Using Resemblance to Support Component Reuse and Evolution

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

The aim of a component-based approach to software is to allow the construction of a system by reusing and connecting together a number of existing components. To successfully reuse a component, alterations generally need to be made to it, particularly if the abstraction level is high. However, existing usage of a component means that it cannot be altered without affecting the systems that reuse it already. This leads to a dilemma which frustrates the goals of the compositional approach to reuse.

To help resolve this dilemma, we introduce the resemblance construct, allowing a new component to be defined in terms of changes to a base component. This allows us to effectively alter a base component for reuse, without affecting the existing definition or any users of the component. We use an example to show how this and other constructs ameliorate the reuse problems of complex, possibly composite, components.

1 Introduction

When taking a compositional approach to system construction, a composite component can be created by composing and connecting together a number of other components. Each of the constituent components of the composite may either be composite themselves or leaf components which have no further decomposition [19, 8]. Complex subsystems, and even entire systems can be represented as composites which can then be reused as parts of other systems. The aim is to assemble systems from increasingly higher-level components, offering a compelling approach to construction and reuse. In practise, however, a number of issues frustrate this goal.

To set the context, consider that a system is constructed from both existing components, and new components developed specifically for that architecture. Existing components are obtained from a component provider, or taken from an existing system. It is unlikely that changes can be made to an existing component specifically to accommodate a new system, as existing usage in other environments places constraints on what can be changed. To be successfully reused, however, existing components generally require alterations before they can be integrated into a new architecture [5].

This situation leads to a dilemma: components cannot be reused without changes, but existing usage heavily constrains any changes. The more complex or higher-level a component is, the less is the likelihood that it will be suitable for reuse in an unaltered form. This situation is closely related to the abstraction problem [4]: components are more valuable when they represent higher-level abstractions targeted at a particular domain, but this specificity limits their reuse. This is particularly a problem with composite components as they hide their constituent components and abstractions.

In order to examine this dilemma more closely, a reuse scenario from an existing system is presented. By analysing this situation, we form a set of requirements that a solution must meet in order to address the identified issues.

From these requirements we develop the concept of *resemblance*, which is an inheritance-like construct for components. This allows us define a new component in terms of changes to a, possibly composite, base component. The key is that the changes are held in the new component, and do not affect the base definition. Combined with a small number of other constructs, we demonstrate how this ameliorates the reuse problems. We further show how the constructs can also help with component evolution, acting as a type of decentralised configuration management (CM) system.

The rest of the paper is organised as follows. We begin by presenting the component model as general background for the discussion and to establish terminology. A simplified example of a component reuse problem from a working system is shown, leading to a conceptual view of the problem. We then introduce the resemblance and other constructs and show how a component can be altered for new requirements, without losing the link back to the original definition. We conclude with a discussion of related work which contrasts this work with architecturally-aware CM systems, and product family approaches.

2 The Component Model

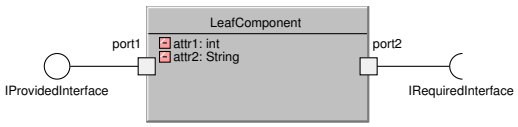
In keeping with Darwin [9] and UML2 [11], we define a component as an instantiable, class-like construct which explicitly describes the interfaces that it provides and requires. An interface represents a collection of methods defining a service and may inherit from other interfaces. Interfaces can only be provided or required via ports, and each port has a name and may be indexed. 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, on 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 1 shows a leaf component with two attributes and two ports. The graphical representation is a UML2 composite structure diagram where a provided interface is shown as a circle,

and a required interface is shown as a semi-circle. Note that the leaf is directly associated with a Java implementation class.



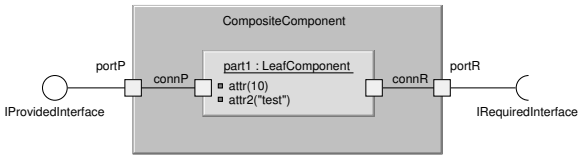

```
component LeafComponent
   describes com.example.JavaLeafComponent
{
   attributes:
      int attr1; String attr2;
   ports:
      port1 provides IProvidedInterface;
      port2 requires IRequiredInterface;
}
```

Figure 1: Definition of a leaf component

The textual definition in the lower half of the figure is from the Backbone ADL. This experimental language has been defined, as part of this work, in order to demonstrate the concepts in this paper, and to also explore the use of UML2 as an ADL. We have developed a prototype Backbone interpreter which can assemble a system from the ADL representation and the Java implementation of the leaf components.

Although Backbone has been designed around the UML2 component meta-model, it bears more than a passing resemblance to Darwin. This presumably reflects the influence that Darwin, ROOM [14], ACME [2] and other ADLs have had on the UML2 specification.

A composite component (figure 2) can additionally contain a number of component instances, each of which is shown as a box within the component. These instances are called *parts* in UML2 terminology. Each part has a name (part1), and a component type (LeafComponent) which is the component it is an instance of. Further, a part can define slots, which hold values for the attributes of the component type e.g. attr(10). The parts of a composite represent its initial configuration and state.




```
component CompositeComponent
{
   ports:
      portP provides IProvidedInterface;
      portR requires IRequiredInterface;
   parts:
      LeafComponent part1
         set attr1(10), attr2("test");
   connectors:
      connP joins portP to port1@part1;
      connR joins portR to port2@part1;
}
```

Figure 2: Definition of a composite component

Ports are wired together using connectors (connP, connR). In UML2, connectors represent little more than an aliasing of two different ports [3].

3 Motivating Example

This example is based on a reuse problem experienced when extending our graphical modelling tool. This tool is being developed as part of this work to provide an environment to support the concepts outlined in this paper.

As we work through the example, we use it to distill four requirements that a solution to the component reuse problems must address.

3.1 Context

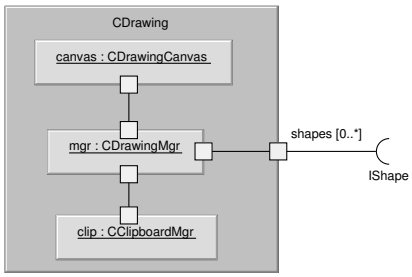
Company X is a component provider that produces components for constructing graphical drawing tools. The major component is a composite called CDrawing, which represents a drawing framework.

Also available are a set of components which can draw various complex shapes when used with the framework. One such component is CPostItNote, which displays a small note surrounded by a border as shown in figure 3.

Figure 3: Post-it notes displayed in a drawing

X sells these components to third parties, providing an ADL representation for each component, along with the Java interface definitions, but not the Java implementation source code. X maintains the components using its own CM systems, and periodically releases new versions, which aim for backwards compatibility. A major aim has been to make the drawing components as reusable as possible. However, due to the large number of customers using the components, changes cannot be specifically introduced for one customer's system.

The definition of CDrawing is shown in figure 4. It is a composite component with parts to handle clipboard functionality and a drawing canvas. An indexed port is used to hold the list of shapes, which are used for drawing the display. This is shorthand for a set of ports: shape[0], shape[1] and so on.



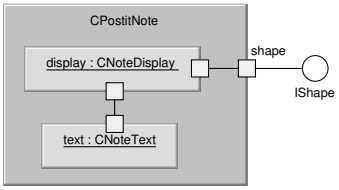

```
component CDrawing
{
   ports:
      shapes[0..*] requires IShape;

   parts:
      CDrawingMgr mgr;
      CDrawingCanvas canvas;
      CClipboardMgr clip;

   connectors:
      shapes joins shapes to shapes@manager;
      canvas joins surface@canvas to view@manager;
      clip joins board@clip to view@manager;
}
```

Figure 4: Definition of CDrawing

The composite CPostitNote component (figure 5) is designed to work with the framework by providing the IShape interface. The CNoteDisplay part handles the display of text on the screen and the word wrapping. The plain text is stored in the CNoteText part.




```
component CPostitNote
{
   ports:
      shape provides IShape;

   parts:
      CNoteDisplay display;
      CNoteText text;

   connectors:
      shape join shape to shape@display;
      text joins data@text to text@display;
}
```

Figure 5: Definition of CPostitNote

3.2 Reuse Scenario

Company Y now wishes to reuse the CDrawing and CPostitNote components to construct a desktop tool for taking notes. For this task, the clipboard is not needed and Y wishes to omit this facility to minimise the size of the application. In addition, CPostitNote must support hyperlinks and the CDrawing component must support changing the zoom level. Although this is a simple scenario, it shares many of the characteristics of real-world reuse situations.

Y clearly must make changes in order to reuse the existing components, leading to our first requirement:

Alter It should be possible to alter a component to allow it to be reused into a new system. The changes required may be extensive.

In our scenario, Y contacts X and suggests that X makes the changes, or at least provide variation points to make the incorporation of the features possible. However, the provider does not wish to alter the components, as this would require a major change for existing customers. In addition, if this courtesy was extended to all reusers, the architecture would quickly descend into a generic mess with variation points for every conceivable option.

This leads to our next requirement:

NoImpact Alterations to a component for reuse must not impact existing users of the component. Further, the alterations should not impose an obligation on the provider to accept or even know about the changes.

At any rate, the alterations required for reuse are often specific to the new application, and cannot easily be generalised for incorporation into a generic component. In this sense, the alterations fall into the same category as glue code which often has to be written to adapt a component for reuse in a new context. Like glue code, the alterations belong with the system where the component is being reused, not with the original component definition.

Subsequently, Y performs an analysis and decides that its requirements could be met by omitting (or stubbing out) the CClipboardMgr part from CDrawing, upgrading the CNoteDisplay part from CPostitNote and introducing a zoom manager component into CDrawing. Graphically, this would look as shown in figure 6 (changes highlighted).

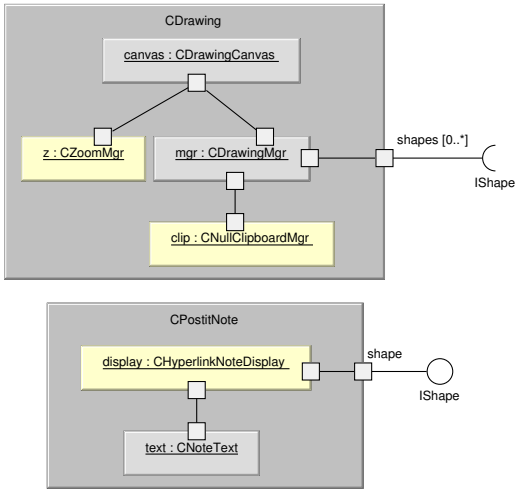


Figure 6: The architecture with Y's changes

As a consequence of the analysis, Y decides to make the changes directly to the components themselves. However, a further obstacle is that X has only released their components in a binary form in order to protect their intellectual property. This leads to the next requirement:

NoSource The reuse approach should work even if the full source code of the implementation is not available.

3.3 Evolution Scenario

Suppose that Y is somehow able to reuse the components for its product, incorporating the changes as described in figure 6. X then issues a new release, upgrading the CDrawing component to use the new CFastDrawingCanvas component, providing improved performance. Clearly, Y wishes to incorporate this improvement into its reuse of CDrawing, leading to a requirement that any reuse solution should not cut off a component from its natural upgrade path from the provider. This effectively rules out copy and paste as a reuse mechanism.

Upgrade It should be possible for a reuser to accept an upgrade to a component, even if that component has been altered for reuse.

4 ADL Constructs for Component Reuse and Evolution

From the analysis of the requirements, constructs have been developed and integrated into the Backbone ADL. The constructs are *resemblance*, *redefinition* and *stratum*.

4.1 Analysing the Requirements

Requirements *Alter* and *NoImpact* appear to be in direct conflict. The provider and other reusers do not have to accept or even know about changes to the component, and yet alterations must still be allowed in order to facilitate reuse.

This situation can be resolved by holding any alterations to a component separately from its original definition. By keeping these alterations with the system that is reusing the component, no-one else will be impacted by, or even aware of, the changes. *Upgrade* further suggests that changes should be held in such a way as to allow them to be analysed and combined with future upgrades of the component. This suggests keeping the alterations explicitly as differences or deltas, rather than storing the entire altered component.

Alter indicates that we need the ability to modify any aspect of a component to facilitate reuse, including interface definitions. This blurs the line between modification for reuse, and the evolution of a component. Such a facility will allow upgrades to also be delivered as a set of differences, distilling the *Upgrade* requirement into the ability to merge two different sets of alterations.

Finally, the requirements imply that we need a way to group related definitions together to differentiate between an existing system and a new system.

The resemblance, redefinition and stratum constructs have been developed in response to the above analysis. Resemblance allows one component to be defined in terms of alterations to a base component, such that the base definition is not affected. Redefinition allows the definition of an existing component to be altered or evolved, and coupled with resemblance allows the new definition to be phrased in terms of alterations to the old definition. Stratum provides a package-like mechanism for grouping a related set of definitions.

4.2 Using Resemblance to Express Change

The resemblance construct allows one component to be defined in terms of changes to another. This is an inheritance-like construct for components, but it does not imply a subtype relationship between components in the way that inheritance usually does between classes [15], as features can be added or removed.

A component can indicate that it resembles a base component, by providing a list of changes in terms of renaming, adding, replacing or deleting elements from the base. For instance, we can form CNewDrawing in terms of CDrawing, thereby altering it for reuse:

```
component CNewDrawing resembles CDrawing {
  replace-parts:
    CNullClipboardMgr clip;
  parts:
    CZoomMgr z;
  connectors:
    zoom joins zoom@z to
      surface@canvas; }
```

This component definition does not perturb the original definition, and does not affect any existing usages of it.

4.3 Using Strata to Control Dependencies

The stratum construct exists to group definitions and control their dependencies. A stratum is a package-like concept which groups a set of related component and interface definitions. It indicates which other stratum are visible for these definitions to refer to through dependency relations. To facilitate strata reuse, circular strata dependencies are not allowed.

To simplify the tracking of dependencies and the analysis of how strata can be combined to create a system, we have restricted the concept to being non-hierarchical. In other words, a stratum cannot contain another stratum. The only valid relationship between stratum is a dependency.

A system is constructed by indicating which strata will be included and in what order. For instance, if CDrawing is in stratum Base and CNewDrawing is in stratum Extension, then a strata load list of {Extension, Base} will cause Base to be loaded into the interpreter, followed by Extension.

4.4 Using Redefinition to Evolve Components

It is not always sufficient to reuse a component by declaring a new component that resembles it. When a component is used in an existing architecture, and a wide-ranging change is required, the original component definition may need to be altered. Redefinition provides a way to alter the definition of the component, but still keep the differences in a separate stratum so that the revised definition is only visible to those systems which include the stratum.

To redefine the CDrawing component, we can use redefinition and resemblance together. The redefinition allows the replacement of an existing definition, and resemblance allows the new definition to be expressed in terms of differences to the previous definition.

```
redefine-component CDrawing
  resembles [previous]CDrawing
{
  replace-parts:
    CNullClipboardMgr clip;
  parts:
    CZoomMgr z;
  connectors:
    zoom joins zoom@z to
      surface@canvas; }
```

Redefinition can also be used without resemblance, in order to wrap and adapt a component. For instance, we can redefine CDrawing to include the old definition as a part which is then delegated to in the new definition.

```
redefine-component CDrawing
{
  ports:
    shapes[0..*] requires IShape;
  parts:
    [previous]CDrawing old;
  connectors:
    delegator joins shapes to
      shapes@old; }
```

If the redefinition is in the Extension stratum and the original definition in Base, then the load list of {Extension, Base} will include the alterations. If, however, another client does not wish to use the changes, Extension is simply omitted from the load list. Conceptually, the changes are applied at start-up time to effect the alterations.

Further, using this construct, a provider can issue updates to a component and release this as another stratum. Suppose that X releases an updated form of CDrawing in a stratum called

Update, where CFastDrawingCanvas has replaced the original CDrawingCanvas part.

```
redefine-component CDrawing
  resembles [previous]CDrawing
{
  replace-parts:
    CFastDrawingCanvas canvas; }
```

We can include both sets of alterations above by using the load list of {Update, Extension, Base}. The base definition is loaded, and then modified by the inclusion of the redefinition in the Extension stratum. Finally, the definition is again modified by the redefinition in the Update stratum.

4.5 Summary of Approach

The relationship between the constructs is shown in figure 7, where component definitions are shown as small boxes within a stratum. Stratum are loaded in the reverse order of the load list, and each successive stratum has the ability to alter any definitions in lower strata via redefinition.

Redefinition is shown as an arrow from an upper to a lower stratum, allowing alterations to be made to a definition in a lower stratum. Resemblance is shown as an arrow from a lower to an upper stratum, allow a definition in a stratum to reuse and alter a definition from a lower stratum without perturbing the original definition. Even though the system is loaded from bottom to top, the eventual view of the system is from the top down.

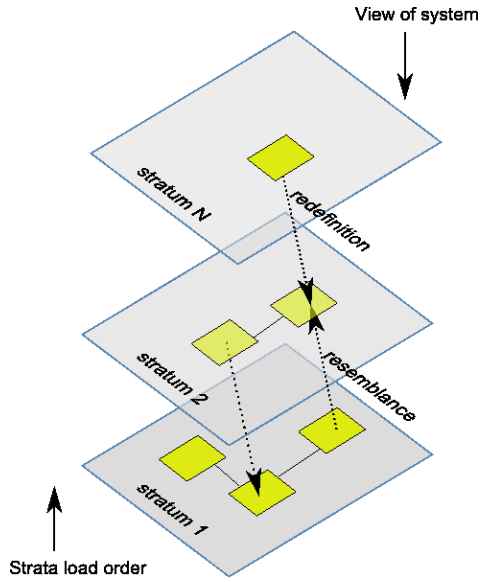


Figure 7: Conceptual view of constructs

Resemblance and redefinition support *Alter* and *NoImpact* by allowing extensive alterations to be made to a component without impacting any existing usages. As explained previously, support for *Upgrade* relies on the ability to combine multiple redefinitions of a single component. This can result in name collisions and other issues, analogous to the problems experienced by the use of multiple inheritance [15]. This situation also occurs when combining two independently developed systems that redefine the same component in a common stratum.

Currently when two redefinitions cannot be merged automatically due to overlap, manual alteration of one of the redefinitions is required. In this case, the replace, add and delete facilities seem rather uncompromising. See the section on further work, detailing possible solutions to this problem.

NoSource is partially supported, as long as the Backbone definitions and interface definitions are provided (even if the implementation code is not). The range of alterations for leaf components is then restricted to adaptation through decoration [1] or outright replacement. It is still possible to freely alter composite components, as they only have a Backbone expression.

Backbone further supports *Alter* by allowing interfaces to be redefined also, and tracking the possible leaf components which also need to be redefined to support this. At an implementation level, this relies on the Java facility where a definition in one JAR file can supersede or hide the definition in another.

The approach integrates well with existing CM systems. Backbone programs are textual and can be controlled like any set of source files. The stratum and resemblance mechanisms address the concerns about either holding the entire architectural configuration of a system in a single file, or having to scatter the configuration across many files [12]. The definitions within a stratum are held in files, and each file can hold a number of Backbone definitions and redefinitions. This allows related alterations to be grouped and controlled in a simple and straight forward manner.

5 Related Work

A number of approaches have been previously proposed that deal with many of the requirements presented. Amongst other mechanisms, parameterisation is used in Koala to capture options supported by a component [18]. This approach only supports planned variation which conflicts with the *Alter* requirement. This can also result in a combinatorial explosion of options if the parameters of the constituent parts of a composite are also exposed.

Koala and other approaches allow for variation in an architecture [19, 17] to be expressed through variation points. These capture possible component variants at predefined points in an architecture. This is referred to *variation over space*. The points must be planned in advance and designed into a system, which mitigates against this technique for the reuse of existing components which must remain unchanged.

In current product family approaches, if deep modifications or new variation points are required for an existing component these must be introduced by forming a new revision of the component. This is known as *variation over time*, and any unplanned changes require perturbing the original definition violating many of the requirements. Further, repeated introduction of variation points can quickly create complex and generic architectures which are difficult to reuse and reason about.

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 [17, 12]. 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 reusers, eventually leading to a complex, very generic architecture which deeply violates the *NoImpact* requirement.

ROOM includes a notion of inheritance which allows for additive and subtractive changes to be specified against actors [13].

A ROOM actor is analogous to a (composite) component with its own thread of control. No formal model of this language has been constructed, and the inheritance facility is not suitable for redefinition, evolution or arbitrary change.

Architectural reconfigurations have previously been used to alter the architecture of a running system, using the property of quiescence to discern when a component can be upgraded [7]. In contrast, the approach presented here provides an intuitive modelling construct for these types of changes, and applies the concepts to the specification and reuse of components. In theory, it is possible to utilise the work on quiescence to effect architectural changes at runtime also.

C2SADEL [10] is a variant of the C2 ADL [16], supporting component specifications through the explicit declaration of state along with pre and post-conditions that indicate changes to that state. This system addresses evolution using a type-based taxonomy of components and connectors and supports configuration evolution, but does not feature composite components. The approach is supported by a modelling environment called DRADEL.

6 Current Status

The interpreter, jUMbLe modelling tool and Alloy model for Backbone are available at the following location: http://www.doc.ic.ac.uk/~amcveigh/backbone.html

6.1 The Backbone Interpreter

An interpreter for Backbone has been developed in order to experiment with the language. This fully supports the resemblance, redefinition and stratum concepts. Note however that in the current interpreter, redefinition automatically presumes resemblance from the base component, as opposed to the examples presented earlier in 4.4.

The interpreter is written in Java, and uses reflection to instantiate and connect components at startup time. A strata load list is supported.

In recent use, it became apparent that names of elements in Backbone programs are being used for two purposes: human understandability and logical identity. E.g. a component specifies that it resembles another component by using its name. Unfortunately, support for renaming interferes with the concept of identity. As a result, it has been decided to explicitly separate the two concepts. The identity will be assigned as a globally unique identifier.

For instance, when defining a component, both the identity and name will be used (identity/name). However, when referring to an element, only the identity is required. This ensures that the identity remains the same, even if the element's name changes. The following definition shows how the code listing in 4.4 might look under this scheme.

```
redefine-component C0012/CDrawing
  resembles [previous]C0012
{
  replace-parts:
    CNullClipboardMgr P009/clip;
  parts:
    CZoomMgr P023/z;
  connectors:
    zoom joins PT001@P023 to
      PT002@P010; }
```

Clearly, assigning and working with identifiers places a large burden on a designer. However, this is not an issue with a graphical approaches to modelling, which explicitly separate the two concepts. For instance, a dependency relation between component A and B is not linked via the name of the components, but by their logical identities. Changing the names will not affect the relation.

6.2 Graphical Modelling with Backbone

In order to support modelling with Backbone, we have developed a prototype UML2 modelling tool called jUMbLe. A key focus of the approach is to completely hide the textual language (including logical identities), and allow designers to work directly with UML2 composite structure diagrams. The tool allows the creation of composite structure diagrams and package (stratum) diagrams.

The next step is to support the resemblance construct in the modeller. The aim is to allow the designer to alter a component by deleting and adding parts, and have the tool record the changes explicitly.

6.3 Formal Model of Backbone

A formal model of Backbone has been created in Alloy [6]. Alloy is a formal language based on a combination of predicate logic and relational algebra. Specifications can be model checked for counter-examples within a finite state space.

The current Alloy model does not support resemblance, although this is currently being added. The aim is to show that two redefinitions of the same component can lead to potential conflict. This model will further be used to verify that any solution to this conflict ameliorates the problem.

7 Conclusions and Further Work

From one perspective, resemblance provides a compelling modelling construct which allows an inheritance-like concept to be applied to components at all levels, including the architectural level. It makes it possible to derive other components from a base component, with changes to reflect new requirements, supporting a more incremental approach to system construction. This partially addresses the abstraction problem, as highly specific components can be altered to be reused in a new context. This is useful for internal reuse within a system, as well as for reusing existing components from providers.

By providing uniform reuse and evolution support, the constructs prevent the need to compulsively genericise and parameterise components that are intended for reuse. Unplanned changes can be catered for at the time when the change is required, rather than requiring a costly and sometimes unused upfront investment.

From another perspective, resemblance and the supporting constructs provide a decentralised form of version control, which integrates well with existing CM systems. This offers a multi-authority approach to change control, and allows the changes to be held where the component is reused, rather than were the component is initially defined. Either CM revisions or redefinition can be used for modelling variation over time, and resemblance combined with redefinition can be used for modelling

variation over space. Alterations are managed by the team that desires the changes rather than the provider of the component, allowing the original component to retain a coherent architectural vision.

There is a potential conflict between Backbone and a CM system when dealing with variation over time. Ideally, alterations will be specified using redefinitions, even for provider-supplied component upgrades, as this allows better reasoning about the combination of changes. However, it is not possible to keep specifying deltas indefinitely in this way, so a utility is provided which can compress multiple redefinitions into one new definition. We call this process *baselining* in keeping with the terminology of CM system. We are also investigating the possibility of constructing reverse redefinitions from a baseline, which preserve the characteristics of previous definitions.

As explained in 4.5, multiple redefinitions of a single component present a problem when incompatible or overlapping alterations are specified in two independent strata. We are currently pursuing two approaches to resolve this situation. The first approach is based around graph transformations. This involves expressing alterations using an extensible set of transformation patterns. We aim to construct the patterns to limit or resolve any interference between redefinitions although we anticipate the need to for human guidance in some cases.

The second approach is more declarative, where we allow behavioural specifications to be registered with each component. These specifications describe the effect that the component is designed to achieve, in terms of the message protocols of the constituent components. An existing architecture can then be analysed in conjunction with a behavioural specification for a new architecture, with the aim of automatically determining the alterations required to effect the new specification.

References

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

[2] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] J. Kramer, J. Magee, and M. Sloman. Configuration support for system description, construction and evolution. In *Proceedings of the 5th international workshop on Software specification and design*, pages 28–33, Pittsburgh, Pennsylvania, United States, 1989. ACM Press.

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

[16] R.N. Taylor, N. Medvidovic, M. Anderson, E.J. Whitehead 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.

[17] 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.

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

[19] 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.