Decentralized System Evolution: An Architectural Approach

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. A compositional approach to software construction allows composite components to be modeled as instances of other components. This is a scalable approach for creating systems with complex architectures.
Systems do not remain fixed, however, and must evolve when requirements change. To handle new versions, an entire architecture is typically placed under centralized configuration management.
Unfortunately, this centralization may not reflect the relationships between the parties making changes to a system. For instance, it is a common requirement that one party construct a system and release updated versions, and others evolve and customize it. Another party may then combine these multiple, parallel evolutions into a consistent system.
To address this problem, we add a small number of constructs to a conventional ADL. We show these provide a rigorous, decentralized approach to both architecture and implementation evolution, permitting any party to evolve a system even if notionally they do not control it.

1 Introduction

Architecture description languages (ADLs) allow a compositional approach to system construction [1]. A composite component is structured as a collection of instances of other components. This powerful and hierarchical method scales up to the architectural level, allowing an entire system to be created and described as a single component [2]. Looked at from another perspective, a component representing a system may be hierarchically decomposed until we get to leaf components (figure 1).

Initial creation is just part of the system development lifecycle, however. Subsequent evolution and associated maintenance is estimated to account for more than half of the development effort over a system's lifetime [3]. Current ADLs provide only limited support for evolution, usually based around the notion of subtyping and contracts [4,1]. This implies that ADLs are not properly addressing a sizable proportion of the development lifecycle.

To address system evolution in this context, we augment a conventional ADL with three additional constructs derived from observations about how components are reused and evolved. The result is the Backbone ADL. The constructs are *resemblance*, *replacement* and *stratum*.

We demonstrate that these constructs provide full support for system evolution from an architectural perspective. Further, they provide a robust, decentralized

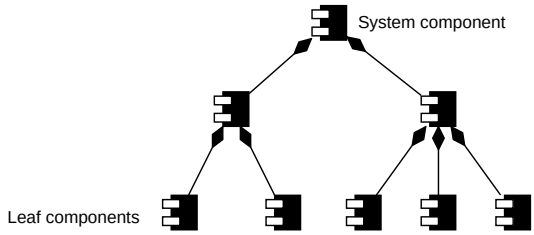

Fig. 1. An example of a system composition hierarchy

approach, where multiple parties can overlap in their control and authority over the architecture. The same constructs can also be used to rectify any conflicts when independently developed evolutions are combined into a single model.

Existing distributed configuration management (CM) systems are able to model decentralized evolution at the level of source files and textual merges [5,6]. Architectural CM systems, such as MAE [7], are able to model evolution of an architectural description in a way that allows distribution of changes to be handled. This distribution is done, however, in a relatively cumbersome way using inter-file branching requiring a map between original and derived artifacts [8].

We distinguish our work from these offerings by adding evolution constructs directly to an ADL, as opposed to adding architectural understanding into a CM system. By increasing the modeling power of the ADL to cover the full software lifecycle, we provide a consistent and rigorous approach to the decentralized evolution of both architecture and implementation. The ownership of parts of the system can be modeled using ADL constructs, providing a clear guide to which party is in authority over various parts of the decentralized system.

Our approach provides additional benefits: component reuse is enhanced, as the added constructs can also be used to adapt (or evolve) components for reuse in different contexts. In this sense, we provide a unified view of both component evolution and reuse.

The rest of this paper is structured as follows: section 2 examines system evolution in an architectural setting in more depth, presenting the Backbone ADL by way of example and showing how the constructs address the issues. Backbone is mapped onto the UML2 in section 3, leading to a graphical, tool-based approach. Related work is reviewed in section 4 and section 5 presents conclusions and discusses further work.

2 Decentralized System Evolution in Backbone

The Backbone ADL was created as part of this work, in order to better understand architectural reuse and evolution. At its core, Backbone is largely similar to Darwin [9], although it currently deals with local (i.e. in the same process) rather than distributed components.

Backbone contains three *evolution constructs*, which enable it to model evolution and reuse for both components and interfaces [10]. The constructs are *resemblance,*

replacement[1] *and stratum.* Although the constructs are equally applicable to interfaces, we will focus primarily on components in our discussion.

Resemblance allows a new component to be defined in terms of its similarity to other components. It operates at a structural level, allowing the constituents of the other components to be inherited and then added to, deleted or replaced using delta changes. It can be seen as a more flexible and structural form of inheritance, where any type of change can be made including deletion. As a result, unlike inheritance, resemblance carries no implication of substitutability.

Replacement allows one component to replace another. The former component assumes the latter's identity, and any other component that referenced the latter will now reference the former instead. Combined replacement and resemblance allows a replacing component to be defined in terms of delta changes to the replaced component. We term the this combined usage *incremental evolution* reflecting the effect of using the constructs together.

The third construct, *stratum,* provides a module construct for Backbone. A stratum groups definitions, and must be explicit about its dependence on other strata. Strata are used to structure the architecture of a Backbone program and indicate the high-level dependency structure. Strata each have a single owner in a decentralized setting, and this ownership defines the authority of parties over the architecture.

Below, we present an example requiring decentralized development amongst multiple companies. We then show how the Backbone ADL and the evolution constructs address the issues found at both the architectural and implementation levels.

For the graphical treatment, UML2 composite structure diagrams are used. For a mapping from Backbone to UML2, see section 3.

2.1 An Example of Decentralized Development

In order to examine the challenges of evolution in a decentralized, multi-party setting, we present a simple scenario as shown in figure 2.

Consider that company D creates an audio desk application, whose function is to control audio devices and combine their output using a mixer. This application is sold to both audio product manufacturers and to radio studios. By default, the desk application controls only a CD player.

Company M produces microphones. To integrate this into the desk application, they create a microphone driver. Because this driver is compatible with the existing interfaces in the system, integration is straight forward.

Company E produces an enhanced, digital mixer component which replaces the standard mixer. Unfortunately, the interfaces and facilities provided by the desk application are not sufficient, and the new mixer cannot be used without some changes to the application. As such, E evolves the application to integrate its driver into the system.

Radio station R uses the desk application to control their on-air shows, and wishes to use the desk application with both the microphone and enhanced mixer.

[1] In [10], we refer to the replacement construct as redefinition. We have changed the term to avoid conflict with terminology adopted by UML2.

To complicate matters, D produces software updates to its desk application on a regular basis, but does not distribute these to M or E – only to R. This reflects the contractual terms between the companies. Further, R sometimes fixes bugs onsite and sends these back to D.

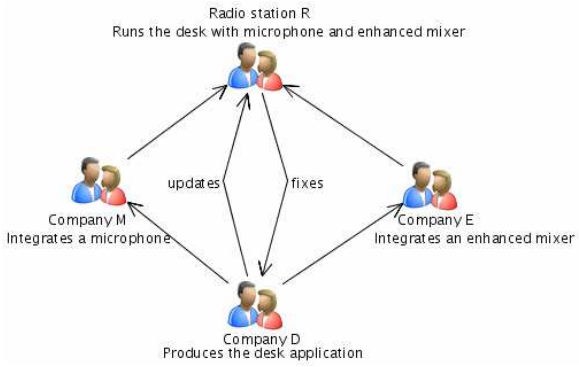

Fig. 2. The flow of software between companies

This scenario, although seemingly complex, is not particularly unusual and all evolution stems from a real need. The decentralization of the software reflects the division of responsibilities of the companies involved. Our solution recognizes that each party may need to control their own changes to the underlying architecture and implementation.

2.2 Modeling Ownership Using Strata

Strata allow Backbone definitions to be grouped. We choose to structure the system so that each company owns associated strata to contain the software they produce. The structure of the system, along with the ownership of each stratum, is shown in figure 3.

After the direction of the arrows has been reversed to indicate dependencies, the company structure of figure 2 maps neatly onto the strata diagram.

This structuring allows us to achieve the following: firstly, each company has a primary stratum which it owns, grouping the software definitions produced by that company. The dependency structure reflects the fact that M and E build on the components provided by D, and that R builds on the components provided by M, E and D[2]. Further, the Dv1.1 stratum allows D to package upgrades to the application that are only visible to R, as per the stated requirements in section 2.1. Similarly, R places any fixes to D's application in RFixes which builds on the definitions contained in Dv1.1. The dependency structure of RFixes means that any changes contained here cannot reference any components from M or E (or even R) as they are not visible to it, as required for the organization of figure 2.

[2] Strata dependency relationships, unless explicitly indicated otherwise, are transitive. This permits R to also see D.

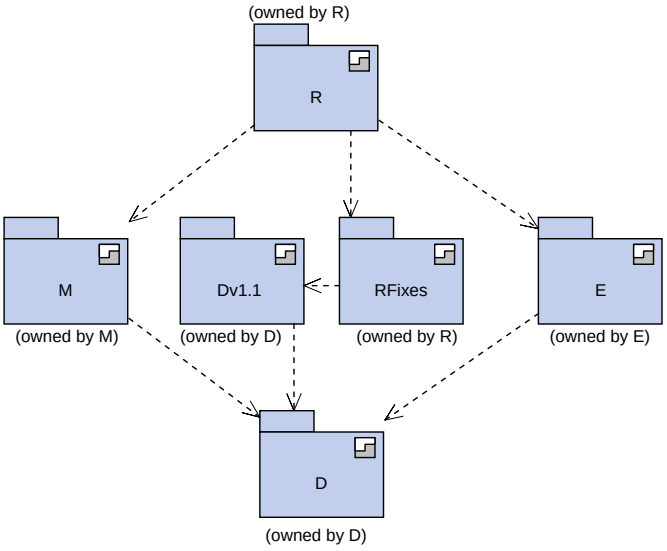

Fig. 3. The strata for the decentralized architecture

The key is that each stratum has a single owner. The owner is the only one permitted to do any direct modification of the stratum, definitions and associated implementation code. The stratum and associated implementations can be distributed to non-owning parties who must treat them as read-only. If a previously distributed stratum is modified by its owner, then it must be re-distributed. Companies can deliver copies of their strata to other parties, secure in the knowledge that these will not be modified except by themselves.

Of course, Dv1.1 shows that even an owner can choose to structure an upgrade as a separate stratum, rather than directly editing an existing stratum. There are costs and benefits to each approach – modifying an existing stratum is simpler, but may cause problems for other parties already using it.

Parties are permitted to make changes to the architecture but only via adding or editing definitions in strata that they own. However, using the Backbone evolution constructs, any party can make a change to any component that is visible to it. For instance, although M cannot directly change the definitions in D via editing, M can contain an incremental evolution (replacement and resemblance together) of the desk application component, allowing M to evolve the components in D.

Parties do not need to see strata upwards in the dependency graph, and do not have to concern themselves with the maintenance of these. For instance, company D would only have the D, Dv1.1 and RFixes strata available to it. As such, none of the evolutions in E, M or R would be applied to its view of the system.

2.3 Stratum D: Modeling the Desk Using Backbone

In this section, we discuss how company D would use Backbone to define the audio desk application. All of these definitions would be placed in stratum D.

The CD driver is modeled as a leaf component, as shown in figure 4. Leaves are not further decomposable, and are associated directly with a Java class that provides the behavior[3]. The driver provides IAudioControl (a full circle denotes a provided interface) and requires IAudio (the semi-circle denotes a required interface). Intuitively, the driver can be controlled (IAudioControl) and it sends output via IAudio. The small boxes on the edges of components are ports which encapsulate component interaction with the environment.

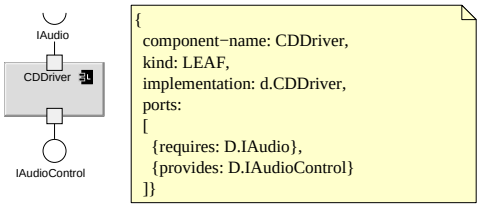

Fig. 4. The CD driver leaf component

Company D models the desk application as the Desk composite component. Figure 5 shows that this is made up of a Mixer component instance and a CDDriver component instance. These instances, in UML2 terminology, are known as parts. The textual view omits the connectors, for reasons of space.

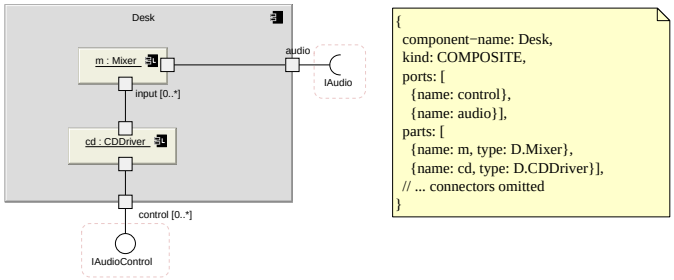

Fig. 5. The desk composite component

The provided and required interfaces of the ports need not be specified for composites – they are inferred from the connections to internal parts. They can, however, be explicitly stated if needed to allow for top-down development. Note that ports may have a multiplicity: e.g. "control [0..*]" indicates that the control port can have any number of connections.

As in Darwin (and unlike standard composite structures in UML2 [11]), Backbone composites have no explicit behavior of their own. All the behavior comes

[3] The implementation language is currently Java, although other languages could be substituted.

from parts, and only leaf components directly have implementations and therefore behavior. Leaf components are associated directly with an implementation class, whereas composite components are Backbone-only artifacts. In a sense, composites are reusable instructions for wiring up instances of other components.

Stratum M: Adding the Microphone Company M wishes to add in their microphone driver. Luckily, this provides and requires the same interfaces as CDDriver, so the new part can be wired directly into the Desk component. To define this, we make use of resemblance and replacement (a dual headed arrow) together as shown in figure 6. This allows us to define a new version of Desk in terms of changes to the old definition. In this case, we add a part and two connectors.

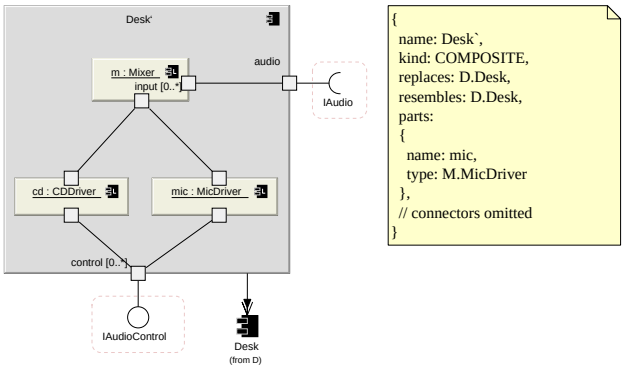

Fig. 6. Evolution of Desk to add a microphone

This is addition only – nothing is deleted or replaced. This type of change is usually handled using a plug-in framework and registry approach such as that employed by Eclipse [12]. Plug-in approaches constrain extensions to be only able to add to the system, in a way that is compatible with existing interfaces, and in a way that hooks into existing artifacts. In this case we have added an extra part and two extra connectors between existing ports.

A key point is that the new definition only contains the deltas (the added part and connectors). If the original D.Desk component changes then Backbone will re-apply the changes in M to the definition in D.

Only parties that have stratum M loaded into their environment would see the changes from this evolution. Companies D and E do not have this stratum, and therefore see the original Desk component.

Stratum E: Adding an Enhanced Mixer Company E wishes to replace the mixer with a enhanced, digital mixer. However, the new mixer is not compatible with the old IAudio interface, and instead uses the unrelated IDigitalAudio interface. To avoid breaking existing clients of Desk, company E creates a new component, DigitalDesk using resemblance.

Figure 7 shows that the use of the enhanced mixer requires a DigitalConverter instance to be placed between the new mixer and existing audio devices. The single bold arrow represents resemblance.

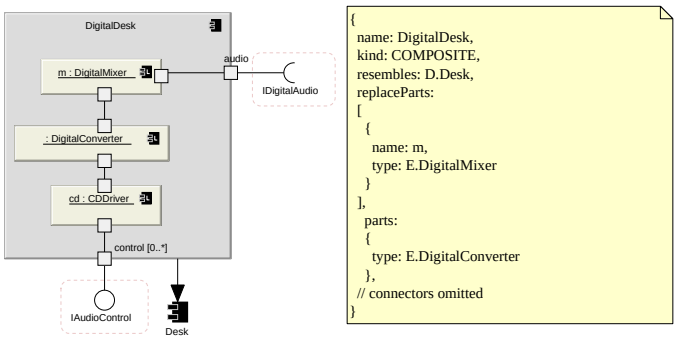

Fig. 7. Replacing the mixer

These definitions (and associated leaf implementations) are packaged into stratum E. This stratum is only available to companies E and R. D and M will not see any changes due to definitions in stratum E.

Stratum R: Combining Definitions at the Radio Studio Company R now wishes to combine the software from all other companies. As stratum R depends on M and E (and D via transitivity), it sees all the definitions.

By default Backbone will combine the different components into a unified resemblance hierarchy reflecting the replacements: DigitalDesk resembles M.Desk' resembles D.Desk. However, this will produce the DigitalDesk component as shown in figure 8. There is an error in this model because there is no DigitalConverter part between the microphone and the enhanced mixer.

The error is not particularly surprising. When E introduced the new mixer, it adjusted the architecture to fix up CDDriver's connection to the mixer. E could not fix up the microphone driver, because it literally cannot see the software in stratum M due to the dependency visibility rules. This reflects the reality of the situation – in the organizational hierarchy of figure 2 company E has no knowledge of company M.

Errors in the model are picked up by the Backbone well-formedness rules [13]. Because we are working at an architectural level, in terms of components, connectors and parts, the model is also checked at this level. In this case, the MicDriver output does not match the DigitalMixer input, and we need to place in another DigitalConverter part.

To insert this part, company R creates an incremental evolution of DigitalDesk, as shown in figure 9. This simply adds in the new part, and a new connector, and rewires (replaces) the old connector to the mixer.

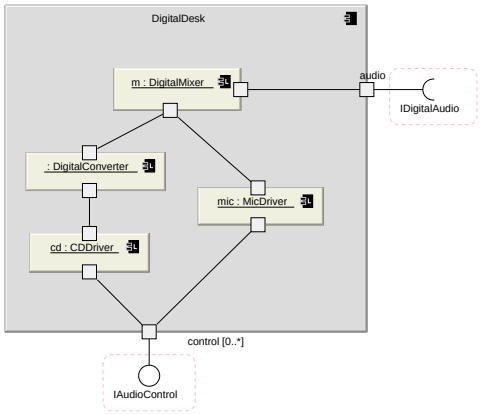

Fig. 8. A naive merge results in errors

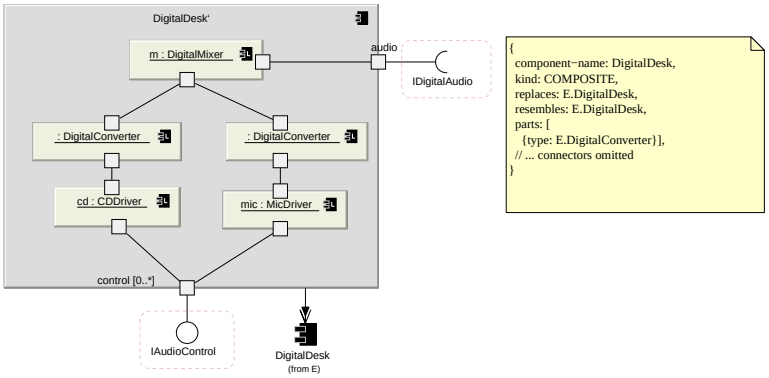

Fig. 9. Errors are corrected by a further incremental evolution

2.4 Strata Dv1.1 and RFixes: Upgrading

Company D can use the evolution constructs to evolve its own desk application in stratum Dv1.1. Although not standard Backbone practice, the presence of the version number in the stratum name clearly communicates the intent. M and E see only the older version of the application, whereas R sees the newer version, combined with both M and E's changes.

It is clearly possible for the changes from v1.1 of the application to be incompatible with the changes from M and E, as they are created independently. If this were the case, the errors would be flagged by the well-formedness rules and R is then able to use the evolution constructs to make any required corrections.

If Dv1.1 contains an incremental evolution of Desk (Desk') then we effectively get into a situation where parallel evolutions (Dv1.1::Desk' and M::Desk') must be merged, from R's perspective. In this case, Backbone rewrites DigitalDesk's resemblance graph so that it multiply resembles both Dv1.1Desk' and M::Desk'. Again, R is able to use the same evolution constructs to correct any errors.

RFixes can be used by R to package any fixes to the D or Dv1.1 software. In this case, company R is the owner of the stratum and distributes it as read-only downstream to D. Because RFixes can only see D and Dv1.1 components, it will not reference the components from M, E or even R, fulfilling the requirements that the fixes be distributable to D.

2.5 Interoperability with Conventional CM Systems

Each party can check their owned strata (both Backbone and implementation code) into a conventional CM system if desired. If changes are made, they can be tracked using standard approaches. Of course, a party is also free to consider even its owned strata as read-only, reflecting a fixed version. This party can then phrase subsequent versions in a new stratum (e.g. Dv1.1), that incrementally evolves the definitions of the fixed stratum. The choice as to when to regard an owned stratum as read-only depends on how sensitive upstream parties are considered to be to changes and how comfortable they are with rectifying errors that may be introduced.

We have found that name changes of components and interfaces are a very frequent occurrence. UUIDs (universally unique identifiers) are used in practice to identify components. This approach has shown its value in keeping the system robust, preventing clients from being affected by cosmetic changes which occur commonly in practice.

Some types of architectural changes do not affect clients. For instance, if a previous leaf component is turned into a composite and then decomposed into a hierarchy of other leaves, clients will not notice any effect on picking up a new read-only version. As long as the "shape" of the component has not changed, nor the identities of previously visible constituents, clients are not affected.

2.6 Relationship to Implementation

Each leaf component must be paired up with a Java implementation class. The idea is that the owner of the stratum also owns and manages the source code for the leaves of the stratum. The code is checked into a conventional CM system of the stratum owner, as described above.

Clients of a stratum reference its associated JAR[4] file for the implementation. This file contains the implementation for the leaves and is distributed along with the stratum. If any changes to the implementation is required, these are performed via incremental evolutions (resemblance and replacement) in an owned stratum. This allows parties upstream from the stratum to make any required changes to the implementation, proceeding from an architectural perspective.

This does mean that some evolutions require extra effort, depending on the size of the element being evolved. For instance, if we are incrementally evolving a very large leaf component that we do not own, we have no choice at the implementation level except to replace it with another class. In this sense, the compositional component model and the evolution constructs work well together. The component model allows fine-grained component decomposition at different levels of abstraction, implying that the replacement can be targeted precisely as a small modification.

[4] Java Archive (JAR) files are the standard way of distributing Java libraries.

3 Mapping Backbone onto UML2

To allow a graphical approach to Backbone modeling, we have mapped Backbone onto the UML2 composite structure diagram and associated concepts [14]. In this section we explain the correspondence between the two languages.

UML2 is a flexible graphical language with an informal description of the underlying software-related constructs. It includes an extensibility mechanism known as stereotypes which allow new rules to be defined for existing constructs in the language. UML2 has previously been found to be a suitable ADL with the caveat that support for connectors is somewhat lacking [15]. In our case this is not an issue, as our connectors do not have complex semantics.

3.1 Component as Structured Classifier

We model components in Backbone as UML2 structured classifiers. A structured classifier is a class which can contain a configuration of instances (parts), effectively forming a configuration.

A clear difference between Backbone and UML2 in this regard is that Backbone composite components do not have any behavior of their own. This is not the case in UML2 where structured classifiers may have associated implementations [11]. In Backbone, only leaves have implementations and therefore behavior. As such, each Backbone composite must contain ports and connectors from the composite to the parts. Composites are really just a form of shorthand for how to wire up a set of leaf component instances.

3.2 Stratum as Package

UML2 uses packages as a module construct. Packages control the visibility and export of their contained elements, and also constrain what those elements are allowed to see from other packages via dependency relationships.

One undesirable aspect of packages, from the perspective of modularity, is that an inner package can see any definitions in the direct scope of its parent (figure 10). This is designed to mimic the scoping rules of programming languages [16]. This is unhelpful for Backbone, as it implies that a child package may be bound to the definitions in the parent package, thereby preventing its isolated export from the system for distribution to other parties.

As such, the stratum concept is modeled as a stereotyped package accompanied by different visibility and scoping rules. Elements contained within a stratum are always public, although these are not visible to another stratum unless it (possibly transitively) depends on the first stratum. Further, child strata cannot reference the definitions of their parent stratum, as was also the case in older versions of UML [16]. In this way, we can always export a contained stratum and decouple it from its parent.

3.3 Resemblance and Replacement as Dependency

Resemblance differs from inheritance, allowing deletion and potentially incompatible replacement. UML2 does include a notion of redefinition, which allows the

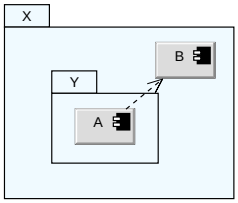

Fig. 10. Children can reference parent definitions in UML2, but not in Backbone

inheriting classifier to replace any element, giving inheritance some of the power of resemblance. However, inheritance also carries the implication of substitutability [17], which resemblance does not. Because the semantics differ from inheritance, we model resemblance instead using a stereotyped dependency with a specific graphical arrow (c.f. figure 7).

There is no direct analog for replacement in UML2. We therefore also model this using a stereotyped dependency.

UML2 includes the notion of package merge, which allows a number of packages to be combined, but this is a form of union with a complex and troublesome specification [18]. As such, we have avoided this construct, which is more appropriate to describing the relationship between different conformance layers of the UML.

3.4 Composite Structure Diagrams

Composite structure diagrams are used to display the Backbone structural form. They are effectively a combined form of class and object diagram, showing the internal structure of composites. Although UML2 component diagrams could also have been used, we felt that the simpler visual appearance of the composite structure diagrams was compelling. The use of these diagrams for complex architectures is shown in [19]. The similarity between this approach and ROOM is clear [20].

We provide a CASE tool for modeling in Backbone, as shown in figure 11. This allows strata to be imported in and exported out of the immediate environment. The full expanded form of any component is always displayed, in a similar way to how we have expanded the resemblance relationship in figure 9. The aim is to make component evolution and reuse as natural and manageable a process as initial creation. The well-formedness rules are checked incrementally as an architecture is elaborated, giving quick feedback on the structural correctness of the design.

4 Related Work

MAE is a CM system which has been fused with architectural concepts. As such, it is able to understand architectural changes at the level of the artifacts themselves, construct difference patches, and reason about merges. This system shows clearly the value of bringing CM and architectural concepts closer together. When combined with a physical method which can be applied to existing CM approaches called inter-file branching [8], MAE can be used to evolve and track a decentralized architectural description. This requires some form of mapping between files

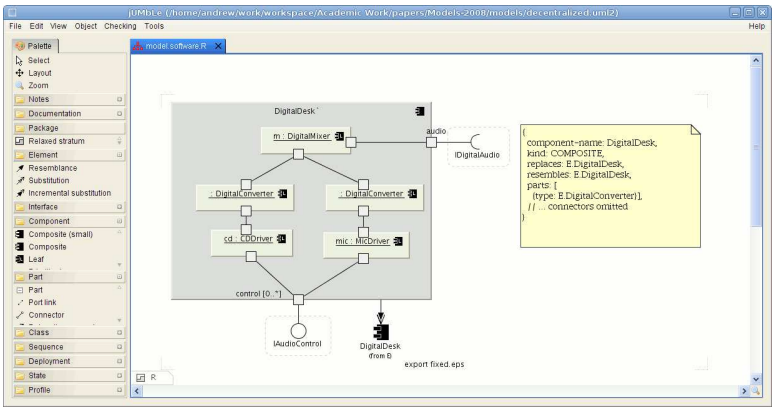

Fig. 11. A screenshot of the Backbone CASE tool

representing the variants. By contrast, our work pushes CM and evolution concepts directly into the ADL, allowing it to function as a decentralized way to track and manage both architectural and implementation evolution. The stratum construct of Backbone also provides a way to naturally model the organization and authority of participants in a decentralized software development process.

Distributed CM systems generally work through allowing multiple copies of a repository to be made, and providing support for bringing these spaces back into centralized consistency via a merge operation [5,6]. A key point is to allow multiple repositories to be merged in any order. For instance, if B and C are copied from A, B and C can merged separately and then merged with A. Alternatively B could be merged with A, and then C with A.

Plug-in frameworks provide a way to express non-destructive evolution. The Eclipse plug-in framework combines an additive approach (by adding extra plug-ins to a configuration) with a versioning strategy that permits replacement. No deletion facilities are provided apart from physically removing a plug-in from a configuration. As we point out in [21], the combination of the versioning approach with the lack of a hierarchical component model produces undesirable effects: the cost of change is often severely out of proportion with the actual change required. In other words, even for a small destructive change it is often necessary to fork an entire plug-in which implies access to the source code. Depending on whether it is a breaking change or not, this can have a cascade effect requiring the update of many plug-ins.

Backbone, although initially developed independently, is very similar to Darwin [9]. Koala, which is derived from Darwin, offers support for building product families through parametrization and variant management [22]. C2SADL is an ADL which has been enhanced with subtyping constructs designed to allow it to express a form of evolution and reason about contractual violations in an evolved system [4]. ROOM includes a notion of inheritance closely related to resemblance, allowing destructive changes to occur [20].

5 Conclusions and Further Work

The Backbone ADL contains three evolution constructs: resemblance, replacement and stratum. Resemblance and replacement allow the evolution of components and interfaces to be expressed.

The use of strata allows us to group definitions (including evolutions) into a structure that maps cleanly onto the organization of participants in decentralized software development scenarios. Strata owners are the only parties permitted to modify their strata – others receive read-only copies. Any party can change the architecture of areas defined by strata they do not own, but must do so by defining evolutions within strata they do own. This allows a multi-authority approach to system evolution, where parties can overlap in their control of the architecture, albeit in a principled way.

The approach taken by Backbone is relatively lightweight, and can be used to replace plug-in approaches, and other less expressive extensibility and evolution mechanisms. Unlike plug-in architectures, Backbone allows destructive changes to be modeled, and the constructs can further be used to rectify any conflicts when strata dependencies cause a merge of parallel evolutions. This power allows both system updates and fixes to be structured as further strata.

We have constructed a formal specification of Backbone [23]. This is expressed using the Alloy logic language, which is coupled with a model finder. The specification details the rules which are used to check that a Backbone model is well-formed.

We also provide a CASE tool for graphical modeling using the approach. For this, we map the Backbone ADL onto UML2 composite structures and diagrams. The tool shows fully expanded components rather than just deltas, with the aim being to make component evolution as natural as initial component creation. Although components are identified by human-readable names on the screen, UUIDs are used behind the scenes for component references, allowing names to be changed without affecting existing clients.

We have recognized that designing with evolution constructs can create a system where ongoing updates are specified using an ever increasing number of strata. Future work is focused on creating a *baselining* construct which will compress these multiple strata into a single one. Other work includes adding protocol modeling [24] and behavioral consistency checks to provide further guarantees when merging.

References

1. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. Ieee Transactions on Software Engineering **26**(1) (January 2000) 70–93
2. Kramer, J., Magee, J., Sloman, M.: Configuration support for system description, construction and evolution. In: Proceedings of the 5th international workshop on Software specification and design, Pittsburgh, Pennsylvania, United States, ACM Press (1989) 28–33
3. Kemerer, C.F., Slaughter, S.: An empirical approach to studying software evolution. IEEE Trans. Softw. Eng. **25**(4) (1999) 493–509
4. Medvidovic, N., Rosenblum, D., Taylor, R.: A language and environment for architecture-based software development and evolution. In: ICSE '99: Proceedings

of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 44–53
5. van der Hoek, A., Heimbigner, D., Wolf, A.L.: A generic, peer-to-peer repository for distributed configuration management. In: ICSE '96: Proceedings of the 18th international conference on Software engineering, Washington, DC, USA, IEEE Computer Society (1996) 308–317
6. Milewski, B.: Distributed source control system. In: ICSE '97: Proceedings of the SCM-7 Workshop on System Configuration Management, London, UK, Springer-Verlag (1997) 98–107
7. Roshandel, R., Van Der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae—a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. **13**(2) (2004) 240–276
8. Seiwald, C.: Inter-file branching - a practical method for representing variants. In: ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management, London, UK, Springer-Verlag (1996) 67–75
9. Kramer, K., Magee, J., N., K., Dulay, N.: 2. Software Architecture Description. In: Software Architecture for Product Families: Principles and Practice. Addison Wesley (2000) 31–65
10. McVeigh, A., Magee, J., Kramer, J.: Using resemblance to support component reuse and evolution. In: Specification and Verification of Component Based Systems Workshop (to be published). (2006)
11. Oliver, I., Luukala, V.: On uml's composite structure diagram. In: Fifth Workshop on System Analysis and Modelling, Kaiserslautern, Germany (2006)
12. Inc., O.T.I.: Eclipse platform technical overview. Technical Report **http://www.eclipse.org/whitepapers/eclipse-overview.pdf** (July 2001)
13. McVeigh, A.: Backbone: An architectural approach for extensible applications. Technical report, Imperial College (2007)
14. OMG: Uml 2.0 specification. Website **http://www.omg.org/technology/documents/formal/uml.htm** (2005)
15. Goulo, M., Abreu, F.: Bridging the gap between acme and uml 2.0 for cbd. In: Specification and Verification of Component-Based Systems (SAVCBS 2003). (2003) –
16. Schürr, A., Winter, A.J.: Formal definition and refinement of uml's module/package concept. In: ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology, London, UK, Springer-Verlag (1998) 211–215
17. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education (2004)
18. Zito A, Diskin Z, D.J.: Package merge in uml 2: Practice vs. theory? Model Driven Engineering Languages and Systems (2006) 185–199
19. Selic, B.: Tutorial d: An overview of uml 2.0 (2003)
20. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons (1994)
21. McVeigh, A., Magee, J., Kramer, J.: (in submission) extensible systems: Plugin versus component substitution architectures. In: 23rd IEEE/ACM International Conference on Automated Software Engineering. (2008)
22. van Ommering, R.: Building product populations with software components. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, New York, NY, USA, ACM Press (2002) 255–265
23. McVeigh, A.: Alloy specification of backbone. Technical report, Imperial College (2007)
24. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans.Softw.Eng. **28**(11) (2002) 1056–1076