage and evolve component architectures [86], and offer a lowest-common denominator approach to extending a system. Combining structurally conflicting extensions 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 behaviourally interfere. An approach to mitigating the limitations of this is presented in [15]. Currently the work is focused on creating architectural deltas in order to structurally merge extensions to a product line architecture. No support is offered for detecting or resolving behavioural interference, although this is mentioned in future work.

The introduction of variation points and the general evolution of architectures has been made more feasible through systems like Mae which have integrated CM and architectural concepts [92, 83]. This approach provides an overarching CM system which understands architectural and evolutionary concepts and can support the creation of variants. This approach assumes that all components are available via a unified and consistent CM system, which is not feasible in an environment with many (possibly commercial) component providers. Further this does not solve the need to create many variation points to satisfy all extension developers, leading to a complex, very generic architecture which violates the NO_IMPACT requirement. Backbone avoids this by effectively providing some of the facilities of a decentralised CM system where the base application need not be aware in any way of the architectural changes made by extensions.

Mae provides a powerful unified architectural and CM approach. Backbone in contrast provides a unified modeling foundation with explicit modeling constructs for architectural definition and evolution. Backbone models are expected to be version controlled using a conventional CM system, reflecting the practical constraints of projects in an industrial setting.

2.5.8 Mixins

Mixins are a language feature for expressing abstract subclasses that can be reused in different parent classes [10]. Any number of mixins can be combined into a parent class, and methods of the mixin may invoke methods of the that class. This implies that the mixin must make assumptions about the names of the parent methods that it will call, which represents an integration issue. Further, multiple mixins may conflict or interact in unforeseen ways (COMBINE, VERIFY_AND_REPAIR).

Scala is a language with mixins which aims to support the combination of independently developed extensions [96]. It provides two dimensions of extension: data extension for the object-oriented view, and behavioural extension for the functional view. Scala does not offer support to alleviate the name collision problem suffered by mixin and multiple inheritance approaches. Further, the combination of independent extensions must in part be performed by manual adjustment of the class that combines the mixins.

2.5.9 Virtual Classes and Nested Inheritance

The BETA language allows virtual patterns, which are a type of virtual class [51]. A type can specify other types, and sub-types can refine those types through inheritance.

Nested inheritance provides a similar facility, which allows an inheriting type to override any contained types of its sub-type [74]. This extends the BETA facilities by guaranteeing type safety through static analysis. The example scenario of a parsing system demonstrates considerable reuse for a compiler family.

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

2.5.10 Difference-Based Modules

MixJuice adds a module system to Java, supporting a variant of redefinition where modules describe the difference between the base application and the desired application [40, 16]. Inheritance is used in place of resemblance.

The intention is to allow an object-oriented system written in Java to be extended in unplanned ways. Resemblance is more expressive than inheritance, however, as it allows constituents to also be deleted and replaced, reflecting their evolution. 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. Because it relies on the Java class model, MixJuice also does not offer a true architectural approach.

A graphical approach to depicting MixJuice architectures has been proposed [39]. Backbone is similar to a hierarchical, component-oriented version of MixJuice in several respects. Backbone additionally offers support for predictably detecting and resolving structural and behavioural conflicts, which are troublesome in MixJuice.

MixJuice uses a naming system of module::class to prevent accidental name collisions in independently developed modules. jUMbLe (the Backbone graphical modeler) uses a related, albeit more powerful, system of assigning globally unique identifiers and mapping onto human names in the graphical CASE tool (section B.1.1). This has the advantage of supporting refactoring of human-readable names without perturbing the underlying identity of a component. Renaming can occur without disturbing other independent extensions that refer to the same underlying component.

2.5.11 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 [27]. The development of components is driven by a hierarchical feature model which contains 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 variants.

Concurrent evolution of single components of the product line is problematic [87], and work to merge and propagate changes from one branch to another is still required if the code for a component is branched [15] (COMBINE). In addition, this approach usually involves maintaining a single feature graph for all product variants, which makes decentralised development difficult (NO_IMPACT).

GenVoca is an approach and environment for generating product lines [6]. The primitive element is of composition is called a *gluon* and these are arranged in layer-like structures called *constants*. GenVoca features a form of redefinition called class extension, which is a subclass that assumes the name of its parent class. This construct is closest to a combination of resemblance and redefinition, allowing for a form of incremental update. The implementation relies on an aspect-oriented approach for the combination of features.

2.6 Combining and Integrating Independently Developed Extensions

2.6.1 Component Integration Issues

The problems involved with integrating independently developed components is well known, and the issues range from simple syntactic mismatches through to more complex behavioural interactions and testability concerns [37, 47, 88, 86]. Component versioning and deployment approaches have been proposed as a way of mitigating this [64], 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 version, or the problem when a single version of the component must always be enforced. The latter situation often occurs when a component is managing a resource that requires a single controlling entity in a system.

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 [37] 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 an implementation level, where maintaining compatibility with existing languages and paradigms is either implicitly or explicitly considered to be of paramount importance. Superimposition [8] 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 it 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 black-box reuse technique which cannot adjust fundamental characteristics of a component, only hide them (EXTEND). Other wrapping approaches are outlined in [46, 91].

2.6.2 Feature Composition

Feature composition has been proposed to allow the addition of independently developed features to a telecommunications system, where existing services must not be affected in an adverse way [36]. 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.

Feature interaction is detected at runtime by examining whether a higher priority feature has caused the invariants of a lower priority feature to be invalidated. A lower priority feature is not allowed to violate the invariants of a higher priority feature. This work does not explicitly mention component structures or connections, and is not directly applicable to a hierarchical component-based architecture. Furthermore, the work is focused on finding the best possible run-time resolution of conflicting requirements rather than supporting a developer in understanding and resolving unwanted interference. The steps for expressing and resolving interactions have to some extent been mirrored in the Backbone approach, with some modifications required for the design-time focus and hierarchical component model.

2.6.3 Architectural Merging

An approach to merging UML models is presented in [3]. This considers only the structure of the models, and does not address behavioural issues.

Product line architecture, as discussed earlier, have to deal with merging issues 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 [15], which works by determining the delta changes between different parts of the product line and applying these changes elsewhere. This work is focused on the architectural level, but currently offers no support for detecting behavioural conflicts or subtle structural issues. In contrast, Backbone expresses component change in terms of deltas. These do not need to be synthesised.

As stated previously in section 2.5.7, Mae [83] is an architecturally aware CM system that understands architectural hierarchy and records delta changes to architectures. As such, architecture merging is fully supported.

2.6.4 Detecting Compositional Conflicts

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 [65]. A critical pair provides a formal foundation for a minimal example of independent, conflicting changes. The technique is currently restricted to a fixed set of refactorings, and relies on having changes expressed in graph form.

An ontology based framework for classifying compositional conflicts has been implemented [52]. This is intended to serve as a basis for component adaptation and reuse. Using this framework, the behavioural and structural properties of a set of components can be specified and analysed for conflict.

2.7 Behavioural Modeling of Components

This section presents a brief survey of the field of behavioural modeling of components. This survey does not intend to be exhaustive, instead referring only to work which directly reflects the approach used to specify the behaviour of Backbone components.

2.7.1 Protocols

The formalisation of object protocols is considered in [73], which argues that it is essential to view the protocol of an object as a process rather than as a function. The notion of a service type is introduced, which characterises the traces that an object can produce in terms of operation actions. This is assumed to conform to a regular language, such as that produced by a labeled transition system. The service type is contrasted with the notion of a value type, which summarises the argument types and return type of a method.

The notion of request substitutability is considered, in terms of whether one object can be substituted for another and a client not be able to detect the difference. It is shown that correct substitutability is closely related to the concept of request failures. Basically, any request which is emitted from a client, but can not be accepted by the service at the current point in time will cause a failure, generating a transition to the error state.

Component protocols are modeled with regular expressions in [80]. The interaction context is the hierarchical SOFA model, where connectors join provided and required interfaces of components. Each event in the protocol language is either a request or response, and can be either emitted or absorbed. This corresponds closely to the modeling of client-service protocols presented in an example in [55], which uses either request or reply and either call or accept as action prefixes.

A set of operators are supported over the regular expressions used to model protocols. These are sequencing, alternative, repetition, and-parallel, or-parallel and restriction (or hiding in FSP terminology). Two composition operators are introduced: composition which models parallel execution of two substrings of the language, and adjustment which insists that tokens between the two substrings occur in the same order. Substitution of protocols is considered, and the concept of bounding a protocol is used to determine if one protocol can be safely substituted for another.

In [2], the composition of protocols of the internal parts of a composite component is considered. Three different classes of protocol composition error are described. Bad activity errors occur when a client component emits a call to a service component which cannot be accepted in the service's current state. The consent operator is developed, which is able to detect these types of bad sequences. This is contrasted to the CSP parallel composition operator, which automatically synchronises actions between the client and service components thereby avoiding such errors. The two other errors types are no activity (where no component can emit or absorb a request) and divergence (where at least one component cannot stop to accept a version update). The results are applied to a system to determine when it is allowable to omit a binding for a component port [1].

2.7.2 Predictable Plugin Behaviour

A model of a plugin behaviour has previously been presented in [14], using a process formalism to ensure that properties hold when plugins are combined. This work does not deal with hierarchical component structures. It handles the situation where multiple components share a single service at a level of abstraction that assumes mutual exclusion of resources and services.

The underlying assumption in this model is that the events of a client and service component will be regarded as shared actions if possible. This prevents a client from emitting an action if the service is not prepared to accept the response, limiting the set of errors that can be detected to no activity errors. These types of errors are manifested as deadlocks when performing an LTSA analysis of the composed protocols.

2.8 Summary

The underlying theory and practice of building extensible applications suffer from the inherent tension between the manageability of an architecture (MANAGE), the freedom given to modify the underlying application (EXTEND) and the potential for conflict when combining multiple, independently developed extensions (COMBINE, VERIFY_AND_REPAIR). The issues involved are a superset of the issues involved with component reuse and integration, when seen from an architectural perspective. An application in such an approach is simply a complex composite component.

The vision of building a system quickly from a set of pre-fabricated components remains surprisingly elusive.

Chapter 3

Using Backbone to Define, Extend and Evolve an Architecture

Backbone is an ADL for creating component-based, extensible applications.

The aim of this chapter is to explain the constructs of Backbone and how these address the requirements presented in section 1.3. We do this through a heavily simplified, but realistic, example based on the architecture of a commercially available application. The author was one of the architects of the original application.

The requirements for developing extensible applications represent a superset of the requirements for component reuse in a component-based system. This is because an application in such an approach is phrased as a complex, composite component. The techniques are equally applicable, and as such Backbone offers facilities for component integration and reuse also by virtue of its focus.

The application is an audio desk that controls a number of digital audio devices. We start by defining the audio desk without any devices, and then proceed to extend the desk to add a CD player device. The desk component is then evolved via another extension to always include a microphone. The two extensions are shown to structurally conflict when combined, and the conflict is resolved through a further extension.

Appendices A and B contain a complete description of Backbone's structural concepts and rules.

3.1 Defining the Audio Desk Application

The AudioSoft company develops and sells an audio desk application.

An audio desk controls a number of digital audio devices which are connected to a mixer. For our example, we start by defining the mixer component, which accepts audio packets on the input port, adjusts the overall volume and routes the packets to both output1 and output2 (figure 3.1).

The mixer is a leaf component, which means that is cannot be decomposed further into other component instances. Each leaf must be directly associated with a Java implementation class which implements the logic. The volume is held as an attribute, which is set to a default value.

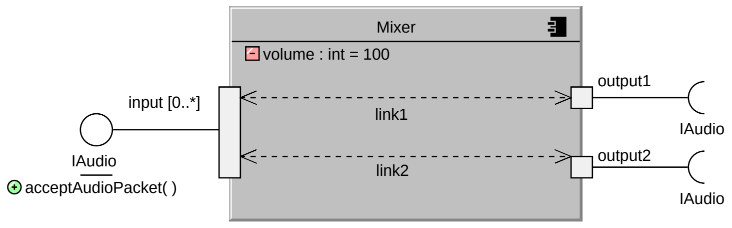


Figure 3.1: The mixer leaf component

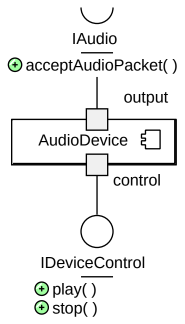


Figure 3.2: A placeholder device shows the "shape" of an audio device

Instead of defining a device component at this point, we will define a placeholder (figure 3.2). This is a dummy (composite) component which outlines the general shape that any device should have. A placeholder can be used to indicate something that must be concretely defined later, and is intended to be descriptive rather than prescriptive.

We can now define the desk component (figure 3.3). The desk is a composite component, consisting of an instance of the mixer component and an instance of the device placeholder. These instances are also known as *parts*. They are wired together using connectors, which connect ports.

A desk contains a mixer, and exposes its outputs. It contains devices which are connected between the indexed *input* and *deviceControl* ports. The latter is how the desk is controlled.

It is worth noting that the desk component cannot be instantiated as it contains a placeholder. This must be replaced, through resemblance or redefinition, before use.

The *[+]* (pronounced "take next") is used to connect to an indexed port, and will assign the next available index to the connector. In this case, the assigned index will be zero. The use of *[+]* prevents connectors from independently independent extensions from taking the same index.

A delegate connector is used to connect the *mixerInput* and *input* ports. This aliases two ports together, and is used to prevent the need for many indexed connectors between two indexed ports.

3.1.1 Port Type Inference

The function of port type inference is to automatically infer the interfaces of a port in a composite component. This minimises the changes required when later modifying components using resemblance or redefinition.

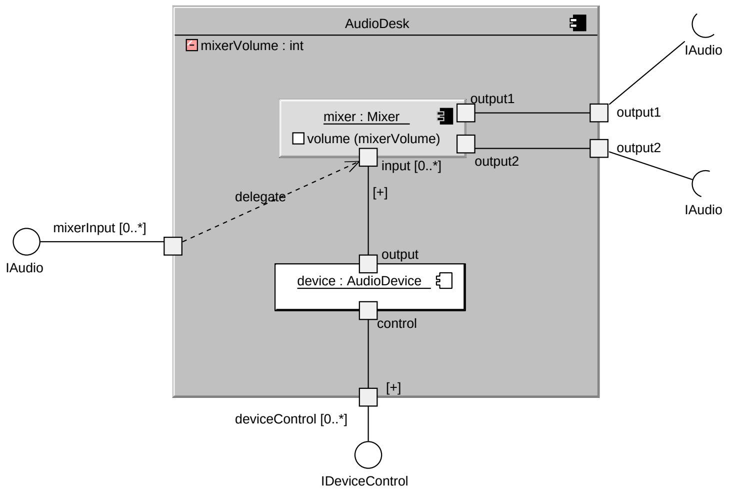


Figure 3.3: The desk composite has a mixer and a device part

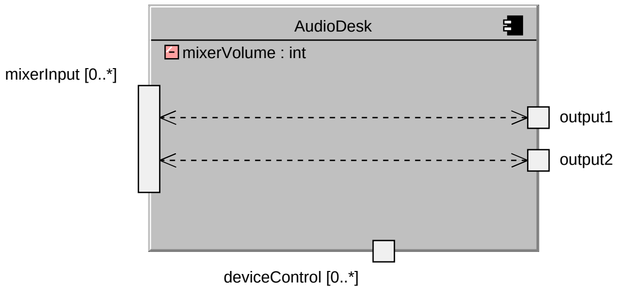


Figure 3.4: The inferred links of the desk component

The port links (*link1*, *link2*) in the leaf mixer component are used to propagate port type information if “more is provided than is required”. For instance, if an instance of mixer is connected so that a sub-interface of IAudio is provided to both *output1* and *output2* then this will result in *input* also providing the sub-interface – the type information has propagated via the links. This also works the other way – if input was connected to something that required the sub-interface, then this would propagate through so that the outputs also required that sub-interface. Port links reflect the internal connections of a leaf, as leaves have no explicit connectors.

The interfaces required and provided by a port (the port’s type) of a composite component can always be inferred. Whilst this is a trivial matter in the case of figure 3.3, the inference rules are useful when replacing parts of a component using the constructs provided for extension. The changes will automatically propagate such that the port types update also.

Composites cannot explicitly specify port links: the links for a composite can always be inferred from the connections between the component and the parts. To support an incremental approach, the port type inference algorithm will determine a set of inferred links for each composite, which indicate how any internal parts propagate type information between ports. The inferred links for the desk are shown in figure 3.4. This shows that the *mixerInput* and *output1* and *output2* ports are linked, reflecting the links from the mixer part that they connect via.

3.1.2 Slots and Aliasing

The mixer defines a volume attribute. An attribute is of primitive type (int, boolean etc) and provides a view on the internal configuration state of a component. An attribute may have a default value, such as in figure 3.1.

When we create a part, we can assign values to each of the attributes of the part's type. These are known as slots. When a slot has a value assigned, there are three possibilities:

1. A literal assignment.
   E.g. volume = 100. This will initialise the attribute with the literal value.
2. A copy assignment from the environment.
   E.g. volume = mixerVolume. This will copy the value of the attribute over from the parent component, but the attribute values can then diverge.
3. An attribute alias.
   E.g. volume(mixerVolume). This aliases the part attribute with an attribute from the parent component.

For the mixer part in figure 3.3, the third option is used which binds the two attributes together into a single one. *This allows us to propagate state from one or more parts into the parent component.* This has allowed us to encapsulate the mixer in the desk, but still allow the mixer's volume to be configured at the desk level.

3.1.3 Packaging the Application

We now have our base application. To package it into a single entity, we use a stratum which is a module-like construct based around the UML2 package concept. A stratum contains component and interface definitions. A stratum may not contain other strata, for reasons of simplicity rather than any technical necessity.

Each stratum must explicitly express its dependencies on other strata. These dependencies constrain the elements that are visible to the stratum's elements. In the case of our application there are no dependencies, as we package everything into a single stratum called Desk (figure 3.5).

3.1.4 Summary

Backbone's architectural approach is similar to that provided by Darwin [53]. It is based on the UML2 composite structure model [76]. Strata are not entirely dissimilar to modules, which serve to group elements in an application. The reasons for the existence of both strata and components are similar to the ones outlined for the coexistence of modules and classes in [11].

A hierarchy of components can be created through composition. This focus on hierarchy, composition and encapsulation allows large architectures to be defined, managed and reasoned about at the appropriate level of abstraction (MANAGE). This is one of the key aspects that differentiates Backbone over conventional plugin architectures [42].

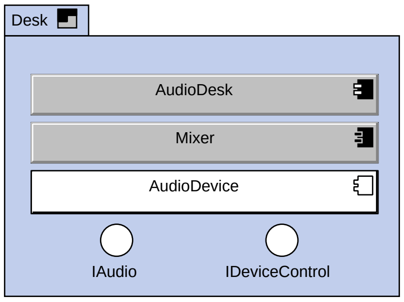


Figure 3.5: The desk stratum packages the application into a single entity

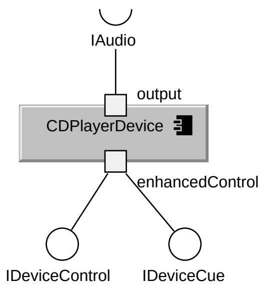


Figure 3.6: The CD player component

Backbone describes a composite component as a connected configuration of other component instances (parts). This captures more of the architecture than an object-oriented description of a system where the instantiations are contained within code. Further, the requirement that a component express both provided and required interfaces enforces a disciplined approach to dependencies. It is often the case when using an object oriented approach that classes have many code-defined dependencies on definitions of other classes, leading many to a complex and implicit tangle of dependencies [31].

Port type inference allows the interfaces provided and required by ports in a composite component to be inferred. As well as saving time, this features limits the changes required when altering parts of the application in order to extend it. Port links provide a way to propagate type information through the ports of a leaf, and are used to represent the internal connections. This is necessary because leaves do not have explicit connectors. This allows leaf components to be more reusable, as will be shown when the mixer is able to automatically adjust for a sub-interface of IAudio (section 3.3.5).

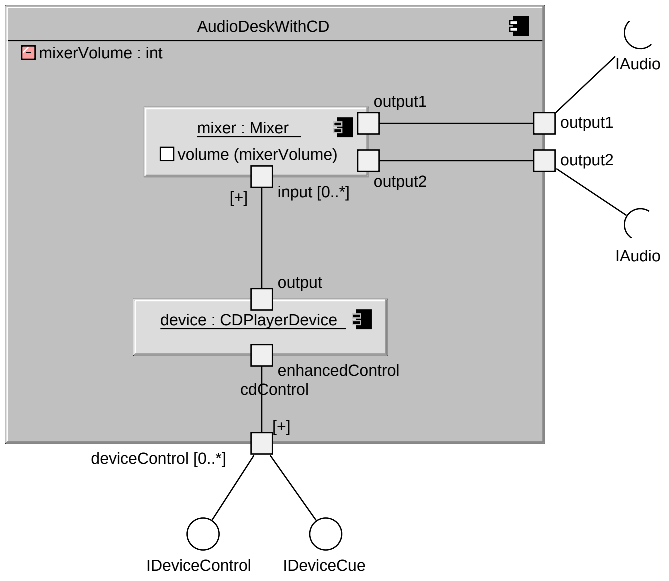
3.2 Extending the Application: Adding a CD Player

An independent developer, X, is asked by a radio station to extend the desk so that it can operate a CD player device. X must use the application as provided by AudioSoft.

The controller for the CD player device is shown in figure 3.6. Note that it does not conform exactly to the shape of the device placeholder component, as it provides an additional interface (IDeviceCue).

3.2.1 Resemblance

The developer decides to make an AudioDeskWithCD component which is similar to AudioDesk, but adds a CD player. To define this, the *resemblance* construct is used, which allows a new component to be defined in terms of delta changes to one or more existing components. Figure 3.7 shows the graphical view of the new component, along with the textual view underneath.



```
component AudioDeskWithCD
  resembles Desk.AudioDesk
{
  replace-parts:
    CDPlayerDevice device;
  delete-ports:
    mixerInput;
  delete-connectors:
    delegate;
}
```

Figure 3.7: Defining a new desk component in terms of differences to the original one

The placeholder device has been replaced by a CD player part, and the *mixerInput* port and *delegate* connector have been removed because they were not required by developer X. Port type inferencing has determined that the *deviceControl* port now provides both IDeviceControl and IDeviceCue.

Note that this is a distinct component from AudioDesk. However, since the new component is defined in terms of changes, if AudioDesk changes, then the new component will change also. This feature allows us to accept upgraded (UPGRADE) versions of AudioDesk in the future, possibly defined using redefinition (section 3.3.2).

3.2.2 Packaging the Extension

The two components are packaged in the DeskWithCD stratum. The components depend on definitions in the Desk stratum, and so we must explicitly indicate this using a dependency (figure 3.8). Another way to look at it is that the strata level dependencies constrain what the elements in a stratum can refer to. With the dependencies as shown, the elements in DeskWithCD can only refer to elements in that stratum, or the Desk stratum.

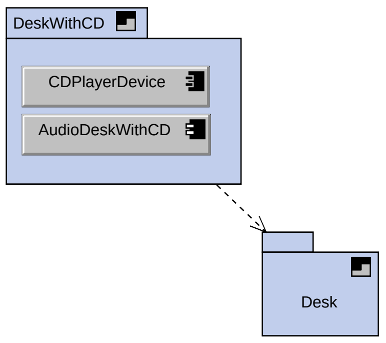


Figure 3.8: The DeskWithCD stratum depends on the Desk stratum

3.2.3 Summary

Using resemblance to define a new component in terms of another has several advantages over a completely new definition. It is economical and convenient – only the changes need be specified. In addition, if the resembled component changes at some point or is upgraded, then so does the new component's definition (UPGRADE).

For a composite component, changes can be addition, replacement or deletion or ports, parts, connectors and attributes. This gives the extension developer a large amount of flexibility to remodel the "insides" that it has inherited (EXTEND). This is in contrast to object-oriented inheritance where only additions and explicit overrides (limited replacement) can be performed [89].

Multiple resemblance is allowed, as long as the resemblance graph is acyclic (see section B.3).

Resemblance is not allowed for leaf components. This sidesteps the troublesome issue of a composite resembling a leaf and vice versa. To get around this restriction, in an actual system, leaf components are wrapped immediately in an "identity" composite, which is then used instead of the leaf (E.g. figure 3.9). The composite wrapper can then be redefined or resembled. The intention is to have this wrapping managed automatically by the graphical CASE tool supporting the Backbone approach, freeing the designer from this burden [58].

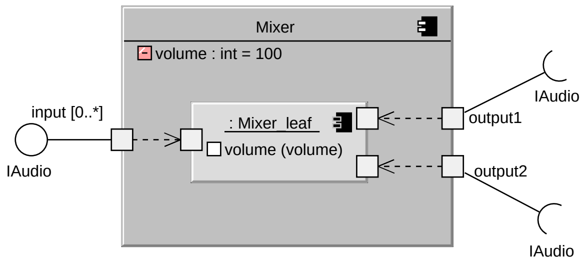


Figure 3.9: Leaves are always wrapped by "identity composites" in reality

Packaging a set of definitions as a stratum provides a useful, coarse-grained view of the dependency structure of a large application. Strata dependencies cannot be cyclic, and this focuses the developer on maintaining a strict layering of the architecture. A stratum can be understood in terms of its own elements and those of the strata that it depends upon. A single stratum can contain an arbitrary number of component and interface definitions, and an extension is usually packaged as a single stratum that depends on the strata defining the application that it is extending.

3.3 Evolving the Application: Adding a Microphone

AudioSoft decides to evolve the desk application to version 1.1, adding a microphone device as standard. AudioSoft has no knowledge of developer X or the changes made to support the CD player.

The microphone component is shown in figure 3.10. Of particular interest is that it requires the IExtendedAudio interface rather than just the IAudio interface. This is because the microphone must continually monitor the output (via getMaximumLevel) to ensure that its output does not exceed the maximum level.

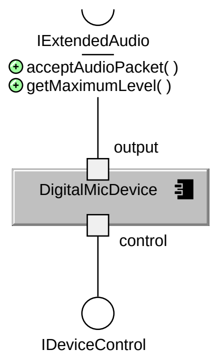


Figure 3.10: The microphone leaf component

3.3.1 Interface Resemblance

Figure 3.11 shows that IExtendedAudio is defined in terms of resemblance from IAudio. In this case, we have added a single operation (getMaximumLevel) and hence IExtendedAudio is a sub-interface of IAudio. It is also possible to delete and replace operations in the resembling definition, but doing this will break the subtype relation.

Inheritance is not used in Backbone. This concept has been subsumed by resemblance, which allows deletion and arbitrary replacement in addition to addition and overriding.

Interfaces and composite components can redefine and resemble elements of the same type. Multiple resemblance is supported in both cases as long as the resemblance graph is acyclic at all times.

3.3.2 Redefinition

Because they are evolving the platform, Audiosoft must change the desk component. However, they do not wish to force existing users of the previous version to upgrade. *Redefinition* can be used to

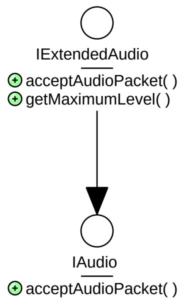
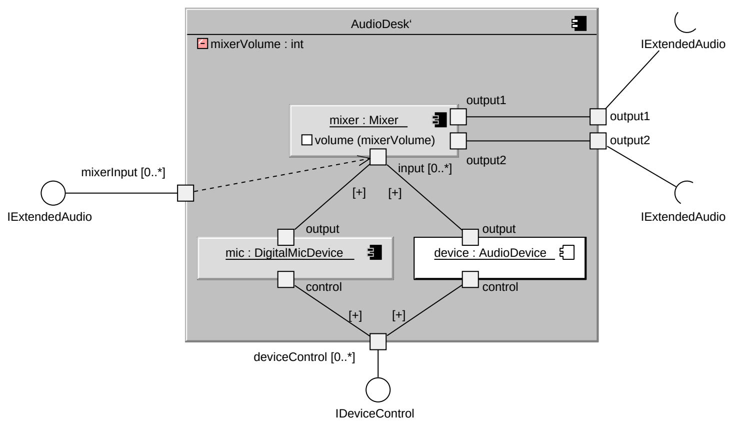


Figure 3.11: Resembling an interface and adding operations preserves the subtype relation

support this.

Redefinition allows a new component definition to be substituted for an existing one. The reason for using redefinition in this case is that AudioDesk is being evolved. Simply producing a new component definition with a microphone in it would mean that every product using the existing desk component would need to be upgraded.

In this case, resemblance and redefinition are used together to evolve the AudioDesk component to include a new device (figure 3.12). The microphone and two connectors have been added, and the original placeholder kept.



```
redefine-component AudioDesk
  resembles [previous] AudioDesk
{
  parts:
    DigitalMicDevice mic;
  connectors:
    mic1 joins control@mic to deviceControl[+];
    mic2 joins output@mic to input[+]@mixer;
}
```

Figure 3.12: Redefining the desk component to add a microphone.

It is important to note that the definition of *mixerInput*, *output1* and *output2* have automatically been upgraded by the port type inferencing rules. Because the output port of DigitalMicDevice requires IExtendedAudio, then this requirement propagates through the port links to ensure that *output1* and *output2* now require the sub-interface.

Similarly, the mixer *input* port must now provide IExtendedAudio. This is also exposed outside of the component via the *mixerInput* port.

The redefinition succeeded with only three additions because IExtendedAudio is a sub-interface of IAudio. It is possible for the mixer *input* port to provide IExtendedAudio, which satisfies both the microphone and placeholder device. If it were not a sub-interface, then it would still be possible to add the microphone, but it would require the mixer and device components to also be redefined to ensure compatibility with the new interface.

3.3.3 Packaging the Extension

The extension is packaged in the EvolvedDesk (version 1.1) stratum, as shown in figure 3.13. To allow the evolved definitions to reference the original definitions in the Desk stratum, a dependency must be present between the two strata.

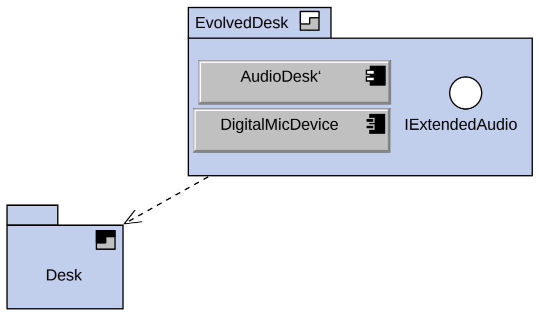


Figure 3.13: Packaging the evolved desk in a stratum

The EvolvedDesk stratum has a slightly different icon, indicating that it is *relaxed* rather than *strict*. A relaxed stratum will propagate its dependencies to any stratum which further depends on it. In effect, it allows other stratum to "see through it" to the stratum that it depends on. This can be used to support *strict* and *relaxed* styles of application layering, or a combination of the two [12].

3.3.4 The Overlapping Extensions Problem

The Backbone interpreter must be provided with a list of included strata in order to assemble a system. If we provide it with only the Desk stratum, then the AudioDesk component will be as defined in figure 3.3. If however, we provide it with the EvolvedDesk and Desk strata, then the AudioDesk component will be defined as in figure 3.12.

This is called stratum perspective. The desk from the perspective of the Desk stratum is the original definition (figure 3.3). From the perspective of EvolvedDesk, it is the redefined definition (figure 3.12). The *home stratum* of a component is the stratum in which it is defined.

The EvolvedDesk and DeskWithCD strata do not depend on each other. X has developed the CD player extension directly against desk rather than the evolved version. The way that the dependencies are defined, no element from DeskWithCD can refer to anything introduced in EvolvedDesk and vice versa. The two strata are said to be *mutually independent* (section B.5.1), and the dependencies imply that the strata were developed independently (without knowledge of each other).

However, both strata share a common dependency in the Desk stratum. The two strata are said to *overlap*. Mutually independent, overlapping strata are used to model architectural extensions which have been developed independently.

It is the case that mutually independent strata that overlap can cause conflicts when combined. This is because each stratum can make adjustments to the underlying overlapped strata that will conflict when used together. We call this potential for conflict the *overlapping extensions problem*.

3.3.5 Port Type Inference Revisited: Top Down Development

The port type inferencing (section 3.3.5) of Backbone works in concert with the redefinition and resemblance constructs. When the microphone was added in this case, the *mixerInput*, *output1* and *output2* ports were all automatically adjusted to cope with the sub-interface that was required.

This works well with bottom-up development where a composite is defined after its constituent parts have already been defined. In this case, the port types can be determined using inference. However, this causes a problem for top-down development where the decomposition of a composite is not yet available.

To support top-down development, the interfaces provided and required by a port may be explicitly specified for a composite (they must always be specified for a leaf). These will be checked against the inferred port interfaces for the home stratum of the component, where they must be the same if the component is considered to be well-formed. However, from the perspective of other stratum, only the inferred port interfaces are used (if available). In our example, it was possible to explicitly define the desk's *mixerInput* port as providing IAudio if this was desired. However, from the perspective of EvolvedDesk, the port provides IExtendedAudio because of the port type inferencing rules. The original, explicity indicated port type is ignored outside of its home stratum.

Placeholders can also be used to support top down development [61].

3.3.6 Summary

Backbone provides the redefinition construct for representing the evolution of a component. This confers a similar level of flexibility to the extension developer as is already available to the application developer (EXTEND). Redefinition allows a component in an underlying stratum to be replaced with a new definition. When combined with resemblance, redefinition allows a component to be incrementally evolved. A component can only redefine one other, which is not in the same stratum. Note that a redefinition cannot be directly referred to – we must instead refer to the component being redefined.

A redefinition does not affect the underlying application unless its owning stratum is explicitly included in the graph that the Backbone interpreter sees. The original application developers do not need to see the changes, or adjust their application to accommodate them (NO_IMPACT). This also allows the extension developer freedom to add new features that the base application developers would not normally consider.

In this example, AudioSoft has used redefinition to evolve an application, keeping the changes due to evolution separate from the original application. Users of the original application are not affected, but can choose to upgrade by combining their extensions with the evolving extension.

Backbone does not limit the changes that can be made to an application and can model the arbitrary evolution of components. However, it is in the developer's best interest to keep the changes small, in order to minimise the chance of conflicts between overlapping but mutually independent extensions. As will be shown later in this chapter, any conflicts can be resolved using redefinition.

3.4 Combining Extensions: Adding a Microphone and a CD Player

Unsurprisingly, another developer Z wishes to have a desk with both a CD player and a microphone. Z decides to use both extensions DeskWithCD and EvolvedDesk together . As the two stratum are independently developed and overlapping, structural and behavioural conflicts are possible when we try to use both at the same time. These conflicts are due to the different modifications and assumptions that the independent developers made when adapting the base application.

3.4.1 Combining Two Overlapping Extensions

All that is necessary to merge a number of overlapping strata is to create a further stratum that depends on them. If the Backbone interpreter is then given this stratum, it will construct a system based on this perspective.

For the example, we define the UnifiedDesk stratum to combine DeskWithCD and EvolvedDesk (figure 3.14).

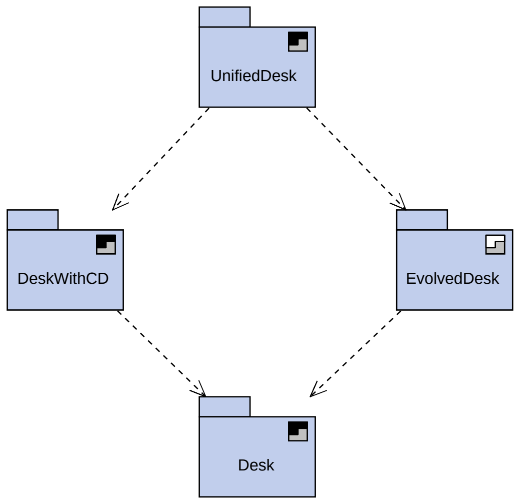


Figure 3.14: Combining two overlapping extensions

3.4.2 Redefinition Is Rewritten as Resemblance

Redefinition is handled by rewriting the resemblance graph of each component to take any redefinitions into account (section B.5.3). All resemblance graphs are recalculated for each stratum perspective.

Figure 3.15 shows the resemblance graph of AudioDeskWithCD from the perspective of UnifiedDesk. The topmost element of the graph becomes the AudioDeskWithCD component.

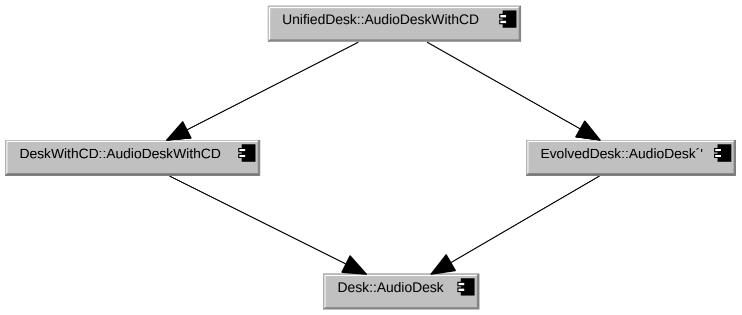


Figure 3.15: Redefinition is rewritten into the resemblance graph

Conflicts may occur because changes made in different branches of the resemblance graph conflict in some way. This is not the only way that conflict occurs. Another possibility is that a redefinition in one overlapping extension is incompatible with a definition in another independently developed extension causing a well-formedness rule to be violated. In practice, conflicts can be subtle.

From the perspective of UnifiedDesk, the AudioDesk component is the same as the redefinition shown in figure 3.12. There are no conflicting redefinitions to this component that must be reconciled. However, the redefinition has changed the definition of the desk component that AudioDeskWithCD resembles, thereby also changing the definition of AudioDeskWithCD.

The UnifiedDesk::AudioDeskWithCD component is implicitly created by the Backbone system in order to form a complete graph. The expanded structure of this component is shown in figure 3.16. The deletion of mixerInput has been applied as requested by AudioDeskWithCD, and both devices are present. However, there is no possible definition of the *deviceControl* port that can satisfy both the microphone (IDeviceControl) and the CD player (IDeviceControl and IDeviceCue) and the component is marked as malformed. We have a structural conflict that requires developer intervention.

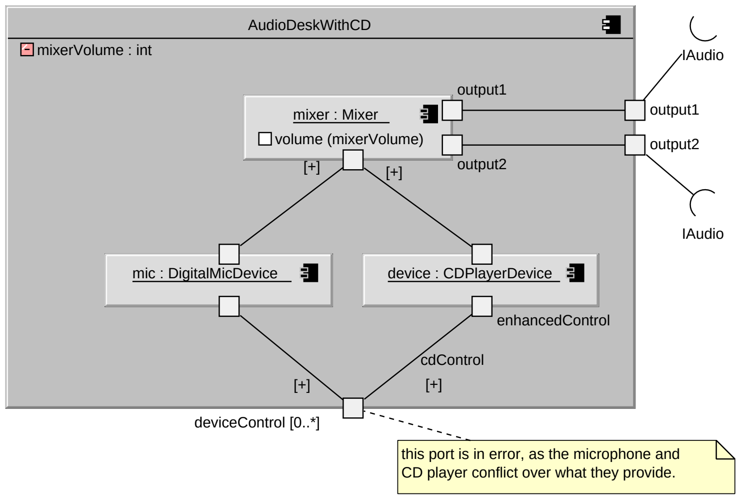
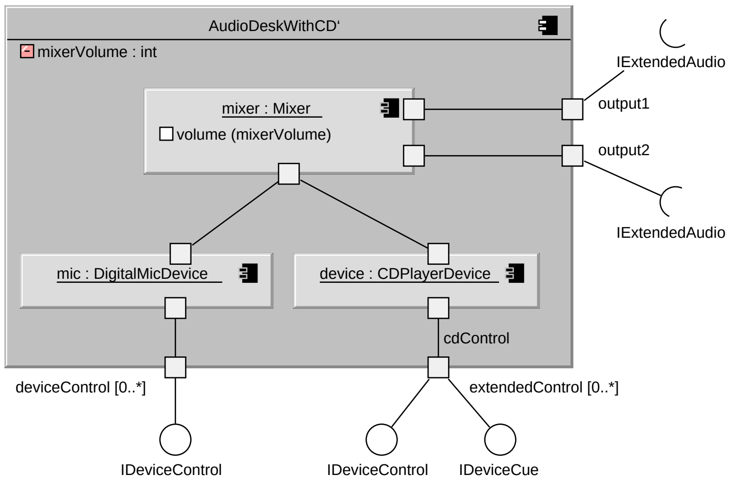


Figure 3.16: The conflicted AudioDeskWithCD component

Using a redefinition we can resolve the conflict neatly as shown in figure 3.17. A further port named *extendedControl* is created which provides both device and cue control. The CD player's control port is routed to this new port.



```
redefine AudioDeskWithCD
  resembles [previous] AudioDeskWithCD
{
  ports:
    extendedControl [0..*];
  replace-connectors:
    cdControl joins control@device to extendedControl[+];
}
```

Figure 3.17: Resolving the conflict by creating another port

3.4.3 Summary

Conflicts can occur when mutually independent and overlapping extensions are combined. Combination is possible because changes are kept as deltas (COMBINE). Conflict can always be resolved by further redefinitions, which make the developer's intentions explicit (VERIFY_AND_REPAIR). The key is to keep the number of changes as small as possible so that the chance of further conflict with other extensions is minimised. Well-formedness rules pick up structural conflict (VERIFY_AND_REPAIR).

Redefinition is rewritten into each component's resemblance graph, for each stratum perspective. Multiple resemblance can occur when combining overlapping extensions, as the rewritten graph must take into account the partial strata order. The detection and resolution of conflicts does not rely on a total order being specified for strata, unlike other approaches [40].

3.5 Dynamic Instantiation: Adding Feed Devices

The configurations presented so far have been static. Component instantiation occurs at start-up time, when the interpreter creates the required parts for the given configuration. However, in most applications a level of dynamic instantiation is required. To support this, Backbone provides *isomorphic factory* components. These are composite components which lazily instantiate their parts on demand,

and can instantiate multiple times. As composites, they can be redefined and resembled. They are isomorphic because the structure of the factory mimics the structure of the lazily created parts and connectors.

Consider that a developer wishes to add a number of audio feed devices to the desk. There are up to ten feeds available, each requiring a separate controller, and we wish to create these devices on demand. The definition of AudioFeedDevice is shown in figure 3.18.

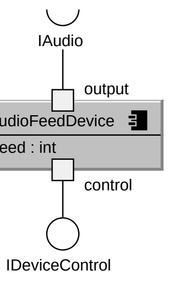


Figure 3.18: An audio feed device controls audio from feed inputs

In order to instantiate feed devices lazily, a factory must be created (figure 3.19). This is similar to a normal composite, except that it contains a create port which is not attached internally. The Backbone interpreter will locate this port and provide an internal implementation with the correct factory behaviour. The port will also contain functionality allowing the configuration attributes of the factory to be set (setFeed).

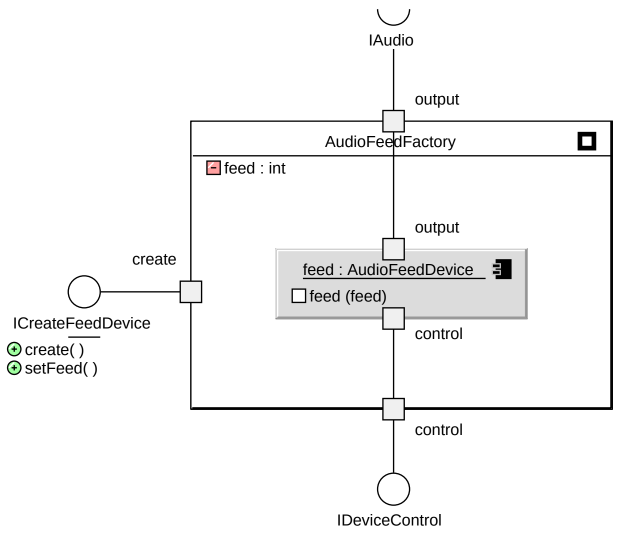
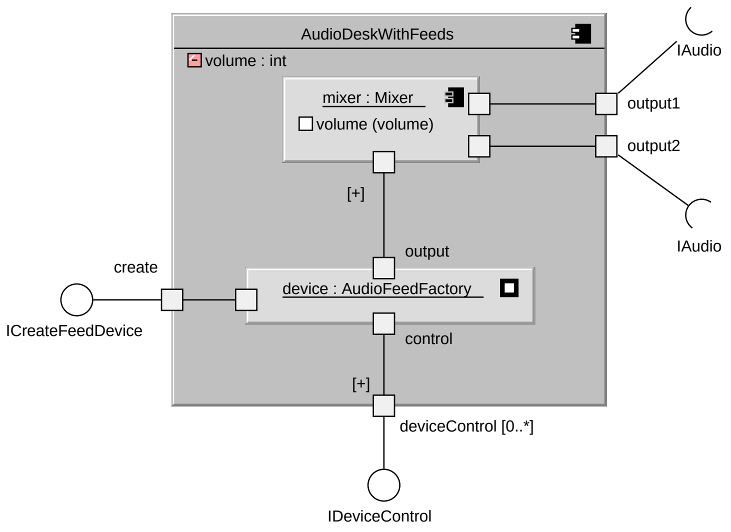


Figure 3.19: A factory for instantiating audio feed devices

Finally, to configure this into the desk, we create a new component AudioDeskWithFeeds which resembles AudioDesk (figure 3.20). The factory's create port has been exposed on the outside of the desk, and can be used to instantiate feeds. It is necessary to explicitly set the feed attribute via the create port before instantiating another part. Note that *[+]* is used on the connections from the factory so that a different index will be used each time.



```
component AudioDeskWithFeeds
  resembles AudioDesk
{
  replace-parts:
    AudioFeedFactory device;
  ports:
    create;
  connectors:
    createConn joins create@device to create;
}
```

Figure 3.20: Wiring up the feed factory into a desk component

3.5.1 Summary

Dynamic component instantiation is supported by factory components. As these are composite components, they can be redefined, resembled and may contain placeholders. This allows a number of useful architectural techniques to be used, including creating a factory which acts as a base component for a set of factories which resemble it. The *[+]* construct helps with multiple instantiation, allowing separate instantiations to take different indices.

3.6 Summary: Backbone as an Extension Architecture

The Backbone approach is designed to support creating and managing architectures and extending them. Extension developers have the freedom to make arbitrary changes to the underlying application, and the constructs support the division of concerns between extension and application developers. The hierarchical model simplifies the definition and management of the complex application architectures (MANAGE). By allowing an extension to adjust the architecture at the appropriate level, change is kept to a minimum which also minimises the chances of any conflicts between independently developed, overlapping extensions.

The changes are specified to the architectural definition, not to the implementation code. This allows the approach to work even when the implementation code is not available, as long as the Backbone architectural description of the system is given. This partially addresses the requirement not to reveal source code in the EXTEND requirement.

Resemblance allows one component to be defined in terms of delta changes to another. Redefinition allows a new definition to be substituted for an existing component, and models component evolution. Redefinition and resemblance together allow existing components to be incrementally modified by an extension (EXTEND). Port type inference cuts down the number of changes required when adding or removing parts from existing components (UPGRADE).

Note that the resembling and redefining components are kept in delta form, and applied to the original definitions at application start-up time. The use of deltas makes possible the combination of extensions, and subsequent repair of conflicts.

Overlapping extensions can be combined using a further extension which brings them together via strata dependencies. This allows multiple redefinitions to be combined into a unified system (COMBINE). Structural conflicts are picked up by well-formedness rules, and can be resolved using further redefinitions (VERIFY_AND_REPAIR).

The Backbone approach is also applicable for component reuse and integration, given that the requirements for dealing with application extensibility represent a superset of these requirements. The constructs provided reduce the abstraction problem, which results from trying to reuse highly specific components which encode assumptions deep within their composition hierarchy [34]. Redefinition and reuse can be used to alter such components to fit into a new context [58].

3.6.1 Limitations

The approach has some limitations in practice:

- Cross-cutting concerns are not handled well.
  Like conventional component-based systems, cross-cutting concerns such as logging and security must be handled in the traditional manner where logic may be spread across many components. Also, changing an interface may have a wide-ranging impact on the definition of a system. Redefinition can always handle the required "repairs", but these may be extensive and may cut through many component definitions. Port type inference helps by propagating type changes, but this cannot handle the implementation implications of deleting an operation of an interface, for example.

- Modeling with deltas is sometimes inconvenient.
  It is not always convenient to model changes uniformly using the constructs provided. For some types of change, it is often simpler just to modify the application directly. In circumstances where this type of modification is permissible or even possible, Backbone provides support for checking that the changes are compatible with existing extensions.

- Dynamic extension is not supported.
  The Backbone runtime model does not support the dynamic application of extensions. This reflects a limitation of the runtime interpreter, rather than a fundamental issue with the approach.

- The application must be specified using Backbone.
  In order to take advantage of the extensibility constructs, an application must be specified in Backbone. To retrofit the Backbone approach onto an existing object-oriented application can involve significant changes and refactoring.

Chapter 4

Work to Completion

4.1 Expected Thesis Contents

1. Introduction
   (phrasing of problem statement, example scenario, presentation of requirements taken from scenario. See actual chapter)

2. Background
   (focus on plugin architectures, component models such as Darwin, and product family approaches. See actual chapter)

3. Using Backbone to Define, Extend and Evolve an Architecture
   (extends on example from (1) to show have Backbone is used in a real architecture, and how it meets the requirements outlined. Focus is on the structural side. See actual chapter)

4. Modeling and Checking Component Protocols in an Extensible System

5. Describing and Checking Goals in an Extensible System

6. Advanced Modeling in Backbone
   (discussion about stereotypes, the breakdown of behaviour into structural components, and baselining)

7. A Graphical Modeling Environment for Backbone
   (the jUMbLE case tool will be presented, showing how it supports modeling with deltas, and conflict resolution)

8. Case Studies
   (evaluation of part of the Eclipse architecture, application to a JEE applications, and also a presentation of the jUMbLe architecture specified using Backbone. See evaluation criteria below)

9. Conclusion and Future Work
   (discussion of contribution, description of limitations, and possible future work)

10. Appendix A: Backbone Structural Concepts
    (informal description of structural concepts. see actual appendix)
11. Appendix B: The Formal Backbone Structural Specification
    (formal description of structural concepts. see actual appendix)
12. Appendix C: Mapping Protocols and Goals to FSP
    (description of the mapping to finite state processes. Work done, needs to be written up)
13. Appendix D: Alloy Models
    (the models that appendix B describe)

4.2 Expected Contribution

The core contribution of this work is the addition of architectural hierarchy to a plugin-like approach for creating extensible applications. The approach arguably offers a superior engineering tradeoff between flexibility of extension and architectural manageability than other approaches. Conflicts, which plague conventional extension architectures, can be checked at both a structural and behavioural level, and resolution can be performed when mismatches occur.

The constructs provided by Backbone also give rise to a set of advanced architectural techniques (to be in chapter 6) which I have not found discussed in the literature. Two of these are (i) baselining which compresses deltas, forming extensions which can reproduce previous versions of an application and (ii) behavioural decomposition into structural components where control logic can be decomposed into fine-grained structural components. In the latter case, the diagrams can be made to look like state charts, but offer all of the architectural advantages of components including support for redefinition.

4.3 Evaluation Criteria

The approach is to look at various applications expressed using existing plugin and other extensibility approaches, and examine how Backbone's support for architectural hierarchy and structural and behavioural checking could describe and verify the architectures more appropriately. The focus will be on detecting conflicts between extensions and evolved versions of the application, and also using an architectural hierarchy to describe the systems at the appropriate level of abstraction.

- The Eclipse Integrated Development Environment (IDE) [42]
  The Eclipse IDE contains a complex nest of components and connections due to its lack of true architectural hierarchy. The intention is to find an example in the current IDE's architecture where this has led to a more complex description of part of the architecture, and express it in Backbone to show the simplification. Other possibilities are showing how Backbone allows for more flexibility in making extensions, and allows old extensions to be checked against an updated form of the application and also against other extensions for conflict. These are all problems in the Eclipse extension model, which is generally considered to be an example of a flexible and powerful model.

- An Enterprise Java Enterprise (JEE) Application
  Backbone will be applied to the architecture of a JEE application. The Spring framework [82] offers a non-hierarchical configuration approach for describing the structure of JEE application. This approach is in widespread industrial use. The intention is to take an existing Spring or Spring.NET application, and show how Backbone can express the architecture more understandably using hierarchy. Further, the aim is to show that Backbone offers a model that can describe both the evolution of the configuration and also a family of applications allowing a product family approach. It is anticipated that it will be possible to modify Backbone so that it generates Spring flat configurations as part of its output, allowing the two approaches to be directly compared.

Finally, as part of this work, the jUMbLe architecture will be restructured into a Backbone model and presented as an example of a complex system modeled using this approach. An example extension and evolution will be shown.

4.4 Plan

The following work forms part of this thesis:

1. Description of Backbone structural concepts
   (completed)

2. Backbone structural specification and well-formedness rules using a formal Alloy model
   (completed)

3. Formal specification for component "flattening" to form the basis of the runtime model. The concurrency approach will also be specified.
   (August 2007)

4. Backbone interpreter
   (mostly completed, current adjusting for the results in 2 and 3)
   (complete September 2007)

5. Translation of protocols into FSP
   (completed)

6. Modification of jUMbLe CASE tool to allow structural modeling with deltas
   (October 2007 to February 2008. Tool is currently written, but requires modifications to deal with results of formal specification)

7. Specification of goals, and translation into FSP properties
   (March to April 2008)

8. Modification of jUMbLe CASE tool to allow behavioural modeling
   (May to June 2008)

9. Refactoring of jUMbLe's architecture to use the Backbone approach
   (July to August 2008. The current architecture of jUMbLe was used to create the Backbone approach, so this is not anticipated to involve large amounts of work)

10. Case studies and writeup
    (September 2008 to January 2009)

Appendix A

Backbone Structural Reference

Backbone is an ADL for modeling component architectures. The focus is on reducing the limitations on the extension and reuse of existing applications and components. It is also designed to allow static analysis of the structure and behaviour of combinations of extensions.

This chapter describes the structural side of Backbone. The structural model is informally outlined in this chapter, and both the textual and graphical forms are presented. In appendix B, an Alloy specification is used to formally describe the rules that govern the correct structure of components, as well as to show how the redefinition and resemblance constructs operate.

A.1 Components and Interfaces

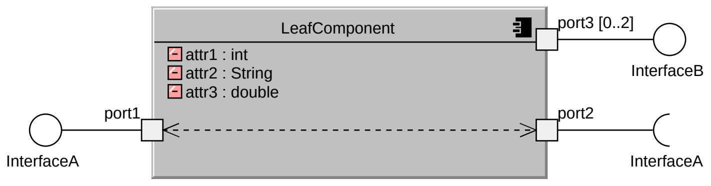
In keeping with Darwin [53] and UML2 [76], we define a component as an instantiable, class-like construct which describes the services it requires and provides via interfaces. An interface represents a collection of methods defining a service. Interfaces can only be provided or required via ports, and each port has a name and a multiplicity. If not multiplicity is specified, it is assumed to be *[1]*. Ports serve to name the role of interfaces as services offered or required by a component.

A component may have attributes, which can only be of primitive type. These present a view, or projection, onto the internal state of the component.

Components are either leaf or composite, where a leaf component cannot be further decomposed and is associated directly with an implementation in (currently) Java.

Figure A.1 shows a leaf component with two attributes and two ports. A provided interface is shown as a circle, and a required interface is shown as a semi-circle. Note that each leaf is directly associated with a Java implementation class.

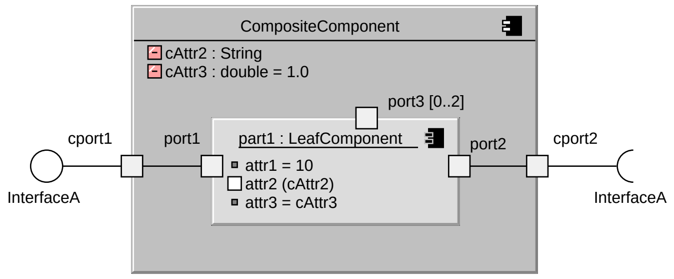
Port3 is shown as having a multiplicity of [0..2]. This means that the port can have between 0 and 2 connections. The number of connections is allowed to change during runtime, as long as the mandatory set (described by the lower bound) are always connected. The upper bound restricts the total number of connectors. The number of mandatory indices is *(lower)*, and the number of optional indices is *(upper - lower + 1)*.



```
component LeafComponent
  describes com.example.JavaLeafComponent
{
  attributes:
    int attr1;
    String attr2;
    double attr3;
  ports:
    port1 provides InterfaceA;
    port2 requires InterfaceA;
    port3[0..2] provides InterfaceB;
  port−links:
    port1 links−to port2;
}
```

Figure A.1: Definition of a leaf component

A composite component (figure A.2) contains a number of component instances, each of which is shown as a box within the component. These instances are called *parts*. Each part has a name (part1), and a type (LeafComponent) which is the component that the part is an instance of. Further, a part can define slots, which hold values for the attributes of the type. The parts of a composite represent its initial configuration and state.



```
component composite CompositeComponent
{
  attributes:
    String cAttr2; double cAttr3 = 1.0;
  ports:
    cport1;
    cport2 requires InterfaceA;
  parts:
    LeafComponent part1
      set attr1 = 10,
           attr2 aliases−environment−attribute cattr2,
           attr3 = cattr3;
  connectors:
    conn1 joins cport1 to port1@part1;
    conn2 joins cport2 to port2@part1;
}
```

Figure A.2: Definition of a composite component

In part1, the attr1 slot is set to 10. Attr2 is aliased to cAttr2 of the enclosing component, as shown by the brackets and the empty box next to the slot. This aliasing offers a way to propagate the state of constituent parts into a higher level component. Attr3 is not aliased to another attribute and holds its own state, but will initially be set to the value of cAttr3.

The ports of a part are referred to as port instances. These are shown as small boxes on the outline of a part. Often port instances are simply referred to as ports when the context makes the meaning obvious.

Components and interfaces are referred to collectively as elements. A composite component is not associated with an implementation class – it only has a Backbone definition.

A.2 Connectors

Ports and port instances are "joined" together using connectors. In Backbone (as in UML2 [33], and Darwin [53]), connectors represent little more than an aliasing of two different ports. A port that only provides interfaces can have an arbitrary number of connections (including none), for any given index. If a port has some required interfaces, it may only have one (or perhaps none, if the port multiplicity indicates it is optional) connector for a given index.

A single delegation connector can be used to alias 2 ports together, for all indices, rather than having to have multiple ordinary, indexed connectors. Figure A.4 shows a delegate connector between cport3 and cport3@part1.

If an ordinary connector goes to a port which has a multiplicity of other than *[1]* or *[0..1]*, then it must indicate an index. It can use *[+]* as the index, which takes up any available index left and increments a counter.

A.3 Stratum

A stratum is a module-like construct, which contains a set of elements. Each element is owned by exactly one stratum. Each stratum may have dependencies on other strata, and these represent visibility constraints that elements must adhere to. Stratum are non-hierarchical, for reasons of simplicity rather than any technical limitation. Amongst other things, a stratum can represent an application layer, an extension, an evolution of a system, a subsystem or library definition or simply a namespace.

Stratum can be marked as relaxed. Relaxed stratum transitively promote their dependencies.

In figure A.3, only stratum B is relaxed. Elements in A can refer to elements in A, B, C and D. B's elements can refer to elements in B and C. C and D can refer to no other stratum's elements apart from their own.

When running a Backbone application, we specify a set (or subset) of strata. For instance, it is valid to specify strata B and D. The application always includes transitive dependencies, making the full set included as B, C and D. Alternatively, specifying A alone would include A, B, C and D.

Strata dependencies define a partial order, and must always be acyclic. Dependencies are allowed to be redundant. For instance, it is legal for A to further express a dependency to C, even though it can already see C's elements.

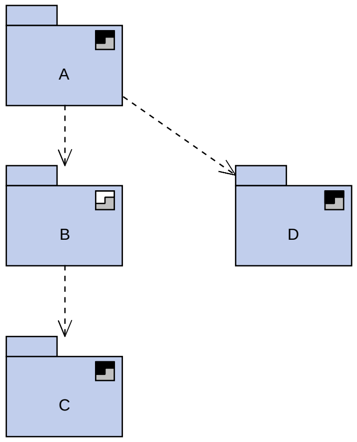


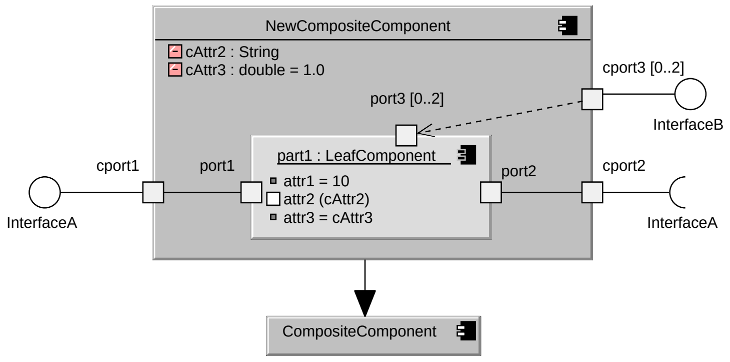
Figure A.3: Strata dependencies

A.4 Resemblance and Redefinition

Composite components and interfaces, can be redefined and resembled.

Resemblance is an inheritance like construct. For composite components, it allows one composite to be defined in terms of differences to other composites. Parts, ports, attributes and connectors can be added, deleted and replaced. For instance, in figure A.4, NewComposite has introduced another port.

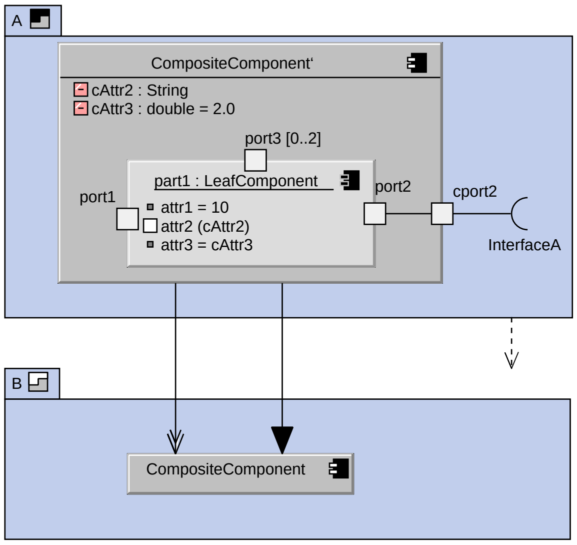
Note that multiple resemblance is permitted. The resemblance graph cannot be cyclic.



```
component NewCompositeComponent
  resembles Component
{
  ports:
    cport3[0..2];
  connectors:
    conn3 joins cport3 to port3@part1;
}
```

Figure A.4: A composite can resemble another composite

Redefinition is a separate concept that allows an existing element in one stratum to be replaced by a definition in another stratum. It literally takes over the logical identity of the element being redefined, and replaces it with another definition. Redefinition can be combined with resemblance to apply incremental changes to an existing definition, which is used to model component and interface evolution. Consider figure A.5, which shows a redefinition of CompositeComponent.



```
redefine−component CompositeComponent
  resembles CompositeComponent
{
  delete−ports:
    cport1; cport2;
  delete−connectors:
    conn1;
}
```

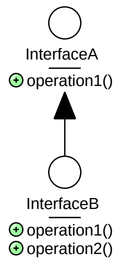
Figure A.5: A composite can be redefined in another stratum

If only stratum B is included when running the application, then CompositeComponent will be as defined in figure A.2. If however, stratum A and B are included, then the definition will be as shown at the top of figure A.5. This shows an important point about redefinition – it provides the facility to change an element in an existing system by the inclusion of another stratum. A practical application of this is when a developer changes the definition of an element in a system supplied by another developer or vendor.

A redefinition cannot be referred to directly, and is not named. All references must be to the redefined element.

Note that a leaf component cannot be redefined or resembled. Generally, a leaf component should be composed (wrapped) by a composite component which represents it, and only the composite should be further used in composition (section 3.2.3). This sidesteps the issues related to a composite redefining or resembling a leaf and vice versa.

Interfaces can also be resembled and redefined by other interfaces. In this case, operations can added, deleted and replaced. Inheritance, where operations can only be added, is not necessary as it is as a subset of resemblance. If an interface resembles another, and does not delete or replace any operations, then the subtype relationship is preserved. In figure A.6, InterfaceB resembles InterfaceA and is also a subtype of it.



```
interface interfaceA
{
  operations:
    operation1;
}

interface InterfaceB
  resembles InterfaceA
{
  operations:
    operation2;
}
```

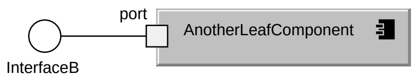
Figure A.6: InterfaceB adds operation2, and is a subtype of InterfaceA

A.5 Port Type Inference

In the definition of CompositeComponent (figure A.2), cport1 did not specify its interfaces, as these can be inferred from the connections. It is always necessary, however, to indicate the interfaces for ports of leaf components.

A port link, such as that defined between port1 and port2 (figure A.1), indicates that if a required interface is actually supplied a subtype, then this propagates through to any linked port's provided interfaces. This also works the other way around.

As an example, consider if we have an additional leaf as in figure A.7.



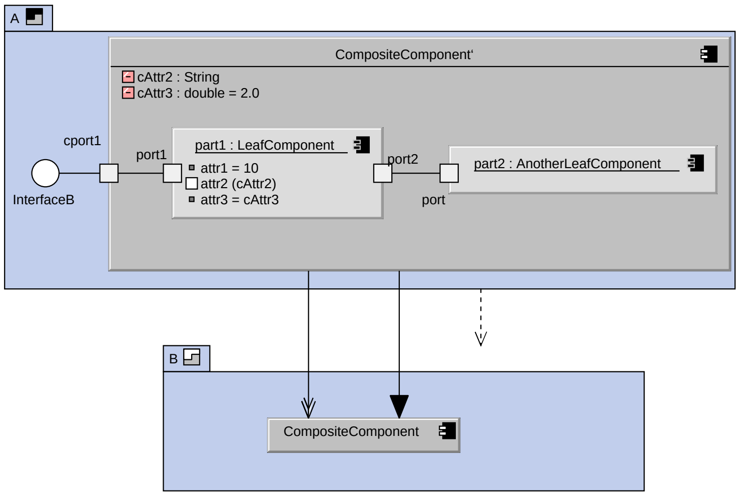
```
component AnotherLeafComponent
  describes com.example.AnotherJavaLeafComponent
{
  ports:
    port provides InterfaceB;
}
```

Figure A.7: Another leaf component

We now redefine CompositeComponent to remove cport2 and add in part2 of type AnotherLeaf (figure A.8). Cport1 is now inferred to provide InterfaceB, which has propagated through LeafComponent via the port link.

Port type inference minimises the redefinitions required when redefining and resembling components. Port types will be updated automatically to reflect any altered parts or connections.

In order to support top-down definition, it is possible to explicitly indicate a composite's port interfaces (as with cport2 in figure A.2). This will then be cross-checked against the inferred interfaces, but only in the home stratum for that component. If, for example, a redefinition in stratum X meant that cport2 was now inferred to require InterfaceB, then this would be acceptable as the check against InterfaceA only occurs in stratum B, which does not see the redefinition.



```
redefine−component CompositeComponent
  resembles CompositeComponent
{
  delete−ports:
    cport2;
  replace−connections:
    conn2 joins port2@part1 to port@part2;
}
```

Figure A.8: Port inference to a subtype after redefinition

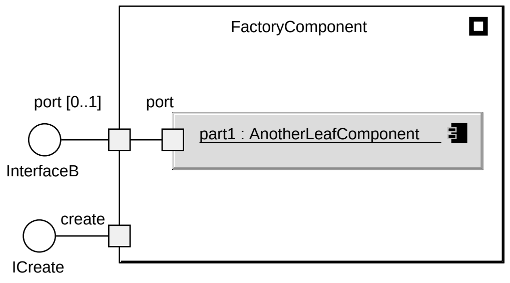
A.6 Factories and Placeholders

A factory is a composite component whose parts are lazily instantiated on demand, and can be instantiated more than once if required. As a composite, it can be redefined and resembled. A factory must have a port, which provides a "create" interface which is not connected. The Backbone interpreter generates the appropriate instantiation code for this port. Figure A.9 shows a factory which instantiates a leaf when the create method on the interface is programmatically invoked from another component. Note that any "non-create" ports of the factory must be optional to deal with lazy instantiation. Repeated instantiation will either overwite existing connections, or taken the next index if [+] is used for the connector end (sectionA.2).

Factories are *isomorphic*, because the structure of the lazily created parts is the same as their internal structure.

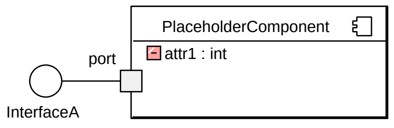
A placeholder is a component which can be used to defer the definition of a component, supporting top-down development. It can also be used to indicate the "shape" of the component which must replace it, and to allow semi-finished components to be packaged. A placeholder is a composite which has no parts, but which may have attributes. It can be redefined and resembled by other composites. A component containing a placeholder may not be instantiated via the Backbone interpreter.

Through redefinition, a placeholder can also be made into a normal composite. Figure A.10 shows a placeholder.



```
component FactoryComponent is−factory
{
  ports:
    port[0..1];
    create provides ICreate;
  parts:
    AnotherLeafComponent part1;
  connectors:
    conn1 joins port to port@part1;
}
```

Figure A.9: A factory is used to instantiate parts lazily.



```
component PlaceholderComponent
  is−placeholder
{
  attributes:
  int attr1;
  ports:
    port provides InterfaceA;
}
```

Figure A.10: A placeholder component

Appendix B

The Formal Backbone Structural Specification

B.1 Approach

The formal specification is described using Alloy, a modeling language based on first order logic [44, 43]. Alloy is supplied with a model finder which can check assertions and find counterexamples within a finite model space.

The formal specification has two primary goals.

Firstly, it states the full set of structural rules governing well-formedness. These rules show how errors can occur when multiple independently developed, but overlapping, redefinitions are combined. Further, it shows how redefinitions can resolve these errors. An important point of the specification is that it explicitly models errors due to conflicts between independently developed strata, as well as the resolution of those errors.

Secondly, the specification must be usable for developing a case tool. We desire to graphically show the full, expanded component definition at all times, including any artifacts which have been acquired through resemblance or redefinition. This allows definition and redefinition to appear the same graphically. This is despite the fact that the CASE tool must record deltas. A secondary concern is to show any errors as they occur without requiring an explicit checking phase if possible.

In order to meet these goals, this specification does not flatten the composition and resemblance hierarchy into connections between leaf component instances. Instead, it defines the well-formedness of a component in terms of any deltas it contains, in conjunction with the well-formedness of any components it is directly composed of, or resembles. The benefit is that this allows an incremental and modular approach to checking which is computationally inexpensive. This is feasible to present in the CASE tool as interactions happen, without requiring an explicit checking phase.

B.1.1 Names, Renaming and UUIDs

Although the Backbone code presented in chapter A contains human-readable names such as "port1" and "attr1", this approach does not work well in practice. Independently developed stratum can easily contain elements which use the same textual names, leading to conflicts. This has been an issue for multiple inheritance in textual programming languages for some time [89] and also causes problems for mixins [66].

In order to resolve this dilemma, we use universally unique identifiers (UUIDs) to identify elements. An element retains the same UUID always as its logical identity. Each time a new UUID is introduced, it is *guaranteed* to be universally unique and a developer can be assured that their UUIDs will not conflict with anyone else's. This approach works well within a graphical CASE tool, where conventionally each element is assigned a unique identifier as well as a human-readable name.

In order to deal with the fact that UUIDs are not particularly easy for a designer to understand, we do not present these to the user and instead allow each UUID to be mapped onto a human-readable name. This mapping is referred to as the *name dictionary.* In the CASE tool, each stratum contains part of the dictionary, and holds the mapping for the UUIDs introduced in that stratum.

Renaming is supported by allowing a new name to be associated with an existing UUID visible to a given stratum. In the case of a name clash, where multiple strata rename the same UUID to different names, both names will be shown, allowing a further stratum to rename to a single name. *It is important to note that any conflicts are presentational in nature only, and do not cause an error in the underlying Backbone model.*

Figure B.1 shows an example where a component is redefined in 2 independent stratum, and then further merged in the top stratum. Although the both attributes have the name "number", in practice these are separately identified by different UUIDs, and through the name dictionary the attributes can be renamed in stratum Top. In the graphical presentation, the user of the case tool will only see the names not UUIDs.

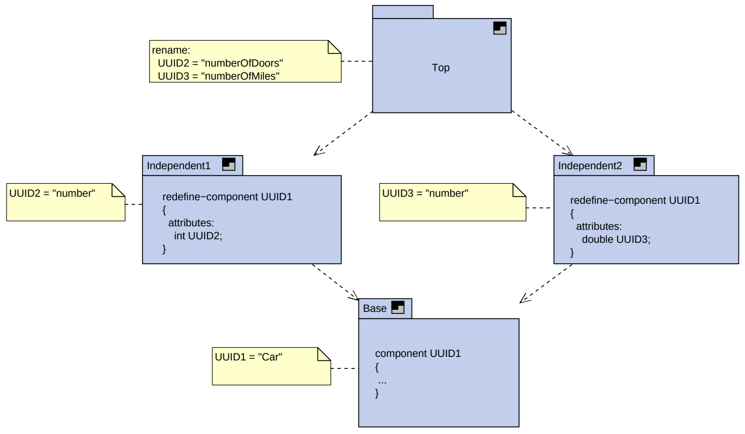


Figure B.1: Renaming as a presentational issue

Note that the formal specification only deals with UUIDs (modeled as PortID etc). The treatment of names is considered to be purely a presentation issue.

B.1.2 Stratum Perspective

In a stratum, it is possible to redefine components in depended-upon stratum. For instance, in figure B.1, if we only include stratum Independent1 and Base in our application, then the Car component will look as in figure B.2.

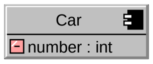


Figure B.2: Car viewed from the perspective of stratum Independent1

However, if we include Top and all other strata, the Car component will look as in figure B.3.

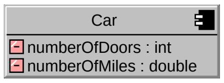


Figure B.3: Car viewed from the perspective of stratum Top

This is known as *stratum perspective*. When designing graphically in a diagram owned by a stratum, the designer sees the component from that stratum's perspective, showing what the components would be like if only that stratum (and others it transitively depends on) were loaded.

B.1.3 Modeling Conflicts

A naive approach to structural well-formedness would identify a set of rules, that if not satisfied would indicate that a given model is not suitable. This was the initial approach taken, but this cannot show both conflicts and how further redefinitions resolve those conflicts which is a key aim of the specification.

To handle this dilemma, we have modeled errors as part of the specification, dividing the logic between a setting up stage which constructs various structures through predicates, and the checking stage which examines these structures and marks any errors.

B.2 Concepts

The following concepts (apart from constituent and r-element) are represented by signatures in the Alloy specification.

Model A model is collection of strata representing an application.

Stratum A stratum is a non-hierarchical package construct, that contains elements. It must explicitly indicate which other strata it depends on.

Element An element is the base signature for components and interfaces.

R-element An r-element is an element which is allowed to participate in resemblance and redefinition relationships. Composite component and interface are r-elements, but leaf component is not. All element rules also apply to r-elements, but not vice versa. This is not an explicit signature in the specification.

Constituent A constituent refers to a port, part, attribute or connector of a component, or an operation of an interface. It is not an explicit signature in the specification.

Interface An interface represents a service and contains a collection of methods describing that service.

Component A component is a unit of software composition. A leaf component is atomic (not further decomposable into parts) and is directly associated with an implementation. A composite component contains parts, which are instances of other components. Components expose interfaces via ports, and expose a projection onto their internal state via attributes. Parts are connected internally to ports or to other parts (via port instances) using connectors.

Part A part is an instance of a component. Parts are only constituents of composite components.

Port A port is a constituent of a component, describing which interfaces are offered or provided. A port may also have a multiplicity, with a lower and upper bound. A part has port instances reflecting the ports of the part's type.

Connector A connector is a constituent of a composite component, used to wire up the internals by connecting the component to parts or parts to parts (always via ports or port instances).

ConnectorEnd A connector has two connector ends, each of which represents where the connector attaches to a port or port instance. A connector end is the base signature for component and port connector ends.

ComponentConnectorEnd A component connector end is a connector end which is attached to the port of a component.

PartConnectorEnd A part connector end is a connector end which is attached to the port instance of a part.

Attribute An attribute is a constituent of a component which exposes a projection onto the component's internal state. An attribute is associated with a single attribute type.

AttributeType An attribute type is the type of an attribute.

AttributeValue An attribute value represents a literal value of an attribute. It has an attribute type.

Operation An operation is a constituent of an interface, defining a single method.

InterfaceImplementation An interface implementation represents an interface in an implementation language. It is specified only for an interface.

ComponentImplementation A component implementation represents a class in an implementation language that implements the functionality of a leaf component. It is a specified only for a leaf component.

Link A link indicates the typing relationships between two ports in a leaf component, for inference purposes. It is not represented by a signature in the specification, but by a tuple in the component definition. A link has two link ends. Links are required to indicate the internal connections inside a leaf.

LinkEnd A link end is one end of a link. It is the base signature of component link end and part link end.

ComponentLinkEnd A component link end is the end of a link connected to the port of a component.

PartLinkEnd A part link end is the end of a link connected to the port instance of a part.

Deltas This is a structure for holding constituents of an element, capable of reflecting changes due to redefinition and stratum perspective.

PartID / PortID / ConnectorID / AttributeID / OperationID These signatures represent the UUIDs of constituents.

B.2.1 Additional Terms

Independence A stratum is independent from another if it does not transitively depend on it.

Top stratum There is one top stratum per model, bringing together all otherwise independent strata. Nothing may depend on the top stratum. This stratum will be implicitly created if the model does not include a suitable candidate.

Home An element's home is the single stratum that owns it.

Stratum persective A view of an element from a particular stratum is known as stratum perspective.

Conflict A conflict occurs when an independent extensions are combined, and changes made in one branch are not compatible with assumptions or changes in another. A key well-formedness rule is that the number of constituents present for a given UUID in an element should be exactly one.

Redefinition This refers to an r-element which takes the logical identity of another r-element.

Port instance This refers to a particular port of a part. i.e. the instantiation of a port.

Interface scope The scope of an interface is the set comprising that interface, and all its sub and super-interfaces.

Interface overlap Two interfaces are said to be overlapping if their scopes intersect.

One-to-one interface mapping A one-to-one mapping between two sets of interfaces exists only when it is possible to construct a unique and complete set of tuples (first, second) where the first and second interfaces overlap.

Provides enough / requires less Interface B provides enough for interface A if B is A or a sub-interface of A. We also say that A requires less than B provides.

Incoming port instance An incoming port instance is a port instance which is found by traveling along a connector. Incoming refers to the direction we are traveling in.

Outgoing port instance An outgoing port instance is a port instance which is found by traveling along a link (i.e. traveling inside a part, away from another port instance, through to an outgoing port instance). Outgoing refers to the direction we are traveling in.

Terminal port instance A port instance which has no links.

B.3 Rules

Rules which are prefixed by WF_ are well-formedness rules. These may be violated in the case of a conflict, and must be corrected using a further redefinition.

Stratum rules:

1. STRATUM_ACYCLIC
   No stratum dependency cycles are allowed.
2. STRATUM_VISIBILITY
   A stratum has visibility of the strata it is dependent on, and any strata that these transitively depend on. The exception is that if a strata is not relaxed, then a stratum depending on it cannot see through to the strata that it depends on.
3. STRATUM_TOP
   All strata must be transitively reachable from the top stratum. Nothing can depend on the top stratum, and there must be exactly one top.

Element rules:

1. ELEMENT_HOME
   Each element is owned by one stratum (its home).
2. ELEMENT_VISIBILITY
   Elements in a stratum can refer only to elements in their home, and the stratum that the home can see.
3. ELEMENT_OK_AT_HOME
   An element must be well-formed from its home stratum's perspective.
4. ELEMENT_CONSTITUENTS
   A constituent (port, part etc.) must be associated with at least one of the correct type of element (component or interface).
5. ELEMENT_RESEMBLES
   An r-element can resemble zero of more other r-elements of the same kind.
6. ELEMENT_REDEF
   An r-element can redefine another r-element of the same kind, but only if it is in a different stratum.

7. ELEMENT_REDEF_PER_STRATUM
   An r-element can only be redefined once per stratum, and an r-element cannot be redefined in its home.
8. ELEMENT_REDEF_NOT_REFERENCED
   Nothing can resemble, redefine or otherwise refer to a redefinition.
9. ELEMENT_NO_RESEMBLANCE_REDUNDANCY
   There can be no redundancy in the resemblance relationship.
10. ELEMENT_REDEF_RESEMBLE
    From the perspective of each stratum, redefinition is turned into resemblance and a possible rename.
11. ELEMENT_INHERITS
    An r-element contains the constituents it "inherits" from any elements it resembles, and then adds, deletes or replaces constituents to get the full set.
12. WF_ELEMENT_ACYCLIC
    Redefinition and resemblance must be acyclic, even after redefinitions have been applied.

Interface rules:

1. INTERFACE_CONSTITUENTS
   The constituents of an interface are operations.
2. INTERFACE_IMPLEMENTATION
   An interface is associated with an implementation interface.
3. INTERFACE_SUBTYPE
   An interface is a subtype of any interfaces it resembles, unless it replaces or deletes the operations it has inherited from those interfaces.
4. WF_INTERFACE_SOME_OPERATIONS
   A well-formed interface must have some operations.
5. WF_INTERFACE_ONE_OPERATION_PER_UUID
   A well-formed interface must have only one operation per specified or referenced UUID.
6. WF_INTERFACE_ONE_IMPLEMENTATION
   A well-formed interface should only have one implementation interface.

Component rules:

1. COMPONENT_TYPE
   A component is leaf or composite.
2. COMPONENT_NO_LEAF_REDEF_RESEMBLANCE
   Leaf components cannot participate in redefinition or resemblance.

3. COMPONENT_LEAF_IMPLEMENTATION
   A leaf component is associated directly with an implementation.
4. COMPONENT_LEAF_CONSTITUENTS
   The constituents of a leaf component are ports, attributes and links.
5. COMPONENT_COMPOSITE_CONSTITUENTS
   The constituents of a composite component are ports, attributes, connectors and parts.
6. WF_COMPONENT_CONTAINMENT
   A well-formed component can never (transitively) contain itself, after taking resemblance and redefinition into account.
7. WF_COMPONENT_SOME_PORTS
   A well-formed component must have some ports.
8. WF_COMPONENT_ONE_CONSTITUENT_PER_UUID
   A well-formed component must have only one constituent per specified or referenced UUID.
9. WF_COMPONENT_SOME_PARTS
   A well-formed composite component must have some parts, and some connectors.
10. WF_COMPONENT_CONSTITUENTS_OK
    A well-formed component must have well-formed constituents.

Part rules:

1. PART_TYPE
   A part is an instance of a component, which is called its type.
2. PART_SLOTS
   A part's slots can either be aliased, have a literal value, or be set to the value of an enclosing attribute in the containing component.
3. PART_SLOT_LITERALS_OK
   Any attribute values must be of the correct type.
4. PART_PORT_REMAP
   In a redefinition, the port instances of the replacing part can be mapped onto the port instances of the part that is being replaced.
5. WF_PART_SLOTS_DEFAULT
   Any slot that does not have a value or an alias, must be of an attribute with a default value.
6. WF_PARTS_SLOT_REFERENCE
   If a slot is aliased or set to the value of an enclosing attribute, that attribute must exist and be of the correct type.
7. WF_PART_NO_ISLANDS
   A well-formed part must be eventually linked to a port, ensuring no "islands" occur where there are no connections back to the component's ports.

Connector rules:

1. WF_CONN_JOIN
   A connector connects from a port or a port instance to another port instance. A port to port connector is not allowed in order to simplify the port type inference specification.
2. WF_CONN_MANDATORY
   A connector from a mandatory port instance must connect to a mandatory port instance or mandatory port.
3. WF_CONN_OPTIONAL
   A connector from an optional port instance must connect to an optional port instance or optional port.
4. WF_CONN_INDEX
   The index of a connector end must match with a possible index of the port or port instance.

Port and port instance rules:

1. PORT_MULTIPLICITY
   A port has a multiplicity *[lowerbound, upperbound]*. Indices (0..lowerbound-1) are mandatorily connected and indices (lowerbound..upperbound-1) are optionally connected.
2. WF_PORT_CONNECTED_ALWAYS
   A port must have a connection on each index regardless of whether it is mandatory or optional.
3. WF_PORT_PROVIDES_ONLY
   If a port instance has only provides, it can have zero or more connections for the same index even if it is mandatory.
4. WF_PORT_COMPLEX
   If a port instance doesn't have only provides, it must have at exactly one connection on each mandatory index, and at most one connection on each optional index.

Port type inference rules:

1. PI_LINKS
   A leaf component may have explicit links between ports.
2. PI_SPECIFIC
   All interfaces for ports of a leaf component must be explicitly specified.
3. PI_LINKS_COMPLEMENTARY
   Two ports that are explicitly linked in a leaf component must have exactly complementary interfaces (provided mapped to required and vice versa).
4. PI_INFERRED_LINKS
   A composite component does not specify links explicitly. They have inferred links which are formed by following connectors and links of the part types between ports and port instances.

5. PI_PROVIDE_MORE
   If a sub-interface is provided to a port instance's required interface, then this propagates through to any provided interfaces on linked port instances.
6. PI_BROKEN_LINKS
   Two ports may not have an inferred link if following the connectors and links from one of those ports terminates on a port instance with a provided interface.
7. PI_INFERRED_INTERFACES
   The interfaces of a composite's ports can always be inferred from the connectors, and do not need to be specified.
8. PI_CHECKED
   If the interfaces of the port of a composite component are specified, they will be checked against the inferred interfaces, but only for the home stratum perspective.
9. PI_INFER_REQUIRED
   The required interfaces for each composite's port are inferred by traversing connectors and links from the port to incoming port instances only. The required interfaces is the set of lowest common sub-interfaces that satisfy these and have a one-to-one mapping.
10. PI_INFER_PROVIDED
    The provided interfaces for each composite's port are inferred by traversing connectors and links from the port to terminal port instances and other ports. The port's provided interfaces is the set of highest common super-interfaces that satisfy the provided interfaces of the port instances and the required interfaces of the ports, and have a one-to-one mapping.
11. WF_PI_SOME_INTERFACES
    Every port must have some provided and/or required interfaces.
12. WF_PI_NO_OVERLAP
    No two interfaces in a port's required interfaces may overlap. The same situation holds between provided interfaces.
13. WF_PI_PROVIDES_ENOUGH
    The provided interfaces for each port instance must provide enough for any port provided interfaces or any terminal port instance required interfaces that are reachable by following connectors and links outward from the port instance.
14. WF_PI_ONE_TO_ONE
    There must be a one-to-one mapping for the provided (required) interfaces of a port instance and any port provided (required) interfaces or port instance required (provided) interfaces that are reachable by following connectors and links outwards from the port instance.

B.4 Interesting Properties of the Specification

1. PROP_PARTIAL_ORDER
   Stratum dependencies form a partial order.

2. PROP_NO_CONFLICT
   Two elements which are newly defined cannot cause a conflict, regardless of which stratum they are defined in.
3. PROP_LEAF_IMMUTABLE
   A leaf component does not change with stratum perspective, as it cannot be redefined.
4. PROP_COMPOSITE_CHANGING
   A composite component or interface can change with stratum perspective, because of redefinition.
5. PROP_REDEF_PARTIAL_ORDER
   The order in which redefinition rewrites the resemblance graph takes into account the partial stratum order.
6. PROP_REDEF_CONFLICT
   A redefinition conflict occurs when multiple redefinitions from independent stratum must be merged, causing the rewritten resemblance graph to feature multiple resemblance.
7. PROP_REDEF_RESOLUTION
   Any redefinition conflict can be resolved through further redefinition.
8. PROP_RESEMBLANCE_CONFLICT
   A resemblance conflict occurs when using multiple resemblance, which can be resolved by adding, deleting or replacing constituents in the definition.
9. PROP_RESEMBLANCE_NO_SUPERSET
   It can be shown that the resemblance graph of a stratum is not always a superset of the resemblance graph of each strata it depends on. This is because a stratum may redefine a component that something else depends upon, in effect inserting an r-element into the middle of a resemblance graph.
10. PROP_REDEF_RESEMBLANCE
    Redefinition and resemblance can be used alone or in concert. When used together, it is possible to have incremental redefinition, where a component is defined in terms of deltas from its definition in another stratum.
11. PROP_DIAMOND
    Diamond resemblance does not result in multiple base elements.
12. PROP_INFERRED_LINKS_COMPLEMENTARY
    Two ports that are joined by an inferred link must have exactly complementary interfaces.

B.5 Definitions

B.5.1 Terms

Where appropriate, the terms are explained using the formal specification.

Stratum a is independent of stratum b if it is not transitively dependent on b.

```
b not in a.*dependsOn
```

Listing B.1: facts.als, lines 53-53

Two strata are mutually independent when neither has a transitive dependency on the other.

```
independent[a, b] and independent[b, a]
```

Listing B.2: facts.als, lines 58-58

Stratum visibility is unimportant for the definition of independence, as it is possible for one stratum to not be visible to another but to still contribute component and part definitions indirectly.

The top stratum represents a *single* stratum that brings together all the strata in a model that would otherwise be independent. The purpose of the top stratum is to ensure that we have a place to merge any competing redefinitions from independent strata. All strata are reachable by transitively following the dependencies of the top stratum.

```
one isTop.True
    ...
  isTrue[s.isTop] <=> no simpleDependsOn.s
```

Listing B.3: facts.als, lines 17-41

In actual architectures, there is sometimes an explicit top stratum that has been created as part of the application. In others, there is no explicit top and we must create an implicit one.

Each element is owned by a single stratum called its home, or owner.

```
abstract sig Element
{
  home: Stratum,
    ...
}
{
  -- owned by a single stratum
  home = ownedElements.this
}
```

Listing B.4: structure.als, lines 56-72

The scope of an interface is the full set of sub and super-interfaces (and the interface itself) present in the model.

```
i.*(superTypes.s) +  ^(superTypes.s).i
```

Listing B.5: inference_help.als, lines 109-109

Two strata overlap if their scopes intersect.

The presence of a one-to-one interface mapping is ensured by the oneToOneMappingExists[] predicate. An interface is constrained to provide enough by the provideEnough[] predicate which tests a single interface against a set of other interfaces for enough provision.

The stratum perspective of an r-element can change due to redefinition. To keep track of this, the specification determines the constituents of each element for each stratum that transitively depends on its home. For an element, this information is held in the delta structures, and also propagated higher into the component structure itself. For a component, for instance, it can be seen that the constituents depend on which stratum the perspective is being taken from.

```
sig Component extends Element
{
  ...
  parts: Part -> Stratum,
  ports: Port -> Stratum,
  connectors: Connector -> Stratum,
  attributes: Attribute -> Stratum,

  myParts: lone Parts/Deltas,
  myPorts: lone Ports/Deltas,
  myConnectors: lone Connectors/Deltas,
  myAttributes: lone Attributes/Deltas,
```

Listing B.6: structure.als, lines 74-91

Each delta is able to hold more than one constituent per UUID, to represent the redefinition conflicts that can occur when multiple independent redefinitions are merged. The addObjects, deleteObjects and replaceObjects of Deltas holds the add, delete and replace instructions respectively. A new UUID (ID parameterised type) can be introduced only when adding a new constituent.

```
sig Deltas
{
  ...
  -- newIDs are any new IDs added
  newIDs:                set ID,
  -- the deltas that are to be applied.  these 3 fields are the input to the
      merge
  deleteObjects:        set ID,
  addObjects:           newIDs one -> one addedObjects,
  replaceObjects:       ID one -> lone replacedObjects
```

Listing B.7: deltas.als, lines 5-35

After redefinition has been written as resemblance (ELEMENT_REDEF_RESEMBLE) then the deltas for each element are combined and possibly merged to create a perspective of that constituent for the given element, in a given stratum. The final constituents are held in objects_e.

```
objects_e: Stratum -> ID -> Object,
```

Listing B.8: deltas.als, lines 19-19

B.5.2 Stratum Rules

A stratum *s* is not involved in a dependency cycle if following transitive dependencies does not lead back to itself (STRATUM_ACYCLIC).

```
s not in s.^dependsOn
```

Listing B.9: facts.als, lines 32-32

Stratum visibility is managed by having each stratum keep track of which dependencies are exposed to other stratum which depend on them (STRATUM_VISIBILITY). If a stratum is not relaxed, it only exposes itself. If a stratum is relaxed, it exposes itself and any stratum that are exposed to it by the stratum it depends upon in turn.

```
isTrue[s.isRelaxed] =>
  s.exposesStrata = s + s.dependsOn.exposesStrata
else
  s.exposesStrata = s
```

Listing B.10: facts.als, lines 21-24

The rules for top stratum have already been shown in section B.5.1.

B.5.3 Element Rules

Components are prevented from accessing elements in stratum that are not visible to their home (ELEMENT_VISIBILITY). Interfaces have a similar constraint.

```
types.home in iCanSeePlusMe
attributeTypes.attrTypes in iCanSeePlusMe
interfaces.home in iCanSeePlusMe
```

Listing B.11: facts.als, lines 234-236

R-elements are prevented from resembling or redefining r-elements which are not visible (ELEMENT_VISIBILITY).

```
resemblingOwningStratum in owner.canSeePlusMe
redefiningOwningStratum in owner.canSee
```

Listing B.12: facts.als, lines 85-86

Redefinition is handled by rewriting all the resemblance graphs for each stratum perspective (ELEMENT_REDEF_RESEMBLE).

```
iResemble = topmostOfRedefined + topmostOfResemblances
```

Listing B.13: facts.als, lines 115-115

Multiple redefinition of the same element, viewed from the perspective of a stratum that merges these two definitions, will find that the rewrite produces multiple resemblance in keeping with the partial strata order. Figure B.4 shows such a situation, where W has combined X and Y which were previously independent.

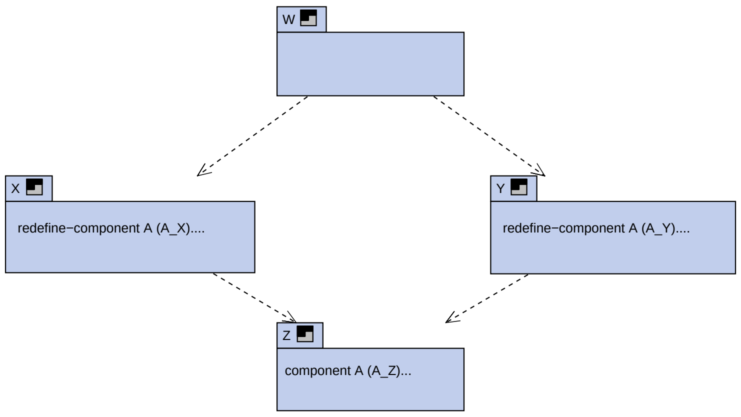

W
X
redefine−component A (A_X)....
Y
redefine−component A (A_Y)....
Z
component A (A_Z)...

Figure B.4: Independent redefinitions require a merge

The resemblance graph from W's perspective is rewritten to take account of the partial order of the stratum dependencies, and in this case multiple resemblance results. From the stratum perspective of W, A_W acts as the definition of A as shown in figure B.5.

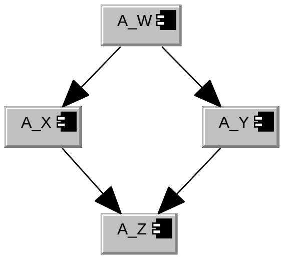


Figure B.5: Redefinition is turned into resemblance

After redefinition has been turned into resemblance for a given stratum perspective, the set of constituents can be determined for each element in that perspective. This is accomplished by the mergeAndApplyChangesForResemblance[] and mergeAndApplyChangesForRedefinition[] predicates.

When merging, it is possible that two r-elements independently replace the same constituent, and we may get two constituents for the same UUID. Another possibility is that through deletion, one constituent may refer to another by its UUID even thought that constituent is no longer present in the merged element. An element must have exactly one constituent for any required UUID in order to be well-formed. This condition is necessary, but not sufficient. For example, the component rules are as follows:

```
c.myPorts.oneObjectPerID[s]
c.myAttributes.oneObjectPerID[s]
    ...
isTrue[c.isComposite] =>
{
    ...
```

```
  c.myParts.oneObjectPerID[s]
  c.myConnectors.oneObjectPerID[s]
```

Listing B.14: wellformed_components.als, lines 29-40

When merging two branches of a resemblance graph, one branch may delete a constituent and another may replace it. In this case, the replacement has priority and the deletion will be ignored.

B.5.4 Interface Rules

An interface is a subtype of any interfaces it resembles, as long as none of the operations inherited are deleted or replaced (INTERFACE_SUBTYPE).

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

Listing B.15: facts.als, lines 158-161

B.5.5 Component Rules

Leaf components may not participate in resemblance or redefinition (COMPONENT_NO_LEAF_REDEF_RESEMBLANCE).

```
isFalse[c.isComposite] =>
{
      ...
  no resembles.c
  no c.resembles
```

Listing B.16: facts.als, lines 220-226

A well-formed component can never transitively contain itself (WF_COMPONENT_CONTAINMENT). To enforce this, we must take resemblance and redefinition into account, and recurse down the containment hierarchy. For instance, a part's type of a component cannot contain that component and so on.

```
let
  resembling = resembles_e.s, partTypes = parts.s.partType,
  original = no c.redefines => c else c.redefines
{
  original not in c.*(resembling + partTypes).partTypes
}
```

Listing B.17: wellformed_components.als, lines 19-24

B.5.6 Part Rules

A part that is replacing another (via redefinition) can remap its port instances onto the port instances of the previous part (PART_PORT_REMAP). This allows us to replace a part, but retain any connections to that part. To support this, each part has a remap structure which indicates how the new port instance UUIDs maps onto the old UUIDs. The information is consolidated into portMap, which holds all the UUID to port mappings for the part.

```
portRemap: PortID lone -> lone PortID,
portMap: Stratum -> PortID lone -> lone Port,
```

Listing B.18: structure.als, lines 162-163

We cannot have a part, or an island of connected parts which are not eventually connected back to the component (WF_PART_NO_ISLANDS). If this were allowed there would be no way to communicate with the parts. To prevent this from occurring, we travel from part to part looking for a direct connection to the component:

```
s -> c in pPart.*(linkedToParts.c.s).linkedToOutside
```

Listing B.19: wellformed_components.als, lines 66-66

B.5.7 Connector Rules

Port to port connectors are not allowed, as it would then not be possible to seed the inferencing algorithm when both port types were unspecified (WF_CONN_JOIN).

```
isTrue[c.isComposite] =>
  no portToPort
```

Listing B.20: port_inference.als, lines 169-170

B.5.8 Port and Port Instance Rules

All ports have a multiplicity, which may be as simple as *[1]* representing a single mandatory connection (PORT_MULTIPLICITY). The lower and upper bounds of a port are each represented as a finite set of indices.

```
mandatory, optional: set Index
```

Listing B.21: structure.als, lines 193-193

These sets are constrained to be contiguous and so that the lower bound is less than or equal to the upper bound. Finally, a port must have at least one possible index.

```
isContiguousFromZero[mandatory] and
  isContiguousFromZero[mandatory + optional]
no mandatory & optional       -- no overlap
```

```
some mandatory + optional     -- but must have some indices
```

Listing B.22: structure.als, lines 198-201

If a port instance only provides interfaces, connections to it are not enforced and can be zero to many (WF_PORT_PROVIDES_ONLY). If a port instance has some required interfaces, then it must have exactly one connection for a mandatory index, and at most one connection for an optional index.

```
(some o.required.c.s or isFalse[Model::providesIsOptional]) =>
{
  idx in o.mandatory =>
    -- must have a connection
    one ends
  else
    -- can only have at most one connection
    lone ends
}
```

Listing B.23: wellformed_components.als, lines 175-183

B.5.9 Port Type Inference Rules

The aim of the port type inference rules are to describe how the type constraints propagate between ports. Intuitively, if more is provided to a required interface on a port instance, then more can then be provided from a linked provided interface on a port instance linked to that port. This is supported by the use of links which propagate typing information between port instances (PI_PROVIDE_MORE) and the predicate providesEnough[].

Transitive closure is used in the Alloy specification to traverse the graphs formed by the connectors, links and parts of a composite.

To infer the required interfaces of a port, we find all ingoing port instances that we can reach from the port. The required interfaces is the set of lowest common sub-interfaces that satisfy these and have a one-to-one mapping. Provided interfaces are found by traversing to any terminal port instances or ports which are reachable from the port. The provided interfaces is the set of highest common super-interfaces that satisfy these and have a one-to-one mapping.

```
matchingRequires = extractLowestCommonSubtypes[s, requiresFromEnds],
      ...
matchingProvides = extractHighestCommonSupertypes[s, providesFromEnds]
{
  infReq = matchingRequires
  infProv = matchingProvides

  s -> c not in end.linkError <=>
    (oneToOneProvidedMappingExists[s, c, infProv, reqEnds] and
     oneToOneRequiredMappingExists[s, c, infReq, reqEnds])
}
```

Listing B.24: port_inference.als, lines 86-111

The required interfaces are not inferred from ports, whereas provided interfaces are. This has been found to be necessary for when a situation as in figure B.6 occurs. In this case, we can "bootstrap" the inferring of the port's required interfaces by using only the port instances. Then, the provided interfaces can be inferred using the port instances and the already inferred required interfaces.

Note that any part's type will either be a leaf in which case the port type information is available, or a composite in which case the port type information will have already been inferred.

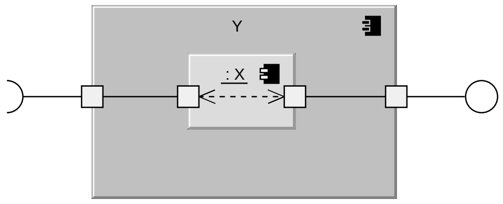


Figure B.6: The inference rules prevent ambiguity in common situations

Each port instance is checked to ensure that it provides enough to satisfy any required interfaces which are reachable from it. Finally, it is verified that each port instance matches one-to-one with the provided (required) and required (provided) interfaces of any port (port instance) that it is connected or linked to.

```
no end.partInternal =>
  providesEnough[s, infProv, matchingTerminalProvides]
oneToOneRequiredMappingExists[s, c, infProv, allEnds]
oneToOneProvidedMappingExists[s, c, infReq, allEnds]
```

Listing B.25: port_inference.als, lines 148-151

Rule PI_BROKEN_LINKS breaks an inferred link when the internal provision of an interface from a terminal port instance can be traced back to a port. That port may not then participate in any inferred links. To see why this is necessary, consider figure B.7. If ports P1 and P2 were allowed to have an inferred link, then provision of a sub-interface B to P1 would involve inferring P2 as providing B also. However, component Y can only provide A, resulting in a mismatch.

This rule is enforced when the inferred links are constructed by the following predicate logic:

```
no (end + other).*portToPort.linkPortID & terminateInternallyIDs
```

Listing B.26: port_inference.als, lines 249-249

B.6 Explanation of Properties

A leaf component is immutable because it cannot be redefined (PROP_LEAF_IMMUTABLE). Hence, a leaf's stratum perspective stays constant However, an r-element can change from one stratum perspective to another due to redefinition (PROP_COMPOSITE_CHANGING).

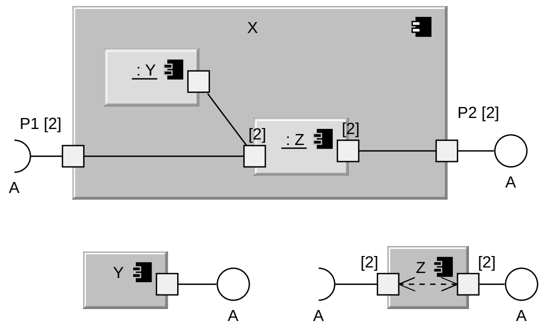


Figure B.7: Internal provision of an interface can break an inferred link

The dependency structure of strata form a partial order (PROP_PARTIAL_ORDER). This is apparent from the fact that it is possible to structure a set of stratum as in figure B.4. This implies that if the architecture includes two or more mutually independent strata that redefine the same element, then it will be necessary at some stage to merge the redefinitions into a coherent definition in at least one stratum perspective. This may be the top stratum perspective if no other stratum "merges" the two independent strata, according to rule STRATUM_TOP.

As redefinition is rewritten as resemblance, the resemblance graph will form a partial order in a similar way to the strata dependency graph, as shown in figures B.4 and B.5 (PROP_REDEF_PARTIAL_ORDER). This means that the rewritten resemblance graph may feature multiple resemblance.

Conflict can occur because a situation is created where two independent redefinitions of a structure conflict (PROP_REDEF_CONFLICT). For instance, one redefinition could delete a port, and another could independently add in a new connector to that port. When they are merged, the port will be deleted but the connector will still reference it (via its UUID). In this case, the connector must be removed or replaced. Redefinition can be used to correct any structural flaw resulting from conflict (figure B.8, PROP_REDEF_RESOLUTION).

Conflict can also occur when merges due to redefinition cause an overlapping stratum's definition to violate a well-formedness rule.

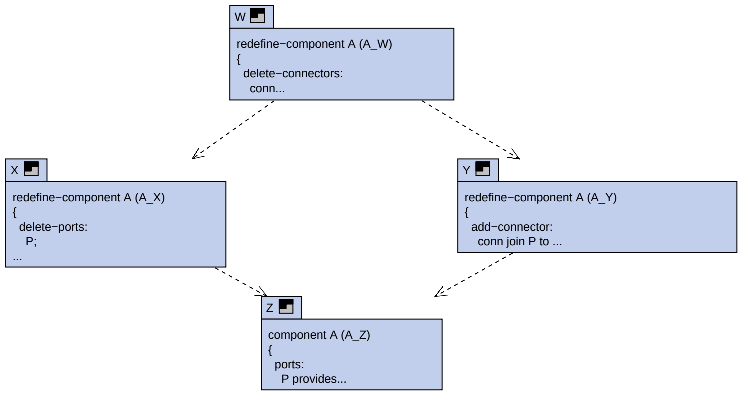


Figure B.8: Redefinition can always resolve redefinition conflicts

If a model contains no redefinitions of the same element in mutually independent strata, then the resemblance graph for the element can be linearised and does not require merging. To see this, consider if the dependencies in figure B.4 were restructured as in figure B.9.

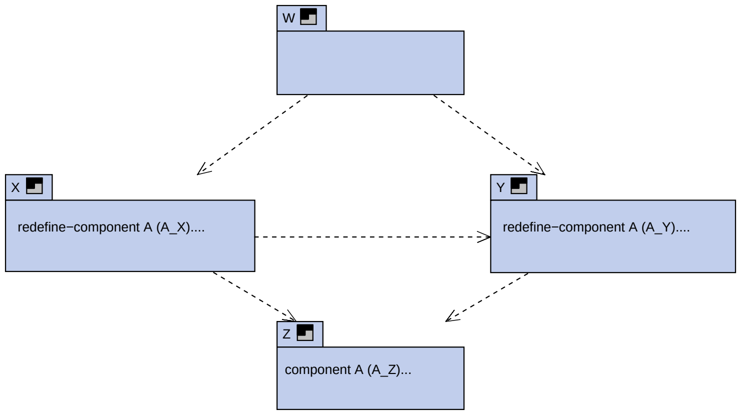

W
X
redefine−component A (A_X)....
Y
redefine−component A (A_Y)....
Z
component A (A_Z)...

Figure B.9: Merging is never necessary if the dependency graph can be linearised

The dependency link between X and Y causes the resemblance graph to be rewritten to indicate that A_X resembles A_Y which resembles A_Z (figure B.10). In effect, it is stating that the changes in X were made with full knowledge of the changes made in Y.

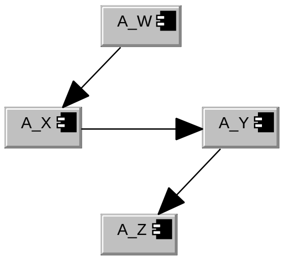


Figure B.10: A linear resemblance graph contains no conflicts

Two newly defined elements (i.e. not redefinitions) cannot cause a conflict because they can cannot refer to one another, and cannot impact any elements in overlapping strata.

It is possible for a design to incorporate multiple resemblance which causes a conflict (PROP_RESEMBLANCE_CONFLICT). However, due to the ELEMENT_OK_AT_HOME it is necessary to resolve any structural issues by adding, deleting or replacing constituents directly in the definition featuring the multiple resemblance.

Diamond shaped resemblance does not result in multiple base elements, as with non-virtual diamond inheritance in C++ [89]. The rules are defined to ensure that any element included from a base via multiple resemblance paths will lead to only a single copy of that element, and the use of UUIDs avoids inadvertent name clashes.

The PROP_INFERRED_LINKS_COMPLEMENTARY property is a consequence of the way that the port inference rules are defined for inferring provided and required interfaces. It also follows from PI_LINKS_COMPLEMENTARY.

In the early stages of creating the formal specification, it was envisaged that it would be possible to create a set of resemblance graphs representing the superset of every stratum perspective. However, this does not work when resemblance and redefinition can be mixed (PROP_RESEMBLANCE_NO_SUPERSET). Consider figure B.11 which contains both resemblance and redefinition.

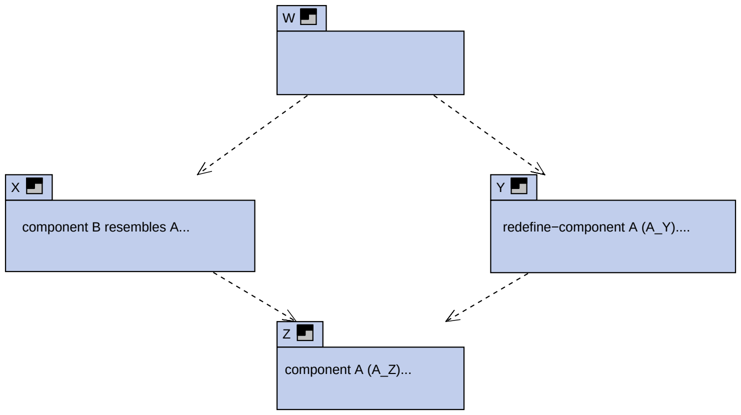


Figure B.11: Redefinition and resemblance

The resemblance graph from X's perspective is shown in figure B.12, and the graph from W's perspective is shown in figure B.13. They cannot be formed into a unified superset because the redefinition in Y has caused A_Y to be inserted into the resemblance graph between B and A_Z.

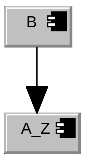


Figure B.12: Stratum perspective of X

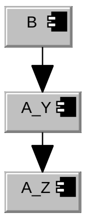


Figure B.13: Stratum perspective of W inserts A_Y between B and A_Z

As a consequence of this, the resemblance graphs for a stratum perspective must be determined separately for each stratum.

This example also demonstrates the utility of redefinition. It provides a principled way of inserting definitions into a resemblance graph, allowing for changes to be made at the correct point.

B.7 Additional Concepts

The following concepts are not described by the specification:

1. Placeholder
   A placeholder is an composite component without parts, which is used to describe the general shape of a future component in order to support top down development [61]. A component with parts of a placeholder type must have these parts replaced before it can be used in a running application. Placeholders can be redefined into normal composites, and resembled. It would be straight forward to extend the specification to allow this, but it was felt it would add little to the exposition.

2. Factory
   A factory is a composite component that can instantiate its parts lazily, and more than once if required. The major constraint that a factory would entail over a composite is ensuring that any ports have only optional indices, reflecting the possibility that the parts may not have been instantiated.

3. Delegate connectors
   Delegate connectors alias two ports together. They can be used to connect a port with an upper bound > 1 to a port instance with the same multiplicity, avoiding the need for many indexed connectors. In the specification, the number of possible indices is finite and as such a delegate connector can be represented by explicit connectors each with explicit indices.

Bibliography

[1] 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.

[2] J. Adamek and F. Plasil. Component composition errors and update atomicity: static analysis: Research articles. *J.Softw.Maint.Evol.*, 17(5):363–377, 2005.

[3] M. Alanen and I. Porres. Difference and union of models. In P. Stevens, Whittle.J, and J. Booch, editors, *UML2003*. Springer-Verlag, 2003.

[4] OSGi Alliance. *OSGi Service Platform: The OSGi Alliance*. Ios Pr Inc (December, 2003), 2003.

[5] OSGi Alliance. About the osgi service platform. *Website*, http://www.osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf, 2007.

[6] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe 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.

[7] W. Beaton. Eclipse hints, tips, and random musings. *Blog entry*, http://wbeaton.blogspot.com/2005/10/fun-with-combinatorics.html, July 2005.

[8] Jan Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41(5):257–273, 25 March 1999.

[9] D. Box. *Essential COM*. Addison-Wesley Professional, 1997.

[10] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[11] Kim B. Bruce, Leaf Petersen, and Joseph Vanderwaart. Modules in loom: Classes are not enough (extended abstract). Technical report, Williams College, 1998.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, and P. Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Number 1. John Wiley & Sons; 1 edition (August 8, 1996), 1996.

[13] Mozilla Developer Center. Firefox extensions. http://developer.mozilla.org/en/docs/Extensions.

[14] R. Chatley, S. Eisenbach, and J. Magee. Magicbeans: a platform for deploying plugin components. *Component Deployment*, 3083:97–112, 2004.

[15] P.H. 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.

[16] Rick Chern. Refactoring with difference-based modules: An experience report. In *CPSC 511 Mini Conference*, 2006.

[17] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. From custom applications to domain-specific frameworks. *Commun.ACM*, 40(10):70–77, 1997.

[18] Eclipse Consortium. Platform plug-in developer guide: Osgi bundle manifest headers. *Eclipse 3.2.1 Online Help*, http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse. platform.doc.isv/reference/misc/bundle_manifest.html, 2006.

[19] M. Cortes, M. Fontoura, and C. Lucena. Using refactoring and unification rules to assist framework evolution. *SIGSOFT Softw.Eng.Notes*, 28(6):1–1, 2003.

[20] B.J. Cox and A.J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[21] Scott Delap. Understanding how eclipse plug-ins work with osgi. *IBM Developerworks*, http://www.ibm.com/developerworks/library/os-ecl-osgi/index.html, 2006.

[22] DesktopLinux.com. Firefox 1.5 upgrade brings extension headaches. http://www.desktoplinux.com/news/NS2432314568.html, 2005.

[23] F. Doucet, S. Shukla, and R. Gupta. Typing abstractions and management in a component framework. In *Asia and South Pacific Design Automation Conference*, pages –, 2005.

[24] Imperial College DSE. The darwin language, version 3d. Technical report, Impertial College, 1997.

[25] Kayhan D. Sadler C. Eisenbach, S. 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.

[26] Gregor Engels, Reiko Heckel, Gabi Taentzer, and Hartmut Ehrig. A view-oriented approach to system modelling based on graph transformation. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–343, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[27] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. Software product line modeling made practical. *Commun. ACM*, 49(12):49–54, 2006.

[28] Clement Escoffier and Richard S. Hall. Dynamically adaptable applications with ipojo service components. In *6th International Symposium on Software Composition (SC 2007)*, 2007.

[29] M.E. Fayad and D.C. Schmidt. Object-oriented application frameworks. *Communications of the Acm*, 40(10):32–38, October 1997.

[30] 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.

[31] Brian Foote and Joseph Yoder. Big ball of mud. In *Fourth Conference on Patterns Languages of Programs*, 1999.

[32] M. Fowler. Inversion of control containers and the dependency injection pattern. *Website*, http://www.martinfowler.com/articles/injection.html, 2004.

[33] 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)*, pages –, 2003.

[34] 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.

[35] R. Harrop and J. Machacek. *Pro Spring.* Apress, 2005.

[36] J.D. Hay and J.M. 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.

[37] 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.

[38] 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*, pages 5–, Toronto, Ontario, Canada, 2001. IBM Press.

[39] Yuuji 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.

[40] Yuuji Ichisugi and Akira 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.

[41] Object Technology International Inc. Eclipse platform. *URL*, http://www.eclipse.org, July 2001.

[42] Object Technology International Inc. Eclipse platform technical overview. *Technical Report*, http://www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.

[43] D. Jackson. Alloy home page http://alloy.mit.edu/, 2005.

[44] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[45] R.E. Johnson. Frameworks = (components + patterns). *Commun.ACM*, 40(10):39–42, 1997.

[46] B.N. 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.

[47] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *conference proceedings on Object-oriented programming systems, languages, and applications*, pages 435–451, Vancouver, British Columbia, Canada, 1992. ACM Press.

[48] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[49] J. Kramer and J. Magee. The evolving philosophers problem - dynamic change management. *Ieee Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[50] K. Kramer, J. Magee, Keng N., and N. Dulay. *Software Architecture for Product Families: Principles and Practice*, chapter 2. Software Architecture Description, pages 31–65. Addison Wesley, 2000.

[51] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Association for Computing Machinery, 1993.

[52] Andreas Leicher. A framework for identifying compositional conflicts in component-based systems. Technical report, Computation and Information Structures, Technical University of Berline, 2004.

[53] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*,

volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[54] 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.

[55] J. Magee and J. Kramer. *Concurrency (State Models & Java Programs).* John Wiley and Sons Ltd, 1999.

[56] 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.

[57] M.D. McIlroy. *NATO Science Committee Report*, chapter Mass produced software components. NATO, 1968.

[58] A. McVeigh, J. Magee, and J. Kramer. Using resemblance to support component reuse and evolution. In *Specification and Verification of Component Based Systems Workshop (to be published)*, 2006.

[59] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Resolving structural conflict between independently developed, overlapping architectural extensions. *(Submitted for consideration) SAVCBS*, 2007.

[60] 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) on SIGSOFT '96 workshops*, pages 24–27, San Francisco, California, United States, 1996. ACM Press.

[61] N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. 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.

[62] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[63] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Ieee Transactions on Software Engineering*, 26(1):70–93, January 2000.

[64] E. Meijer and C. Szyperski. Overcoming independent extensibility challenges. *Commun.ACM*, 45(10):41–44, 2002.

[65] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, April 2005.

[66] Mira Mezini. Dynamic object evolution without name collisions. *Lecture Notes in Computer Science*, 1241:190–219, 1997.

[67] Microsoft. Com: Component object model technologies. *Website*, http://www.microsoft.com/com/default.mspx, 2006.

[68] Microsoft. Microsoft office online: Excel 2003 home page. *Website*, http://office.microsoft.com/en-gb/FX010858001033.aspx, 2006.

[69] SUN Developer Network. Enterprise javabeans technology. *Website*, http://java.sun.com/products/ejb/, 2006.

[70] SUN Developer Network. Javabeans. *Website*, http://java.sun.com/products/javabeans/, 2006.

[71] Keng 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.

[72] Magee J. Ng K., Kramer J. Subscribed content a case tool for software architecture design. *Automated Software Engineering*, 3:261–284, 1996.

[73] O. Nierstrasz. Regular types for active objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 1–15, Washington, D.C., United States, 1993. ACM Press.

[74] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Not.*, 39(10):99–115, 2004.

[75] OMG. Catalog of omg corba and iiop specifications. *Website*, 2004.

[76] OMG. Uml 2.0 specification. *Website*, http://www.omg.org/technology/documents/formal/uml.htm, 2005.

[77] OMG. Corba component model specification, v4.0. *Website*, 2006.

[78] OMG. Model driven architecture. *Website*, http://www.omg.org/mda/, 2006.

[79] J.K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[80] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans.Softw.Eng.*, 28(11):1056–1076, 2002.

[81] Sylvain Robert, Ansgar Radermacher, Vincent Seignole, Sébastien Gérard, Virginie Watine, and François Terrier. The corba connector model. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, pages 76–82, New York, NY, USA, 2005. ACM Press.

[82] Alef Arendsen Thomas Risberg Colin Sampaleanu Rod Johnson, Juergen Hoeller. *Professional Java Development with the Spring Framework.* Hungry Minds Inc, U.S., 2005.

[83] R. Roshandel, A. Van Der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276, 2004.

[84] B. Selic. Tutorial d: An overview of uml 2.0, 2003.

[85] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[86] A. Stuckenholz. Component evolution and versioning state of the art. *SIGSOFT Softw.Eng.Notes*, 30(1):7–, 2005.

[87] 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.

[88] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Melbourne, Australia*, pages –, Melbourne, Australia, 2006.

[89] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[90] R.N. Taylor, N. Medvidovic, M. Anderson, E.J. Whithead Jr., and J.E. 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.

[91] E. Truyen, B. Vanhaute, B.N. 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.

[92] 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.

[93] R. van Ommering. Mechanisms for handling diversity in a product population. In *ISAW-4: The Fourth International Software Architecture Workshop*, 2000.

[94] 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.

[95] R. van Ommering. Horizontal communication: a style to compose control software. *Softw.Pract.Exper.*, 33(12):1117–1150, 2003.

[96] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical report, Ecole Polytehnique Federale de Lausanne, 2004.

[97] Dingel J Zito A, Diskin Z. Package merge in uml 2: Practice vs. theory? *Model Driven Engineering Languages and Systems*, pages 185–199, 2006.