Resolving Structural Conflict Between Independently Developed, Overlapping Architectural Extensions

Andrew McVeigh, Jeff Kramer and Jeff Magee

Department of Computing
Imperial College
London SW7 2BZ, United Kingdom
{amcveigh, jk, jnm}@doc.ic.ac.uk

Abstract

Applications based on an extensible component architecture allow third parties to form extensions to them by connecting new components to existing extension points and replacing existing components. Plugin architectures are a prominent example of this approach and have proven useful as a way to allow others to add functionality to a framework or application.

An inevitable outcome of this approach is the desire to combine two or more independently developed extensions which extend the same application. Depending on the freedom allowed to the extension developer, this leads to a situation where the changes to the application made by one extension overlap with another extension's changes, resulting in structural conflicts that prevent their use together. This is part of a wider problem we call the overlapping extensions problem. We address the structural side of this problem in this paper.

The Backbone component model is introduced as an alternative to non-hierarchical plugin architectures, in order to ameliorate the overlapping extensions problem without restricting the freedom of the extension developer. Backbone uses a hierarchical model, allowing for unplanned change at the appropriate level in the architectural hierarchy. Conflicts are detected using well-formedness rules. Further, the same constructs that allow for unplanned changes can be used to resolve any structural conflicts and bring the combined extensions back into a coherent, unified architecture which can be structured to minimally conflict with any future extensions.

A formal specification of Backbone in Alloy is used to show how Backbone can describe the overlapping extensions problem and guarantee predictable merging. Further it is demonstrated that conflicts can be modelled and that the provided constructs can resolve any conflicts that occur.

1 Introduction

An increasing number of applications are built around an extensible component architecture. This allows third party developers to treat an application as a platform, and build on it by adding features. This is typically done by connecting new components to existing, planned extension points of the application's architecture. In some advanced plugin architectures, it is possible to replace arbitrary components from the application with new ones under certain conditions [6]. Many applications use a variant of this approach, including the Eclipse integrated development environment [10], and the Firefox web browser [4]. An extension is a set of components, packaged as a single entity, that extends the feature set of an application.

A natural consequence of this approach is that independent developers will separately create extensions by adding a set of components, and possibly altering the application's existing architecture through component replacement. If we then try to combine these independent extensions into a single architecture, it is often the case that they will conflict structurally due to overlapping and conflicting changes made to the base application. For instance, both extensions may try to replace the same component in the application (figure 1).

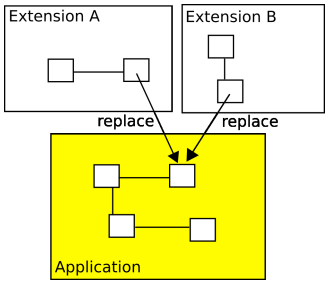


Figure 1: Structural conflict can occur between extensions

We call this potential for conflicts the *overlapping extensions problem*. This has a structural and behavioural side. In this paper, the focus is on the former.

The component model we use is called Backbone. We previously introduced Backbone in [14], where we showed that its constructs facilitate the reuse of complex components. In this paper, we show how Backbone can be used as an extension architecture and can detect structural conflicts between extensions. Further, it allows any conflict to be corrected. We use a formal specification written in Alloy to illustrate these properties.

One way to prevent overlapping extensions is to simply restrict extensions so that they cannot overlap. This implies that components from the existing application architecture cannot be replaced or modified. Although this is a common feature of most plugin architectures, this (often unacceptably) limits the features that a third party developer can add to those which have been pre-planned and are explicitly supported by the existing extension points. It is also possible to allow a limited form of replacement where the replacement component must be backwards compatible with the previous one. This is less restrictive on the types of extension possible, but does not solve the problem where two overlapping extensions replace or modify the same application component.

Our approach to dealing with the overlapping extensions problem is to employ a hierarchical component model along with constructs (redefinition and resemblance) that allow for arbitrary modifications to existing components. This grants an extension developer much flexibility, but offers no guarantee that overlapping extensions will not conflict. To address this problem, we show that the supplied constructs minimise the chance of conflicts by allowing fine-grained incremental modification at the appropriate level in the architectural hierarchy. Merging of independent redefinitions is guaranteed to be predictable and any conflicts are picked up using well-formedness rules. The supplied constructs can can also be subsequently used to resolve any structural conflicts.

In contrast, non-hierarchical extension architectures (we know of no hierarchical plugin architecture) only provide the choice of fully replacing an atomic component. In addition, the lack of a hierarchy and the need to present a manageable architecture tends to lead to more coarse-grained components. This further compounds the problem by magnifying the unit of possible conflict. Backbone avoids the need for this trade-off through explicit support for hierarchical composition.

The rest of the paper is organised as follows. The component model is briefly explained in section 2. An example is then described which shows how overlapping extensions can conflict, requiring resolution. The resolution is handled using redefinition. The formal specification of the component model is presented, focusing on how redefinition is rewritten as resemblance, with multiple resemblance resulting from independent redefinitions. Finally, the approach is summarised, related work is discussed and Backbone is contrasted with CM (configuration management) systems and other approaches to the problem.

2 The Component Model

The Backbone component model is a simplification of the UML2 composite structure model [16]. Backbone uses the UML2 composite structure diagrams for its graphical view, and has an equivalent textual view. The Backbone ADL (architecture description language), runtime interpreter, and supporting graphical case tool have been developed as part of this work, to explore the use of UML2 as a component definition language.

A component is a unit of software composition, which explicitly defines the interfaces it requires and provides via ports. Ports can be indexed by associating them with a multiplicity which has a lower and upper bound. A port may both require and provide interfaces. Leaf components cannot be further decomposed and are associated directly with an implementation class (figure 2). A component may have attributes which represent a projection onto the internal state of the component.

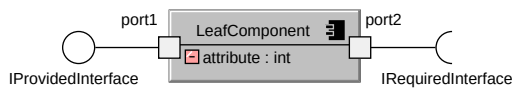


Figure 2: A leaf component

A composite component may contain parts, which are instances of other components (figure 3). Parts are wired together, or back to the component, using connectors. The parts of a composite represent its initial configuration.

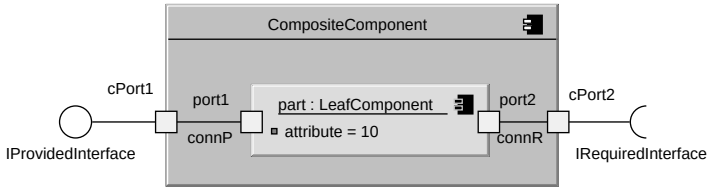


Figure 3: A composite component

For details on the textual language, refer to [14].

2.1 Resemblance and Redefinition

Backbone contains resemblance and redefinition constructs which facilitate the reuse of existing components and handle unplanned extension of an application.

Resemblance allows one component to be defined in terms of delta changes to another. One component may resemble one or more others, and then add, delete or replace any inherited constituents (ports, parts etc). They key point is that component definition is held as deltas, and these are only applied at runtime.

Redefinition allows a new definition to be substituted for an existing component. This does not destroy the original definition, but substitutes the new definition wherever the component is used, if the redefinition is included in the architecture. When combined with resemblance, we call this incremental redefinition and it can be used to evolve the definition of a component in an application to support a new feature.

The example in the next section shows the practical use of these constructs, illustrating how the freedom that they offer can cause conflicts. Any conflicts can further be rectified using the same constructs.

3 Motivating Example

The following example has been heavily simplified from the author's experience as one of the architects of a commercial digital radio console. This was designed to manage a set of audio devices for a radio studio.

3.1 A Digital Audio Desk Application

The application is a digital audio desk, presented as a composite component (figure 4). Its function is to control multiple audio devices, and integrate their audio into a single output via a software mixer.

No devices are configured into the desk by default. Devices should be added between the *in* and *control* ports. The control port has a multiplicity of *[0..*]* which means that any number of devices can connect to it. The desk is composed of a mixer part.

The mixer (figure 5) is a composite component which accepts audio packets from any number of devices via the in port. It mixes the audio together and outputs the audio via the out port.

Note that these composites have internal parts and connections, but these are not shown in order to simplify the presentation.

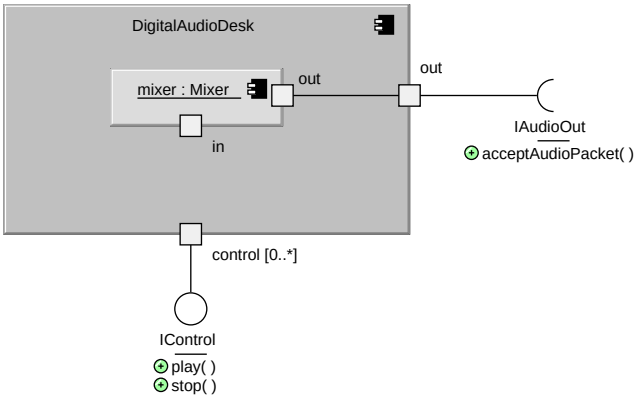


Figure 4: The digital audio desk component

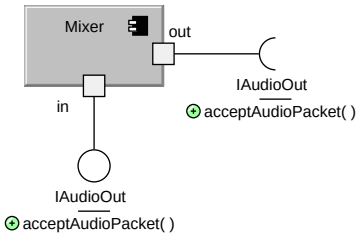


Figure 5: The software mixer component

3.2 Adding Support for a CD Player

Developer A is asked to use an existing CDPlayerDevice component and integrate it with the audio desk. This composite (figure 6) acts as the controller for a physical CD player. It supports control via the IControl interface, but also supports cueing via the ICue interface. Cueing a device plays it “off-air”, allowing a presenter to find the right point of a song to start playing from.

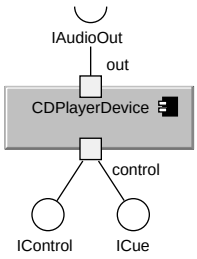
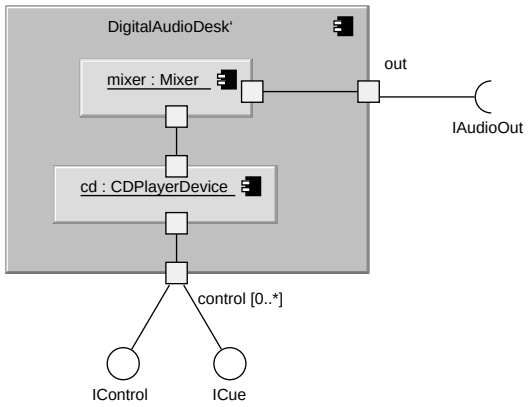


Figure 6: The CD controller component

An obvious issue with integrating this with the desk component (figure 4) is that the desk currently offers no support for cueing. However, the developer cannot just modify the source code for the desk component, as this is used by many other companies as the basis for their products. In reality, the developer may not even have access to the source if this is a commercial application.

Using incremental redefinition however, it is possible to make changes to the desk component, but store the deltas separately from the original definition (figure 7). The ICue interface is added to the desk, and at the same time the CD controller is wired in. The textual Backbone redefinition is shown under the graphical view. Note that in the graphical view of the redefinition, the name of the redefinition is shown with a prime (DigitalAudioDesk’) reflecting the evolution of the component.




```
redefine-component DigitalAudioDesk
  resembles [previous]DigitalAudioDesk
{
  replace-ports:
    control [0..*] provides ICue, IControl;
  parts:
    CDPlayerDevice cd;
  connections:
    cd joins control@cd to control[+];
    cd-mixer joins out@cd to in@mixer;
}
```

Figure 7: Redefining the desk to support a CD player

The redefinition is phrased in terms of resemblance. It represents a delta change to the previous definition, consisting of three additions and one replacement.

The architectural changes need to be packaged into a single extension and applied somehow. In Backbone this is done using a stratum. A stratum is a module-like construct which is used to group a set of definitions and redefinitions. If the original desk application is supplied in stratum Desk, then the extension must be supplied in another stratum. Stratum CD contains the CD player definition and the desk redefinition, and depends on the definitions in stratum Desk (figure 8).

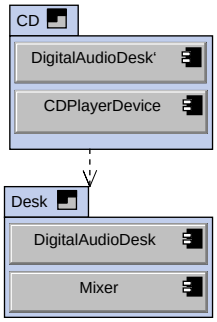


Figure 8: The redefinition of desk is contained in the CD stratum

The Backbone interpreter is given a set of stratum to run, and will apply any redefinitions at start-up time. If both strata are chosen, then the desk will contain a CD player component because the redefinition will be applied. However, if only stratum Desk is chosen, then it will not contain the player, and the original application will run.

3.3 Adding Support for a Digital Microphone

Developer B is independently tasked with integrating an existing DigitalMicDevice component with the desk. This component controls a digital microphone. Unlike the CD player, no cue support is available.

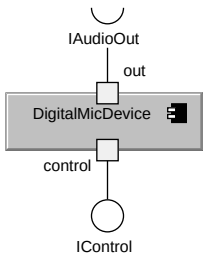
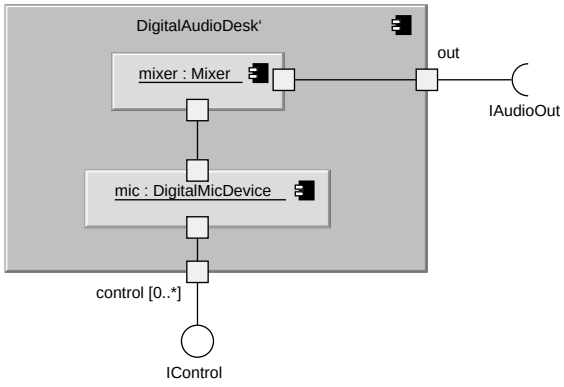


Figure 9: The digital microphone controller

The microphone can be wired into the desk using a redefinition of DigitalAudioDesk to add the microphone and the new connectors (figure 10).




```
redefine-component DigitalAudioDesk
  resembles [previous]DigitalAudioDesk
{
  parts:
    DigitalMicDevice mic;
  connectors:
    mic joins control@mic to control[+];
    mic-mixer joins out@mic to in@mixer;
}
```

Figure 10: Wiring the microphone into the desk

The microphone definition and desk redefinition are packaged into a stratum Mic that depends on the Desk stratum (figure 11).

3.4 Creating a Unified Application

Unsurprisingly, someone eventually wishes to have a desk with both CD and microphone support. This involves merging the changes made to the desk together into a single view. This merge is simplified by the fact that the changes are held as fine-grained deltas.

Figure 12 shows what would happen if the deltas were all combined.

This has produced a structural conflict, which will be picked up by the Backbone well-formedness rules. The desk's control port provides IControl and ICue, whereas the DigitalMic's control port only provides IControl. This is because the redefinition

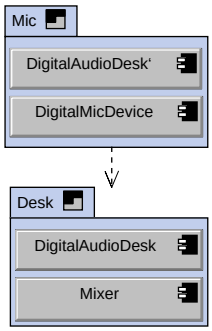


Figure 11: The redefinitions are contained in stratum Mic

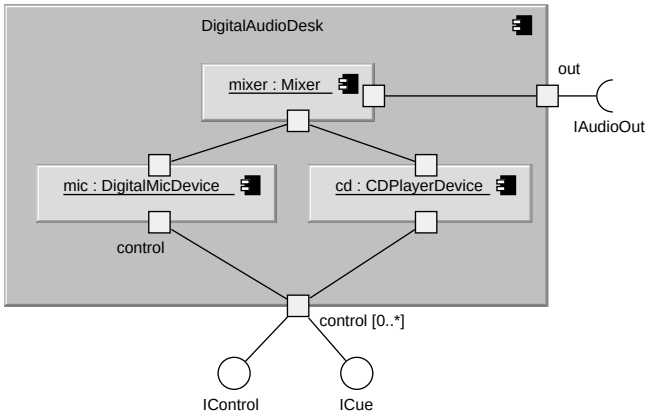


Figure 12: A structural conflict results from combining the extensions

in the CD stratum added ICue to the desk's port to support the CD's cue facility. We cannot use both the CD and microphone together without correcting this somehow.

3.5 Resolving Conflicts

Structural conflict is a common occurrence when combining independently developed, overlapping extensions. Giving the freedom to add new features to existing components also provides the possibility for one extension to be made incompatible with another.

Limiting the types of features that can be added (i.e. preserving backwards compatibility both internally as well as externally) or preventing existing components from being modified will prevent the possibility of structural conflict. Predictably though, this constrains the types of features that can be added to those which the designer of the original application had planned for. In the previous example, it would not have been possible to add cueing support until this was added into the original application. If the developers of the original application are unwilling to add the changes or cannot because the upgrade will cause too great an impact upon existing users, then the features can never be added.

The Mic and CD strata overlap in that they both depend on the Desk stratum. They are independent because the stratum dependencies do not allow them to refer to each other. To resolve the conflict between the two extensions, we redefine the desk component and package this in the CDAndMic stratum (figure 13).

The redefined desk separates the ports into control and control-

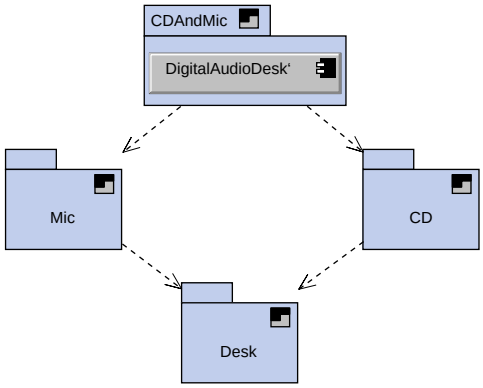
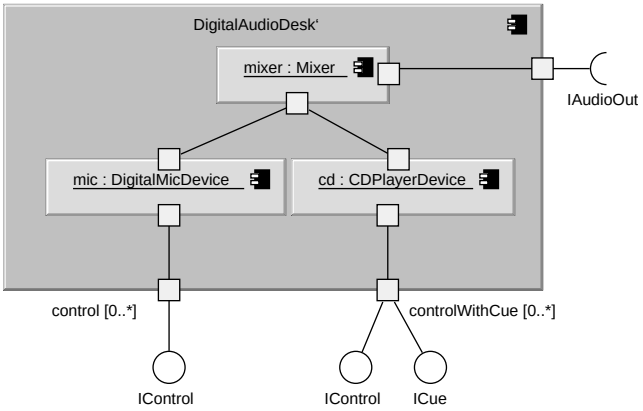


Figure 13: Merging the CD and Mic stratum

WithCue (figure 14). The connector that used to join the CD control port has been rerouted to connect to the controlWithCue port.




```
redefine DigitalAudioDesk
  resembles [previous] DigitalAudioDesk
{
  ports:
    controlWithCue [0..*] provides IControl, ICue;
  replace-ports:
    control [0..*] provides IControl;
  replace-connectors:
    cd joins control@cd to controlWithCue[+];
}
```

Figure 14: Correcting the structural flaw in the combined desk

Since redefinition can be used to arbitrarily remake a component, all structural flaws can be corrected. The key is to keep the conflict-resolving deltas as small as possible so that they will be less likely to conflict with any further merges that may be required when adding later features.

Why is redefinition used at all, when it is possible to create DeskWithMic and DeskWithCDPlayer using resemblance? Even if this were done, adding the devices into the same desk will involve multiple resemblance (DeskWithMicAndCDPlayer) which will lead to exactly the same structural conflicts as presented already.

Another use of redefinition is to allow us to make changes to a component whose instantiation we have no control over. For instance, the application may be "hardwired" to run by instantiating an instance of DigitalAudioDesk, in which case we have no choice but to redefine it.

4 Formal Specification

The structural side of the Backbone component model has been formally specified using Alloy [11], and is available at www.doc.ic.ac.uk/~amcveigh/papers/savcbs-2007.html. Alloy is a modelling language based on first order logic, which is supplied with a model finder which can check assertions and find counterexamples within a finite model space.

Note that this is the specification for the Backbone component model, illustrating general properties including predictable merging of extensions. It is being used to develop the behaviour and rules governing the runtime interpreter. No actual Alloy model is constructed for each architecture described using Backbone. All the well-formedness rules in the interpreter are specified in the formal model.

The essence of the formal model is that resemblance graphs are rewritten to take redefinition into account. This must be done from the perspective of each stratum. Conflicts occur when multiple branches of a single resemblance graph incompatibly add, replace or delete component constituents. Predictable merging is handled by the multiple resemblance rules.

4.1 Overlapping Strata and Extensions

A stratum owns component and interface definitions and redefinitions. It must explicitly declare any other strata it depends upon. The dependencies constrain the elements that components and interfaces (owned by that stratum) can refer to.

```
sig Stratum
{
  -- strata that this directly depends on
  dependsOn: set Stratum,
```

Listing 1: structure.als, lines 19-22

The dependencies of a stratum must be acyclic.

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

Listing 2: facts.als, lines 32-32

Stratum *stratum1* is independent of stratum *stratum2* if it does not transitively depend on it.

```
stratum2 not in stratum1.*dependsOn
```

Listing 3: facts.als, lines 53-53

Two extensions are considered to be overlapping if they share common dependencies (representing the base application), but are mutually independent. This is the case between CD and Mic in figure 13. Two stratum have the *potential* to conflict if they overlap. In this case, the overlap is the Desk stratum.

Strata dependencies define a partial order. Both {CDAndMic, CD, Mic, Desk} and {CDAndMic, Mic, CD, Desk} are valid orders and the model must ensure that either order results in the same architecture and conflicts when the extensions are merged.

4.2 Redefinition Rewrites the Resemblance Graph

The *perspective* from a given stratum is the application that would be formed if only the definitions from that stratum and

any that it transitively depends upon are included, and all others were ignored. The perspective of a stratum is the reflexive transitive closure of its dependency graph:

```
stratum.*dependsOn
```

Listing 4: facts.als, lines 63-63

Redefinition is turned into resemblance by rewriting the resemblance graph for each component, for each stratum perspective. For example, from perspective CDAndMic, the resemblance graph of the desk component is shown by figure 15. Each component is prefixed by its stratum name, and resemblance is shown as a large arrow between components.

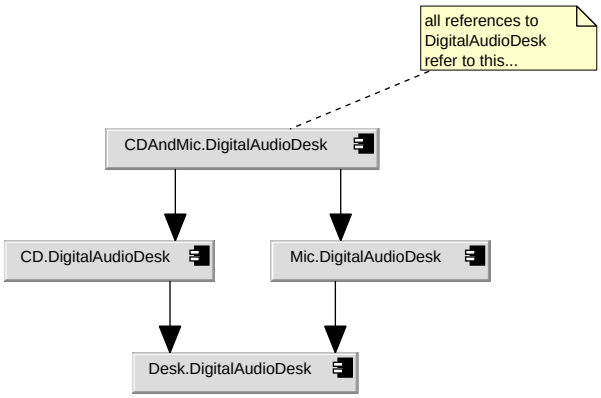


Figure 15: The rewritten resemblance graph of desk

The rewritten graph contains multiple resemblance, and defines a partial order consistent with the partial order of the strata dependencies. The topmost component now acts as the DigitalAudioDesk component – any reference to the desk component will refer to this. This is the effect of being the redefinition at the top of a resemblance graph.

The rewriting of the resemblance graph for each perspective is performed by the following logic:

```
all s: Stratum |
let
  -- who should I resemble
  -- (taking redefinition into account)
  iResemble = e.resembles_e.s,
  -- if we resemble what we are redefining,
  -- look for the original under here
  topmostOfRedefined = getTopmost[
    owner.simpleDependsOn,
    e.redefines & e.resembles],
  -- look for any other resembled components
  -- from here down
  topmostOfResemblances = getTopmost[
    s,
    e.resembles - e.redefines]
{
      ...
    iResemble = topmostOfRedefined +
        topmostOfResemblances
}
```

Listing 5: facts.als, lines 92-116

`iResemble` holds the components that the component `e` resembles, from the perspective of stratum `s`. This is determined by the `getTopMost[]` function which uses the stratum dependency order to find any redefinitions and resemblances that must be taken into account when rewriting the graph. As an example, from the perspective of CDAndMic, the redefinition in this stratum will find the redefinitions in the CD and Mic strata.

4.3 Merging

To expand out a component definition, we must applying all of the deltas in the resemblance graph. This is performed by operations associated with the `Deltas` signature, which is defined in a parametrised Alloy module. It represents the deltas held by a definition or redefinition. For a component, we have the following possible deltas:

```
sig Component extends Element
{
  ...
  myParts: lone Parts/Deltas,
  myPorts: lone Ports/Deltas,
  myConnectors: lone Connectors/Deltas,
  myAttributes: lone Attributes/Deltas,
```

Listing 6: structure.als, lines 74-91

Each delta holds a set of tuples (`Stratum -> ID -> Object`) of a component, where `Object` represents either `Port`, `Part`, `Attribute` or `Connector`. This is an `(ID -> Object)` mapping from the perspective of each stratum. `ID` may be thought of as synonymous with the concept of a name, although in practice, the graphical case tool uses a mapping layer to associate human readable names with IDs. IDs are generated artifacts that are guaranteed not to conflict between independent extensions.

```
sig Deltas
{
  ...
  objects_e: Stratum -> ID -> Object,
```

Listing 7: deltas.als, lines 5-19

If, for example, two branches of a resemblance graph replaced the same port P, then the merged result would hold both ports against the single name P. As we only want one constituent per name, a key well-formedness rule is the following:

```
pred Deltas::oneObjectPerID(s: Stratum)
{
  let objects = this.objects_e[s] |
    function[objects, dom[objects]]
}
```

Listing 8: deltas.als, lines 55-59

This ensures that the `objects_e` relation is a function with at most one constituent per name for a given stratum. Merging is predictable because independent redefinitions replacing the same constituent will result in several constituents for a given name and will not overwrite each other based on a partial order. This can only be corrected through a further replace or delete.

4.4 Resolving Conflict

The delta operations of add, replace and delete allow any conflict in a multiple resemblance graph to be resolved. If there are two or more constituents for a given name, then replace is used to replace both with a new constituent. Delete removes any constituents with a given name, and add provides a new constituent for a new name.

This logic is embodied in the following listing:

```
this.objects_e[s] =
  (iResembleDeltas_e.originalObjects_e[s] -
      this.deletedObjects_e[s]->Object)
    ++ this.replacedObjects_e[s]
```

Listing 9: deltas.als, lines 157-159

`iResembleDeltas_e[s]` is a set holding the components that are under a given component in the rewritten resemblance graph, from the perspective of stratum `s`. The logic takes the original objects (`originalObjects_e`), removes any names and constituents that have been deleted, and replaces all constituents with a given name by `replacedObjects`, using the Alloy override operator (`++`).

It is interesting to note that even in simple cases, the nature of a conflict produced by an overlap can be subtle. In figure 12, the multiple redefinition of the desk caused a structural issue with the microphone component rather than a conflict directly in the desk component.

5 Related Work

Eclipse supports a plugin model, based on OSGi [1] where components (OSGi bundles) indicate how they are connected to the service provisions of other components using a set of manifest files [5]. The model is non-hierarchical in that 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. Structural conflict is a problem in this model, although in practice it is restricted because replacing individual components in a configuration is difficult and not often done [2].

Firefox uses both the terms plugin and extension. Plugins are intended to display extra content to the screen in an extensible way. Extensions are synonymous with the extensions discussed in this document. They are a way to extend the functionality of Firefox and customise the feature set. Versioning of extensions and assessing compatibility with previous versions of the browser represents an ongoing concern [7].

COM is a component model and infrastructure built into the Windows operating system [3]. It forms a common plugin architecture for many applications including the Office suite applications such as Excel [15]. COM supports a hierarchical model, and composition of instances is via a registry based approach for indirectly locating service providers. This model does not focus on supporting or resolving overlapping extensions, 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" [8].

Backbone is related to ADLs such as Darwin [12]. The core of Backbone is very similar to the core of Darwin, even though Backbone was initially developed independently. This presumably reflects the influence that Darwin and other ADLs had on the UML2 specification on which Backbone is based.

Koala [19] is a component model based on Darwin that allows for variation in an architecture through variation points and parametrisation. Component variants can be plugged into the variation points, supporting a family of applications. The points must be decided in advance and planned into the architecture, limiting this to a technique for planned extension.

Mae is an architecturally-aware CM system that understands how deltas can be merged into a new revision of an architecture [17]. This approach presumes the use of a central version control system. Further, if many extensions are made then the central version control graph will get very complex. Backbone avoids this by effectively providing 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 modelling foundation with explicit modelling 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.

The Backbone resemblance construct is similar to the structural inheritance facility provided by ROOM [18]. Backbone extends this by adding support for redefinition, in order to model the arbitrary evolution of components.

The MixJuice system adds a module system to Java, supporting a variant of redefinition. Inheritance is used in place of resemblance [9]. The intention is to allow an object-oriented system written in Java to be extended in unplanned ways. Resemblance is more powerful than inheritance as it allows constituents to also be deleted and replaced, reflecting their evolution. Further, MixJuice relies on a total order to be specified, although it can use implicitly included "complementary" modules to resolve several common types of conflict. Because it relies on Java, MixJuice also does not offer a true architectural model.

6 Conclusions and Future Work

Backbone offers a compelling, architecturally-focused model for component extension architectures. It explicitly supports composite components and composition, unlike conventional plugin architectures, allowing an extension to replace a component at the appropriate level in an architectural hierarchy. The redefinition and resemblance constructs offer the ability to arbitrarily modify a base application, allowing new features to be added. These constructs guarantee predictable merging when combining overlapping extensions, and also allow any structural conflicts to be resolved by adding another extension.

The Backbone model also allows interfaces to be redefined and resembled. Both components and interfaces are handled in the same way, as they are both sub-signatures of Element in the formal model.

The Backbone ADL, runtime interpreter and UML2 case tool have been implemented as part of this work. The formal specification of the structural model has been completed, and has shown that the existing interpreter currently produces inconsistent results with some partial strata orders. We are now reimplementing this part of the interpreter to follow the results of the formal model. Similarly, the case tool is being upgraded to support the results also.

Even if structural conflicts have been resolved, it is still possible for extensions to behaviourally conflict. We have already started modelling component protocols with FSP, a process algebra [13]. The aim is to detect protocol errors when combin-

ing extensions by composing together the FSP protocols and looking for violations of generated safety properties.

It is possible to statically express the evolution of an architecture using resemblance and redefinition. When this is done for multiple versions of a system, the result is multiple "chained" strata expressing the changes to a base application as deltas. This gets difficult to work with, and we are investigating a construct called *baselining*, which merges multiple redefinitions with the base application, creating a new, evolved base. An interesting property of this is that it is possible to produce extensions, which when applied to the evolved base create *previous* versions of the application. This allows legacy versions of the application to be reconstituted and selectively combined with newer features.

References

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

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.

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

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

[17] 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.

[18] B. Selic, G. Gullekson, and P.T. Ward. Inheritance. In *Real-Time Object-Oriented Modeling*, volume First, pages 255–285. Wiley, 1994.

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