Intrinsic Definition in   
Software Architecture Evolution

Andrew McVeigh, Jeff Kramer, and Jeff Magee

**Abstract**— Incremental change is intrinsic to both the initial development and subsequent evolution of large complex software systems. We present an approach that captures this incremental change in the definition of software architecture. The predominate advantage in making the definition of evolution intrinsic to architecture description is in permitting a principled and manageable way of dealing with unplanned change and extension. We show how intrinsic definition also facilitates decentralized evolution in which software is extended and evolved by multiple independent developers. Further, we show how unplanned extensions can be deployed to end users with the same facility that plugins extensions are currently added to systems with planned extension points. The approach is model-driven in that architecture definition is used to directly construct both initial implementations and extensions to these implementations. We have implemented intrinsic evolution definition in Backbone - an architectural description language (ADL), which has both a textual and a UML2, based graphical representation. The paper uses Backbone to illustrate basic concepts through simple examples and reports our experience in applying it and its associated tool support to a larger example.

**Index Terms**—Software Architectures, Design Tools and Techniques

xxxx-xxxx/0x/$xx.00 © 200x IEEE

—————————— ◆ ——————————

# 1 Introduction

I

NITIALLY recognized by Boehm in his spiral model of software development [] and more recently in agile development methods [], iterative and incremental software development is fundamental to the delivery of complex software intensive systems. As early as 1976, Belady and Lehman[] recognized that to retain their usefulness, complex software systems are subject to continuous incremental change throughout their lifetime. In 1991, Lehman[] refined this observation and noted that systems are subject to incremental growth due to the need to add functionality to maintain user satisfaction. In summary, incremental change and extension can be regarded as intrinsic to both the initial development and subsequent evolution of complex software systems.

————————————————

1. A. McVeigh, J. Kramer and J. Magee are with Imperial College London.  
    E-mail: {a.mcveigh, j.kramer, j.magee} @imperial.ac.uk.

*Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.*

To deal with the increasing complexity of software systems, the Software Engineering community has focused on dealing with systems at a level of abstraction, known as software architecture []. The software architecture of a system deals with multiple views of a system including both its functional and non-functional aspects. The central view is the structural one in which a system is viewed as a set of components that interact via connectors. Control of complexity is achieved by hierarchical structure in which a component can be composed from subcomponents with the leaf components of the hierarchy representing code modules. A number of Architectural Description Languages (ADL) have been proposed by the research community [], and some of these have found their way into commercial practice []. While software architecture is seen as an appropriate level for system redesign and restructuring to permit change [], the proposed ADLs do not deal directly with evolution, considering it to be an extrinsic concern dealt with by tools and processes external to those concerned with architecture definition.

In this paper, we explore the alternative in which we regard evolution as a concern that is intrinsic to architecture definition such that the structural constructs we propose to capture change and extension can be used during both initial development and subsequent evolution. This intrinsic definition brings with it the requirement to deal with unplanned extension, for it is impossible, whichever development process is adopted, to foresee all possible future requirements for change and evolution of a system.

Dealing with unplanned extension introduces a difficult dilemma in designing constructs to support intrinsic definition. We would prefer to have constructs that constrain change in such a way that their application always results in structurally well formed and type correct systems. However, such an approach inevitably means that only a sub-class of all possible valid change to a system is permitted. This is clearly incompatible with dealing with unplanned extension since we cannot predict the change that will be necessary. As a result, we have chosen constructs that do permit changes that result in invalid systems; however, we have ensured that these constructs are used in an environment that comprehensively detects structural and type errors. In other words, we combine the freedom to perform incorrect changes with the ability to detect these errors so that we have sufficient expressiveness to deal with unplanned changes. We permit change to be destructive – deleting elements of an architecture – in addition to being constructive – adding elements to an architecture.

In exploring intrinsic definition, we have considered the requirements of distributed design and evolution in which development and extension is carried out by different organizations. In particular, we are motivated by the following situation which current approaches find problematic. A development organization produces a software framework product that is used by other organizations to build applications. In meeting their local development requirements, these organizations may need to modify and extend the framework to support their applications. The original framework will evolve over time and the organizations that use it need to apply their local changes to the framework before using the evolved framework for their applications. In addition, a third party may wish to use applications from more than one extenders of the framework and thus need to merge changes from both these organizations and the original framework provider. In the sequel, we examine the impact of intrinsic definition on distributed evolution of this form.

If software architecture description is regarded only as design documentation, a major problem arises in keeping this documentation in step with the software implementation as a system evolves. We adopt a model driven engineering approach in that architecture definition is not just a documentation artifact but is a precise model used to directly construct both initial implementations and extensions to these implementations.

In the following: Section 2 describes the key concepts for the intrinsic definition of evolution in software architecture and explains their use by mean of a simple example, Section 3 provides a more rigorous definition of the these concepts and shows how they are computed over a dependency graph of extensions, Section 4 recounts our experience in using our prototype tool Evolve applied to a large case study and evaluates the approach against the requirements for distributed evolution, Section 5 discusses related work and finally Section 6 presents some conclusions and directions for future work that build on the approach.

# 2 Defining Change

We first describe and motivate the key concepts we use to define change in software architecture. Specifically, these constructs provide the facility to remake a composition hierarchy. We demonstrate this in a simple example described using our description language Backbone in which the constructs have been implemented. It has both a graphical form based on UML2 component diagrams and a textual form. The textual form is simply a printed version of an XML representation used to store and exchange Backbone definitions.

## 2.1 Key Concepts

### Resemblance

Resemblance defines a new *component* as the difference in structure from one or more existing components. It is the delta, consisting of the set of additions, deletions and replacements of the elements of these components, that is applied to arrive at the new definition. For a component, these elements are: *parts* – instances of subcomponents, *ports* – instances of interfaces, *connectors* – bindings between ports, and *attributes* – component parameters. Resemblance may also be applied to interfaces in which case the modified elements are operations. If a resemblance delta consists only of additions then when applied to an interface, it defines a proper subtype. The type inference and checking algorithm we have implemented in our prototype tool Evolve takes full cognizance of this.

Resemblance is a many to one relation to permit the merging of multiple component definitions that may have arisen due to, for example, distributed development. It might be of concern that if a sufficiently radical delta is applied to a component then the new definition will bear little or no resemblance, in the general sense, to the component definitions from which it is derived. However, this is of more philosophical than practical import as the primary intent of resemblance to record change or evolution and we can find many examples in both engineering and nature where things evolve dramatically from their original form.

### Replacement

Replacement globally substitutes the definition of one component for another while preserving the identity of the original definition such that any use relations that a larger system has with this definition are preserved. When combined with resemblance, replacement permits the incremental evolution of a component definition without necessarily having to change the composite component definitions that use this component.

Components and interfaces in the Backbone ADL are given globally unique identifiers to permit the correct unambiguous application of replacement. Replacement is the key to managing change in composite hierarchical definitions since it permits substitution of component definitions at one level of the composition hierarchy without necessarily affecting higher layers.

### Stratum

A stratum is a package or module that holds the definitions relating to an initial software architecture expressed as a composite component or definitions relating to an evolution of that architecture. Each stratum records the strata that it depends on such that to assemble a system, the builder uses the current stratum and the transitive closure of all strata that this stratum depends on. The computation of resemblance in the context of the stratum dependency graph is addressed in the next section.

In addition to extension, the stratum is the unit of sharing and ownership. Each stratum is owned by a single party that has modification rights to it. We can therefore use strata to model the ownership structure of an architecture and map this onto a community of base and extension developers. Our development tool Evolve and the Backbone runtime environment support the import and export of strata for distributing extensions and subsets of an architecture.

A stratum is also a unit of deployment in that its compiled version together with associated component code can be sent to an end user to extend their system in a similar fashion to Eclipse Plugins.

It should be noted that although resemblance allows elements to be deleted in forming a new definition from existing defintions, this is not destructive editing in the usual sense since the existing definition is preserved and the deletion simply recorded in a delta. Our approach preserves definitions and at no point overwrites old definitions with new definitions. When replacing a definition from a base stratum with a new definition using resemblance in an extension stratum, we do not remove the base definition simply record the delta definition in the extension stratum. Indeed, if we do not want a component definition to be available for use in an extension, we simply replace it with a resemblance delta that sets its *retirement* status to true.

## 2.2 Example

To illustrate the use of these three concepts we use the Singlelane Bridge example from []. This concurrent system models the access of cars to a single lane bridge. Its component architecture can be modeled in Backbone as shown in Figure 1.



Figure 1 – SingleLaneBridge Components

The textual architecture description that corresponds to this component diagram is listed below:

**stratum** SingleLaneBridge

**depends-on** backbone {

**interface** I\_access

implementation-class bridge.I\_access {

**operations**: enter; leave; }

**interface** I\_move

**implementation-class** bridge.I\_move

{**operations**: move; }

**component** Controller

**implementation-class** bridge.Controller {

**ports**:

red **provides** I\_access;

blue **provides** I\_access;

}

**component** Car

**implementation-class** bridge.Car {

**attributes**: carNo: int = 0;

**ports**:

move requires I\_move;

bridge requires I\_access;

}

**component** Display

**implementation-class** bridge.Display {

**attributes**: maxCar: int = 1;

**ports**:

redmove **provides** I\_move;

§ bluemove **provides** I\_move;

run **provides** IRun;

}

**component** SLB {

**ports**:

run **provides** IRun;

**parts**:

b0: Car;

c: Controller;

r0: Car;

d: Display;

**connectors**:

bb **joins** blue@c **to** bridge@b0;

br **joins** bridge@r0 **to** red@c;

vr **joins** move@r0 **to** redmove@d;

vb **joins** move@b0 **to** bluemove@d;

r **delegates-from** run **to** run@d;

}

}

This architecture represents a system with one red car and one blue car, moving in opposite directions, competing for access to the bridge. Access control is implemented by the Controller component, which provides two I\_access interfaces. A Car calls enter to gain access to the bridge and calls leave on exit. Cars display their movement using the Display component. Note that all of the definitions are contained in the SingleLaneBridge stratum that appears in Figure 3. Car, Controller and Display are leaf components with an implementaion defined by Java classes. SLB is a composite component that contains parts made from these components interconnected by connectors to form the system.

### Using Resemblance

Now suppose that another developer wishes to evolve this system to accommodate multiple red and blue cars moving in opposite directions. This developer creates the stratum with the components shown in Figure 2. The developer defines three new components: CarFactory dynamically creates a Car component when it receives an invocation on its creator port, CarCreator is a leaf component that calls its create port maxCar times, and finally, the composite component MultiCar creates nCars Car components.



Figure 2 – MultiCarBridge components

The MULTSLB composite component is defined as a resemblance of the SLB component from the SingleLaneBridge stratum. Graphically, resemblance is depicted as a solid arrow pointing to an icon representing the component from which the resemblance is derived. Although graphically MULTSLB is depicted with all its component parts and connectors, in fact as shown in the Backbone listing below, it is defined by a delta that simply adds the nCar attribute and replaces the parts of type Car with parts of type MultiCar. In addition, it replaces the instance of Display with an instance of the same component type but with a different attribute value. The new definition is formed graphically by editing the previous definition that the Evolve tool displays when the resemblance relation is established.

**stratum** MultiCarBridge

**depends-on** SingleLaneBridge {

**component** CarFactory **is-factory**

**resembles** FactoryBase {

**attributes**: carId: int;

**ports**:

bridge; move;

**parts**:

c: Car

carNo (carId);

**connectors**:

b **delegates-from** bridge **to** bridge@c;

m **delegates-from** move **to** move@c;

}

**component** CarCreator

**implementation-class** bridge.CarCreator {

**attributes**: maxCar: int;

**ports**:

create **requires** ICreate;

}

**component** MultiCar {

**attributes**: nCars: int;

**ports**:

bridge; move;

**parts**:

cc: CarCreator

maxCar (nCars);

cf: CarFactory

carId = 0;

**connectors**:

c **joins** create@cc **to** creator@cf;

b **delegates-from** bridge **to** bridge@cf;

m **delegates-from** move **to** move@cf;

}

**component** MULTISLB **resembles** SLB {

**attributes**: nCar: int = 4;

**replace-parts**:

r0 **becomes** red: MultiCar

nCars (nCar);

b0 **becomes** blue: MultiCar

nCars (nCar);

d **becomes** d: Display

maxCar (nCar);

}

}

To produce this extension permitting multiple cars, some new program source code must be written to implement the CarCreator component, however, there is no requirement to access or modify any of the source code relating to the base SingleLaneBridge stratum. Access to the architecture description is sufficient to permit extension. Note that resemblance is also used to define CarFactory, which extends FactoryBase provided by the underlying backbone stratum. The stratum dependency graph for the extension is shown in Figure 3.



Figure 3 – MultiCarBridge Strata Dependency Graph

### Using Replacement

The single lane bridge Controller component works well until the number of cars increases to the point that a stream of cars, either red of blue, continuously occupies the bridge denying access to cars moving in the other direction. In other words, the Controller component is safe but not fair. Figure 4 shows how a revised Controller that does implement fairness is introduced.



**stratum** FairBridge

**depends-on** SingleLaneBridge {

**component** Controller`

**implementation-class** bridge.FairController

**resembles** Controller **replaces** Controller

{}

}

Figure 4 – FairBridge components

Replacement is so often combined with resemblance that we use a single graphical symbol (combined solid and fishbone arrow) to indicate this combination as shown in Figure 4. The diagram and text show that we are replacing Controller with a component that resembles it exactly with the exception of the implementation class that we have changed. We can now combine this fair bridge controller with the MultiCarBridge stratum as shown in Figure 5. The stratum FairMultiCarBridge represents a system that permits multiple cars and has a fair controller. The stratum contains no definitions of components or deltas; it simply indicates its dependences as shown in the Backbone text below Figure 5.



**stratum** FairMultiLaneBridge

**depends-on** FairBridge, MultiCarBridge {

}

Figure 5 – FairMultiLaneBridge Strata Dependency Graph

Figure 5 illusrates how strata are merged. This can of course lead to conflicts in definitions, which we discuss in the next section. In the case of the example, a more plausible scenario would be for the original base developer, or a third party, to export this new stratum to the multi-car developer who could import it as shown in Figure 6. If we were to write out the Backbone text again, it would now show a dependency on FairBridge rather than SingleLaneBridge for MultiCarBridge.



Figure 6 – Alternative Dependency Graph

# 3 A Model of Intrinsic Definition

The ability to evolve an existing architecture in a decentralized manner leads to the requirement to merge independently developed strata back into a unified architecture. In the following, in addition to defining resemblance, replacement and strata, we give the rules for merging strata – showing that structural errors that result from merging can always be corrected by adding component definitions. Since dependencies between strata govern the order of applying changes defined by resemblance and replacment, we begin with strata.

## 3.1 Strata

A stratum is a hierarchical module that owns and groups element definitions. An element is either a component or an interface. Each stratum must explicitly indicate its dependence on other strata.

A stratum *s* is a triple < *popt*, *D*, *E* > where:

* *popt* is either the singleton set {*p*} containing parent stratum p or the empty set Ø if *s* is not nested inside a parent stratum.
* *D* is the set of strata that *s* depends on.
* *E* is the set of elements owned by *s*.

If we view stratum *s* as the node in a directed graph with arcs to the set of strata *D* that it depends on then we can define the set *visD*(*s*) as the set of strata reachable from *s*. We require that the dependency graph is acyclic:



Two strata can share common base strata and still be evolved independently if neither has visibility of the other via their dependencies. We consider two strata that share a common base to be independent in this way if and only if:



In the simple case in which independent strata do not share a common base and consequently cannot evolve common elements, conflicts cannot occur due to merging.

In Figure 5, MultiCarBridge and FairBridge are independent strata that build on a common SingleLaneBridge stratum. Each is developed independently and then merged to form the unified stratum FairMultiLaneBridge stratum.

## 3.2 Elements

An element is either a component or an interface and is owned by a single stratum. An element is defined by its constituents – in the case of a component, these are ports, parts, connectors and attributes.

An element *e* is a sextuple < *σ*, *ropt*, *R*, Δ*+*, Δ*-*, Δ→>where:

* *σ*  is the stratum that owns *e*.
* *ropt* is either the singleton set {*r*} or Ø where *r* is a element from a different stratum that *e* replaces.
* *R* is the set of elements that *e* resembles.
* Δ*+* is the set of added constituents
* Δ*-* is the set of deleted constituents
* Δ→is a replacement relation that maps old consituents from elements in *R* to new constituents.

An element’s expanded structure, represented by a set of constituents, is obtaining by applying the addition, deletion and replacement deltas (Δ*+*, Δ*-*, Δ*→*) to the constituent structure of the elements *R* it resembles. Elements in the set *R* must be owned by either the stratum *σ* that owns *e*, or must be owned by strata in *visD*(*σ*).

The sextuple representing MULTISLB from section 2.2 is:

<*σ = MultiCarBridge, r =* Ø, *R =* {*SLB*}, Δ*+ =* {*nCar*},   
Δ*-* = Ø, Δ→ = {*r0* → *red*, *b0* → *blue*, *d* → *d’*} >.

Key to expressing evolution is the ability to replace element definitions as in FairBridge from Figure 4 where the sextuple representing Controller’ is:

<*σ = FairBridge, r =*{*Controller*}, *R =* {*Controller*}, Δ*+ =* Ø,   
Δ*-* = Ø, Δ→ = {*implementation* → *bridge.FairController*} >.

We require that in forming new element definitions using resemblance or replacement, reference is always made to the original definition rather than to definitions that replace others. This rule, which facilitates change to the dependency order of strata without invalidating delta relationships, is expressed as:



In addition, a stratum *s* cannot contain two elements that both replace the same element. Without this restriction, we could have multiple replacements within a stratum that would not be ordered by the dependency relationship that we will see in the following section is used to order replacements among multiple strata. This constraint is:



## 3.2 Resemblance Graph

An

# 4 Intrinsic Definition In Practice

Proposers of a new approach to software architecture description have the responsibility to demonstrate that it can be put into industrial practice. In this section, we discuss three significant barriers to doing so and outline how they are addressed. They are: tool support, compatibility with existing libraries and toolkits, and application to existing legacy software.

## 4.1 Tool Support

The Backbone ADL is supported by a graphical modeling tool Evolve and a runtime environment which instantiates and interconnects components from a Backbone description. Figure 7 is an overview of the elements that make up the modeling tool and runtime environment. The DeltaEngine layer implements the resemblance algorithm described in the previous section and is used in the modeling tool to build graphical representations of composite components from delta definitions and in the runtime environment to instantiate components from these same delta definitions. In addition to this interpreted runtime the modeling tool can optionally compile a “flat” description of the model as a builder class. In this case, no runtime environment is required. It should be noted that even the interpreted runtime environment incurs no overhead during execution of a system; it is only active at startup time when it directs instantiation and interconnection.

Figure