The Modelling and Analysis of Extensions to a Component-Based Architecture

Andrew McVeigh

First Year Report
Supervisors: Professor J. Magee and Professor J. Kramer

31st May 2006

Acknowledgements

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

Contents

Chapter 1

Introduction

1.1 Motivation

Software systems are increasingly based around architectures which allow third party developers to extend them by adding new components and customising existing components. We call this type of architecture a *component-based extension architecture*. This approach allows a system to be used for a family of applications, and extending it is possible without further involvement from the original developers. The original system, before it is extended, is termed the *base*. An *extension* is a packaged set of new components, along with instructions about how to customise and reconfigure [32] existing base components in order to integrate the new components into the architecture.

Many extensible systems use a variant of this type of architecture. For instance, plugin architectures [26] are a constrained form of component-based extension architecture, supporting a non-hierarchical component model with limitations on the adjustment of existing components. A plugin is, therefore, a constrained form of an extension. Eclipse [25] is a successful and sophisticated example of this approach, and has spawned a vibrant ecosystem of commercial and open source plugins [27]. Some of these plugins offer significant extra functionality over the base Eclipse product, forming base systems in their own right. Another prominent product using the plugin approach is Firefox [13].

By allowing third party developers the freedom to extend a base system, component-based extension architectures lead naturally to a situation where a number of independently developed extensions must be combined to form a new, coherent product. For instance, one developer might write a spell-checker extension for an extensible word processor, and another might write a thesaurus extension. Clearly, it is desirable to combine these into a new word processor product, featuring both a dictionary and a thesaurus. Unfortunately, combining independently developed extensions into a common architecture often results in unexpected conflicts and behaviour. We term this the *independent extensions problem*.

Now suppose that the base word processor system is evolved from version 1.0 to 1.1. We would like to add the existing thesaurus extension into this new version, but it has been written against the architecture of version 1.0 and various aspects of 1.1 are no longer backward compatible. It is not uncommon for existing extensions to no longer work correctly when applied to an evolved base architecture. We term this the *old extensions problem*. In plugin architectures it is generally not possible to even detect many types of problems and the plugins must simply be combined and manually tested [5].

There is evidence that these problems are of growing concern [5, 6].

A further issue is that extension developers are currently not afforded the same power to alter the system as the developers of the base. Extension developers are able to extend the system, in general, in the ways that the base developers have planned for. For instance, a developer might wish to customise the way that characters are displayed in the word processor. Unless a component in the base architecture provides an extension point to allow for this, the extension developer will find adding this feature to be virtually impossible. Of course, Eclipse-like architectures phrase the entire platform as a set of components, allowing for arbitrary replacement. This does not solve the problem, however, if the component to be replaced is large and the feature to be added is relatively small. It will not be practical to implement the feature, and even if it were this does not solve the old extensions problem when an upgraded form of the replaced component is delivered in version 1.1 of an evolved base. In general, existing component-based extension architectures do not deal well with unplanned extension. We term this the *unplanned extensions problem*. A variant of this problem is when an extension developer wishes to alter a code-level interface or code-level component from the base architecture. Generally, this will break the base system if it is even allowed.

These three problems make existing component-based extension architectures less extensible and more brittle than is desirable. Resolving, or ameliorating these problems would make these software systems more extensible and more useful.

1.2 Focus

The aim of this work is to model component-based extension architectures in a way that allows for the static analysis of the structural and behavioural combination of extensions. One important application of this analysis is to automate the detection of issues, and automate or guide their resolution.

The focus is on design-time analysis of extensions. This work does not consider arbitrary run-time reconfiguration or allow extensions to be dynamically applied to an architecture.

1.3 Approach

The approach taken is to model component-based extension architectures using an ADL, called Backbone, which has been specifically created to deal with the issues mentioned above.

Backbone is based heavily on the UML 2.0 [49] composite structure meta-model and the graphical form of Backbone is represented by various UML diagrams. One of the benefits of this approach is that the visual form is familiar to practising software engineers. The language also has a textual form.

A hierarchical component model is used, with leaf and composite components. This allows for the fine-grained decomposition of a system. By supporting multiple levels of abstraction, the hierarchical model offers greater flexibility for extension developers to replace smaller and more targeted base components at the correct level of abstraction than with a standard plugin architecture. This minimises the likelihood of overlapping structural changes between extensions and partially addresses the unplanned extensions problem.

Extending and evolving a system are modelled uniformly via extensions, giving extension and base developers the same power. This reduces the independent extensions problem and the old extensions problem to the same issue: the ability to reason about the combination of multiple extensions.

In order to analyse the behaviour of a system, each leaf component must specify its protocol, which describes the set of traces that it can produce. Further, each extension specifies goal-level properties that must be preserved when extensions are combined.

The three stages involved in combining extensions are:

1. Unifying the structural changes made to the base architecture by reconfigurations.
2. Checking that all component protocols are still observed.
3. Checking that all goal-level properties are still preserved.

The approach to resolving conflicts is to allow a further extension to correct the issues.

The reconfiguration instructions of an extension are described using an intuitive component-like construct, which represents architectural deltas. Modelling with deltas makes it possible to reason more clearly about structural changes due to conflicting extensions. Reconfigurations are not applied destructively to alter the base source. Instead, the reconfigurations are performed just prior to system execution to an in-memory representation of the architecture.

Tool support for modelling using Backbone is provided by a graphical case tool. This will allow extensions to be constructed as a conventional set of components, but will record a set of architectural deltas where appropriate.

1.4 Report Structure

Chapter 2 presents background information and alternative approaches to creating extensible systems. Chapter 3 presents the graphical and textual form of the Backbone ADL. Chapter 4 shows how extension combinations can be analysed. A key issue is shown to be the need to share previously unshared base components between extensions. This is partially facilitated by the translation of protocols into FSP [35] in order to detect behavioural problems.

Chapter 2

Background

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

2.1 Component-Based Software Engineering (CBSE)

A longstanding goal of software engineering has been the ability to efficiently and reliably construct software systems from prefabricated components. An early reference to the general concept and vision was outlined by M. D. McIlroy at the 1968 NATO Software Engineering Conference [37]. In this influential white paper, software production techniques are compared unfavourably to industrial manufacturing techniques in electronics and hardware. A key element is found to be the idea of a component or interchangeable part, which provides a level of modularity. The lack of support (at that time) for a component industry where component producers provide catalogues of parametrised components is taken as a sign of the lack of maturity of software production techniques relative to other fields.

Following from this, a vision of the production of systems through the customisation, "translitteration" and assembly of parametrised software components is proposed. The proposal outlines options such as space-versus-time tradeoffs for algorithms, and considers the need to automatically translate algorithms into different languages for different operating systems.

Apart from the generation and translation perspective, this vision of components is similar in many ways to the modern concept of a class library. The examples chosen are based very much around algorithmic concerns, and parametrisation and generation are proposed as ways to handle the level of choice required. The wider concerns of a component-based architecture, such as dynamic structures and multiple levels of architectural abstraction, are not discussed although dynamic memory allocation is briefly mentioned.

In essence, the wider vision remains largely unfulfilled and surprisingly elusive. Fundamental issues remain where even syntactic mismatches prevent component integration [23].

In [15], software engineering is compared to the production of muskets in the pre-industrial era, where hand crafting of parts was routine. The lack of interchangeability and standard methods of measuring component compliance to specifications are cited as being key impediments to the development of component catalogues.

2.2 Component Technologies

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

CORBA [48] provides a platform-independent distributed component model. Interfaces are specified using an interface definition language, which can then be compiled for a specific language and platform choice into stubs and skeletons. Component references can be passed via interfaces or turned into a textual form. Bindings between components are formed by passing references, and are not specified using connectors. A hierarchical model is not supported. A CORBA marketplace for reusable components did not occur, largely because the approach was focused on the integration of components implemented for heterogeneous platforms.

COM [8] is a Microsoft component standard focused on the creation of components running under the Windows platform. COM components can be written in a number of languages and interoperability is specified at the binary level. A registry approach is used for forming component connections, rather than explicit connectors. A hierarchical model is somewhat supported through a compositional approach that mandates delegation. COM and its successor ActiveX [43] have successfully produced a marketplace for Windows GUI components.

JavaBeans [46] is the client-side technology for the creation of reusable Java components. It is primarily focused on GUI components, and supports an event model where clients are notified of changes to state attributes. Connections are modelled as references from a JavaBean to clients interested in changes in the underlying state model. This supports a form of hierarchical composition through object aggregation. It does not support explicit service interfaces.

Enterprise JavaBeans (EJB) is the distributed server-side Java component model [45]. Connections are formed by using a registry to locate named components. Component hierarchy is not supported. As an alternative to EJB, so-called lightweight frameworks have become popular in recent years. The Spring framework [21] uses XML configuration files to explicitly connect interfaces between Java classes. This is known as dependency injection [18], and is analogous to a simple, non-hierarchical ADL.

The Open Services Gateway Initiative (OSGi) defines a component-based environment for the deployment of network-enabled services [4]. Facilities are provided to manage the deployment and life-cycle of components. Components can either be bundles (analogous to plugins) or services, and these may be updated or removed without restarting the system. Bundles may depend on and also provide interfaces, although OSGi does not provide an explicit connection model. Eclipse [25] uses an implementation of the OSGi standard for its plugin architecture.

With the above approaches, components do not have to explicitly indicate required interfaces, although Spring and OSGi offer a level of support for this. References to a required component are generally retrieved through a name via a registry or through some other means.

2.3 Architecture Description Languages

Architecture description languages (ADLs) evolved out of the desire to explicitly describe the architecture of a software system as a set of connected components. Many ADLs exist, and the general consensus is that they must support composition of components, explicit connectors and

provide an underlying formal model which can be analysed [41]. Components must specify all required and provided interfaces. Most ADLs also use a hierarchical component model.

Darwin [34, 33] is an ADL used for specifying the architecture of distributed systems. It offers support for multiple views and allows information to be added to the architectural specification in order to aid analysis. The separation of the structural description from the implementation is often referred to as a configuration-based approach. Behavioural information is specified using FSP [35], a process calculus. The Labelled Transition System Analyser (LTSA) tool can process FSP into labelled transition systems, allowing deadlock checking, the checking of safety and progress properties, and other types of analysis to be performed.

Dynamic reconfiguration of architectures specified in Darwin is explored in [32]. This work considers how to accommodate evolutionary change to the structure of an architecture, whilst the system is running. Configuration changes are modelled as deltas: component and link creation and removal are supported. The property of system quiescence, indicating that a given component is not currently involved in a transaction, is used to determine when to transfer application state from the old component to the new one.

C2 [58] is a non-hierarchical ADL designed to support the explicit requirements of GUI software, including the reuse of GUI elements. A component has a top and bottom domain. The top domain specifies the notifications that the component can accept and also specifies the requests that are issued to the rest of the architecture. In essence, this domain models provided interfaces for the handling of notifications, and required interfaces for issuing requests.

The bottom domain indicates the notifications that will be emitted, essentially modelling the required interfaces for sending events to the rest of the architecture. Each domain may be connected to only one connector, but connectors can accept links from many components. A key principle is one of substrate independence, where the component knows of the components that are connected to its top domain, but does not know which components are connected to the bottom domain. Pictured visually, an architecture can be reused by slicing it horizontally at a certain level and taking the components and connections above this level as a reusable set. Each C2 component may be active, with its own thread.

C2 SADL [38] is a variant of C2 designed to support dynamic instantiation of components. It also supports upgrading components in a running system and removing unwanted components via reconfiguration. C2 SADL defines the concept of placeholder components [39], for representing conceptual entities which have not been fully elaborated. This provides support for top-down design in addition to bottom-up construction of an architecture. Later work in this area developed C2 SADEL [40] which explicitly provides support for analysing the architectural evolution of a system. The protocol of a component is modelled via explicit state, invariants, and pre and post-conditions. Further, this approach is used to implement the design environment and language, demonstrating its applicability.

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

ROOM [54] is an ADL for modelling and constructing real-time software systems. Components are called actors, and must explicitly specify any required or provided interfaces via ports. Actors may be hierarchical and active. Actor protocols are described using extended state machines (ESMs), which are a type of automata allowing variables. The presence of variables makes it difficult to analyse protocols using model checking, as the state space can be very large indeed. From an

engineering perspective, ESMs are more attractive to design with because the number of states is usually far smaller than in the equivalent finite automaton. ROOM also includes a concept of structural inheritance that effectively allows for component reconfiguration in a sub-actor.

ROOM outlines a pragmatic and wide-ranging vision of CBSE which includes a virtual machine for model debugging and execution, and tools to translate models into implementation languages. It has had a far reaching impact on graphical modelling techniques, and is one of the key influences that led to the vision of Model Driven Architecture (MDA) [50]. The ROOM approach and similar techniques have been successfully used in the real-time software arena for many years.

2.4 The Unified Modelling Language 2.0

The Unified Modelling Language (UML) is a graphical language for describing the structure and behaviour of object-oriented systems [49]. UML was standardised by the Object Management Group (OMG), which subsequently evolved it from version 1.0 through to version 2.0.

UML2.0 introduced a number of diagram types which allow it to be feasibly used as the basis of an ADL [20]. The design of these diagram types have been heavily influenced by ROOM and several other ADLs. Composite structure diagrams can be used to model composite and leaf components. Component diagrams are also provided, but these are essentially a variant of composite structure diagrams and are not considered further here.

Component protocols in UML2.0 are modelled either as sequence diagrams [53] or activity diagrams, with the latter seemingly preferred. Essentially, this amounts to modelling protocols as ESMs, with the same engineering advantages and limitations on their formal analysis.

UML represents an amalgam of loosely integrated techniques and graphical diagrams from many areas, showing its heritage as standard designed by a committee. Part of the challenge of applying the UML to a software system is to choose an appropriate subset of the language and give it a more precise meaning.

2.5 Approaches to Extensible Systems

In an influential article [51], scripting is presented as a way to integrate components implemented in conventional compiled languages. The argument is that scripting languages are more flexible and faster to develop in than their statically typed and compiled counterparts. This approach can also be used to produce systems which are extensible via customisation of scripts.

Many systems have opted for a scripting approach in order to provide an extensible base which supports the introduction of new features by extension developers. A prominent example of a platform with an important scripting focus is Excel [44] which uses Visual Basic for Applications as the scripting language. The scripts can call out to DLL-based components, which can be added to the system.

Scripting approaches do not address the independent extensions problem, although they tend to minimise it as extensions are only given limited freedom to adjust the system.

Frameworks are a reuse technique allowing a semi-complete application to be customised and extended into a complete application [17]. A framework allows objects implementing particular

interfaces to be registered with it, and will invoke these objects at various points in its processing. This is known as inversion of control [28]. This offers an effective and large-scale form of reuse, but in practise a number of fundamental issues limit this approach [12]. Many of these issues stem from ownership of architectural changes, where changes to a framework must be performed by the base developers, or else a copy of the framework's code must be taken and independently modified leading to subsequent maintenance problems. In addition, evolving a framework is difficult because the base framework is unaware of all of the different uses that it is being put to, leading to a variant of the old extensions problem. Various techniques have been proposed to mitigate this situation [14, 24, 36].

Plugin architectures have one or more frameworks at the core of their architecture, but are generally packaged as an application [26]. Plugins represent components that can be added to the platform by adjusting the system configuration. Large extensions to these systems represent platforms in their own right, and can literally consist of hundreds of plugins. Plugin architectures support a constrained form of component-based extension architecture, as described in the introduction, and have problems with the combination of independent extensions.

Aspect-oriented programming [31] aims to allow orthogonal features to be developed and woven together at a later stage. The main issue with this approach is that although it works well with truly separate concerns such as transactions and logging, it does not provide a convincing approach to handling features which are designed to interact. In addition, there has been little take up of this approach for the mainstream evolution of systems.

Version control systems have been successfully used to manage and evolve component architectures [61, 55], and offer another approach to extending a system. Combining structurally conflicting extensions has the disadvantage of requiring the developer to fully understand the source code and any changes made in order to perform a sensible source code merge. It also offers no guarantees that the properties of the each extension will be preserved in the combined system or that apparently independent extensions will not behaviourally interfere. An approach to mitigating the limitations of this is presented in [11]. Currently the work is focused on creating architectural deltas in order to structurally merge extensions to a product line architecture. No support is offered for detecting or resolving behavioural interference, although this is mentioned in future work.

Mixins are a language feature for expressing abstract subclasses that can be reused in different parent classes [9]. Any number of mixins can be combined into a parent class, and methods of the mixin may invoke methods of the that class. This implies that the mixin must make assumptions about the names of the parent methods that it will call, which represents an integration issue. Further, multiple mixins may conflict or interact in unforeseen ways. Scala is a language with mixins which aims to support the combination of independently developed extensions [63]. It provides two dimensions of extension: data extension for the object-oriented view, and behavioural extension for the functional view. It is not clear that Scala offers any support to alleviate the naming problem suffered by mixin approaches. Further, the combination of independent extensions must in part be performed by manual adjustment of the class that combines the mixins.

A product-line architecture focuses on a set of reusable components that can be shared by many systems in a product family. Concurrent evolution of single components of the product line is problematic [56], and work to merge and propagate changes from one branch to another is still required if the code for a component is branched [11].

2.6 Approaches to Integration of Independently Developed Components

The problem of integrating independently developed components is well known, and the issues range from simple syntactic mismatches through to more complex behavioural interactions and testability worries [23, 30, 57, 55]. Component versioning and deployment approaches have been proposed as a way of mitigating this [42], where multiple versions of a component may be deployed and present in a running system. This does not address the issue of migration to the newer version, or the problem when a single version of the component must always be enforced. The latter situation often occurs when a component is managing a resource that requires a single controlling entity in a system.

A common theme of several approaches is to solve the integration issue by wrapping the original component and delegating selectively to it. Wrapping is proposed in [23] in order to adapt an interface for naming mismatches, although it is pointed out that this introduces a performance problem. It also introduces a problem with identity as both the wrapped object and the original need to assume the same identity in some situations. The desire to wrap components is more focused on solving the problem at an implementation level, where maintaining compatibility with existing languages and paradigms is either implicitly or explicitly considered to be of paramount importance. Superimposition [7] is a variation on this theme which aims to address the identity problem (referred to also as the "self problem"). The aims of adaptation, in this case, are that it can be applied transparently, and that the wrappers can be reused in other contexts. This does not solve the fundamental problem that wrapping is a black-box reuse technique which cannot adjust fundamental characteristics of a component, only hide them. Other wrapping approaches are outlined in [29, 59].

Feature composition has been proposed to allow the addition of independently developed features to a telecommunications system, where existing services must not be affected in an adverse way [22]. This approach uses a series of relational assertions to model states and events, along with invariants which characterise the intended effects of the added features. Feature interaction is detected at runtime by examining whether a higher priority feature has caused the invariants of a lower priority feature to be invalidated. A lower priority feature is not allowed to violate the invariants of a higher priority feature. This work does not explicitly mention component structures or connections, and is not directly applicable to a hierarchical component-based architecture. Furthermore, the work is focused on finding the best possible run-time resolution of conflicting requirements rather than supporting a developer in understanding and resolving unwanted interference. The steps for expressing and resolving interactions have to some extent been mirrored in the Backbone approach, with some modifications required for the design-time focus and hierarchical component model.

A model of a plugin architecture has previously been presented in [10], using a process formalism to ensure that properties hold when plugins are combined. This work does not deal with hierarchical component structures. It handles the situation where multiple components share a single service at a level of abstraction that assumes mutual exclusion of resources and services.

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

2.7 Behavioural Modelling of Components

The formalisation of object protocols is considered in [47], which argues that it is essential to view the protocol of an object as a process rather than as a function. The notion of a service type is introduced, which characterises the traces that an object can produce in terms of operation actions. This is assumed to conform to a regular language, such as that produced by a labelled transition system. The service type is contrasted with the notion of a value type, which summarises the argument types and return type of a method.

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

Component protocols are modelled with regular expressions in [52]. The interaction context is the hierarchical SOFA model, where connectors join provided and required interfaces of components. Each event in the protocol language is either a request or response, and can be either emitted or absorbed. This corresponds closely to the modelling of client-service protocols presented in an example in [35], which uses either request or reply and either call or accept as action prefixes.

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

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

The plugin model of [10] uses FSP to describe the behaviour of components in a system. The underlying assumption appears to be that the events of a client and service component will be regarded as shared actions if possible. This prevents a client from emitting an action if the service is not prepared to accept the response, limiting the set of errors that can be detected to no activity errors. These types of errors are manifested as deadlocks when performing an LTSA analysis of the composed protocols.

Chapter 3

The Backbone ADL

> *Any time you build an extensible system ... the gist of it is that you can't predict in advance what changes your users will want to make to your system. You can go to any length even expose every single line of code in your system as its own virtual function and your users will inevitably run into something that they can't specialize.*
> Steve Yegge [62]

3.1 Overview

Backbone is an ADL designed to model component-based extension architectures. The focus is on reducing the limitations on extensions, and allowing for static analysis of the structure and behaviour of combinations of extensions.

Backbone is based on a simplification of the UML 2.0 composite structure meta-model [49]. The simplifications permit arbitrary visual nesting of internal component structure. Component structure is depicted graphically via composite structure diagrams and component protocols are depicted as sequence diagrams.

A benefit of using a subset of the UML is that many software engineers are familiar with this visual language. This reduces the barrier of entry for practising engineers, whose systems could potentially benefit from the types of analysis that Backbone allows. In the examples below, the textual form is sometimes shown to the right of the graphical form.

3.2 Strata

All Backbone definitions and constructs are contained within a package-like construct known as a *stratum*. Stratum are not hierarchical—they cannot be nested. A stratum must explicitly declare its dependencies on other strata. A stratum can represent an extension, or it can represent the base architecture. The concepts of extension and base become relative as extensions are further extended. Circular dependency structures are forbidden.

Figure 3.1 shows a base stratum, with two extensions. The resolution stratum resolves any conflicts that result when the two extensions are combined.

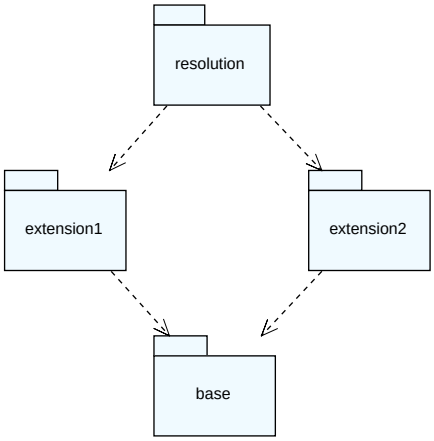


Figure 3.1: Strata

3.3 Interfaces

An interface specifies a collection of operations. It represents a service that can be provided or required by a component. An interface is shown graphically as a small circle, with the operation names optionally listed underneath. An operation is not considered threadsafe (synchronised in Java) unless explicitly declared as such. Each operation is *synchronous*, and even if no value is returned, a *call action* is always followed (eventually) by a *return action* in order to model Java's method call semantics.

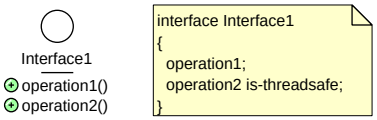


Figure 3.2: An interface with operations

Each Backbone interface must map onto a corresponding Java interface of the same name. Each Backbone operation must map onto an operation of that Java interface.

3.4 Component Model

A *component* must explicitly specify its *provided* and *required interfaces*.

A *leaf* component (figure 3.3) has *attributes*, *ports* and a *protocol*. It always has a corresponding Java implementation, which the Backbone definition describes the form and behaviour of.

Properties are analogous to the attributes of a class. The type of an attribute may only be a primitive type such as *Integer, Double, String* or some other user-defined type.

A port describes the interfaces that a component provides or requires and are shown as small boxes on the component boundary. Provision of an interface is shown by a line to the interface. A required interface is shown by a line to a half-circle representing the interface. Ports may have a *multiplicity*, in which case they are called *indexed*. A multiplicity of [0..*] indicates that zero or more connections are allowed. A multiplicity of [0..1] indicates that connecting the port is optional. A component cannot provide or require an interface except via a port.

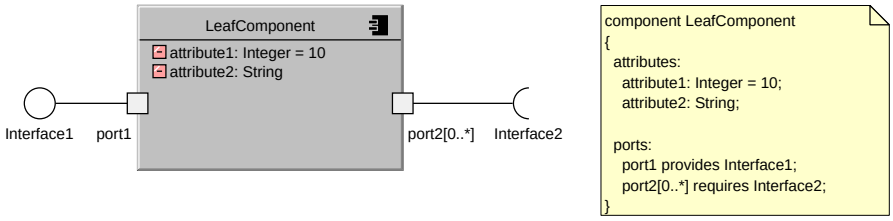


Figure 3.3: A leaf component

A hierarchical component model is used, allowing *composite* components (figure 3.4). A composite describes its internal structure as a set of component instances, which are known as *parts.* A part has a name and a type. A part may only have a multiplicity of 1. The attribute values of a part are called *slots.*

Connectors are used to wire between ports. To connect between indexed ports, a delegation connector must be used.

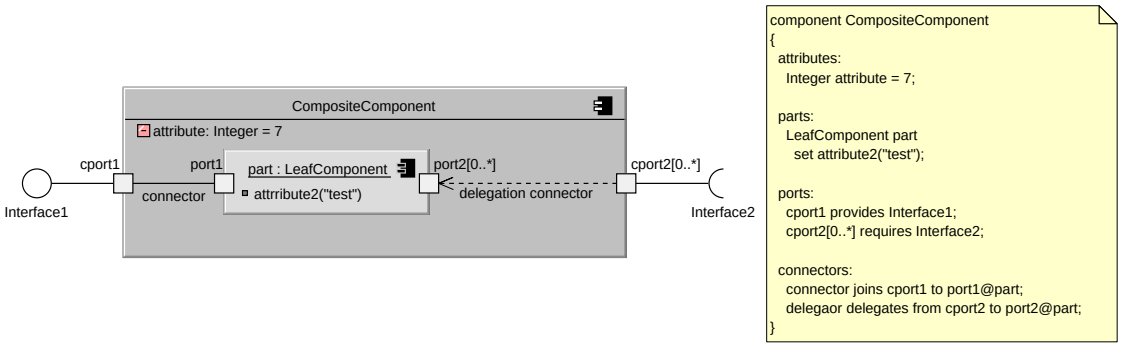


Figure 3.4: A composite component

A composite component has no intrinsic behaviour of its own—it is a convenient way to refer to the set of connected leaf parts that it describes. In figure 3.4, the composite component is shorthand for a leaf component where the external ports have been renamed to *cport1* and *cport2.* Composite components can also have attributes.

A slot of a part can be marked as *environment* which means that it will be aliased to a attribute in the enclosing component. In figure 3.5, *attribute2* of *partA* and *partB* have been made into environment slots. The attribute *topLevel* of the composite is now the actual attribute, although it can be set or read in all three components. This facility is useful because it allows a composite component to hold the shared state of the combined parts, and to map this selectively onto each part's attributes.

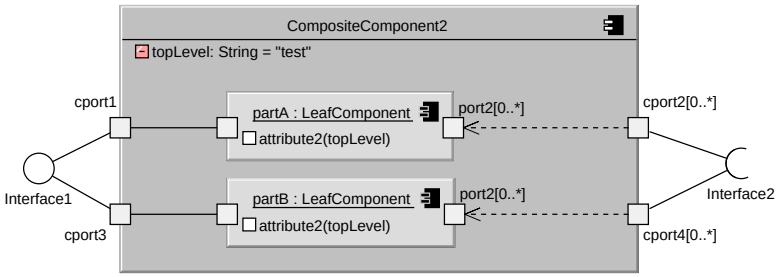


Figure 3.5: Environment slots / attributes

Arbitrary visual nesting of parts is supported. Figure 3.6 shows a composite component which has been visually expanded out to its leaf parts over two levels of composition.

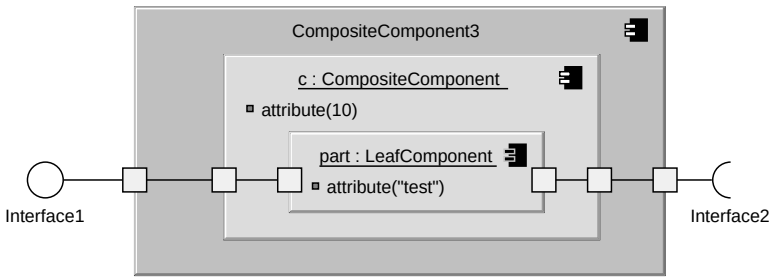


Figure 3.6: Two levels of visual nesting

3.5 Component Protocols

A leaf component can specify a protocol, which describes the set of traces that the component can generate. The graphical form of a protocol is a sequence diagram where the actors are the interfaces of the component's ports, along with the component itself. A component making a call to a required interface is shown as an arrow originating from the self actor (representing the component) and terminating at a required interface. A component handling a call from a provided interface is shown as an arrow originating from the provided interface and terminating at the self actor.

All communication is synchronous, reflecting Java's method call model. Each call action must be followed by a return action. A call is shown as a solid line with a solid arrow. A return is shown as a dotted line with a line arrow. To express protocols more concisely, a return immediately following a call may be overlapped to form a double headed arrow.

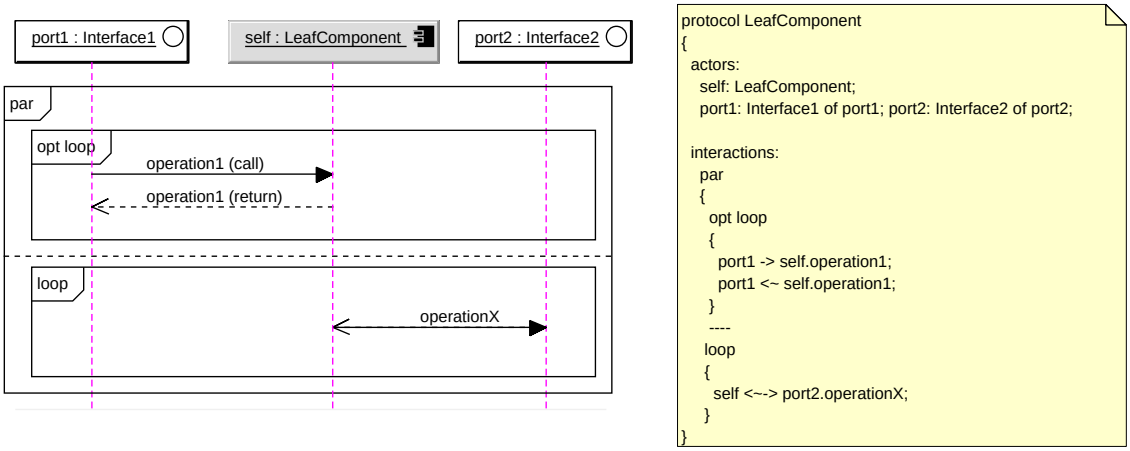


Figure 3.7: The LeafComponent protocol

As shown in figure 3.7, a protocol may use operators such as par, opt and loop [53]. The arguments to an operator are called operands. Each operand consists of a sequential series of operations or other operators. Operands are demarcated using a horizontal dotted line. Some operators, such as par may need two or more operands.

The allowed operators and their informal meanings are as follows:

- *Seq*
  This represents sequential execution of the operations contained in the single operand. A protocol is contained within an implicit seq operator.

- *Par*
  Each operand represents a series of operations, to be run in parallel with the operations of the other operands. When the par operator is entered, all of the operands execute in parallel. This operator does not terminate until each operand has fully completed.

- *Loop* and *loop(n)*
  Loop(n) represents a finite loop, which will execute at least once and will eventually exit. Loop without a parameter represents an infinite loop. There are some constraints around infinite loops—for instance, an infinite loop cannot be nested inside another infinite loop. Also, no operations may follow an infinite loop as these could never be executed.

- *Alt*
  An alt operator models conditional execution, and supports two or more operands. Exactly one of the operands will be executed.

- *Opt* and *opt(port)*
  The single operand of this operator will optionally be executed. If opt(port) is used, the entire operator will be either statically omitted or turned into a seq operator depending on whether the specified port is bound in the composite component whose protocol we are modelling.

Operators may be concatenated (e.g. opt loop) if the prefixed operators only require a single operand.

The protocol in figure 3.7 has two operands that will be executed in parallel: the first operand is an optional, infinite loop where the component repeatedly accepts operation1 calls and sends out the return. The second operand is an infinite loop where the component repeatedly sends operationX calls, and receives the return.

Protocols do not allow variables, which allows them to be translated into a labelled transition system with a relatively limited number of states. This facilitates analysis, as will be demonstrated in the next chapter.

3.5.1 Leaf and Composite Protocols

Leaf components will generally specify a protocol. The protocol for composite components is formed by flattening out the part graph down to leaf components, and composing each of the leaf protocols together, after taking into account connection information. See chapter 4 for details of this procedure.

If a composite component includes a leaf part without a protocol, it is permissible to specify the protocol of the composite directly. This provides support for top-down elaboration of a system as well as bottom-up construction.

3.6 The Upversion and Supplant Constructs

The aim of these two constructs is to fully represent all possible changes in the evolution and extension of a system. This includes modelling changes to code-level interfaces and components. Note that these constructs must be in a different stratum to the definitions that they operate on.

The *upversion* construct models changes to the implementation of an interface or leaf component. This must be used whenever a code-related artifact is changed. Upversion supports renaming if required. In addition, it is possible to describe the new version in terms of the previous version by using the *is-like* construct. Is-like allows a component to be described in terms of differences to an existing component, and supports adding, deleting or replacing attributes, ports, parts and connectors.

Upversion changes a definition globally, and reflects the fact that the previous definition is now completely removed. This models the way that a system can only have a single version of an code-level interface or leaf component at any single point in time. Backbone tracks the dependencies, and requires that other components that directly depend on the changed artifacts are also upversioned or supplanted as appropriate.

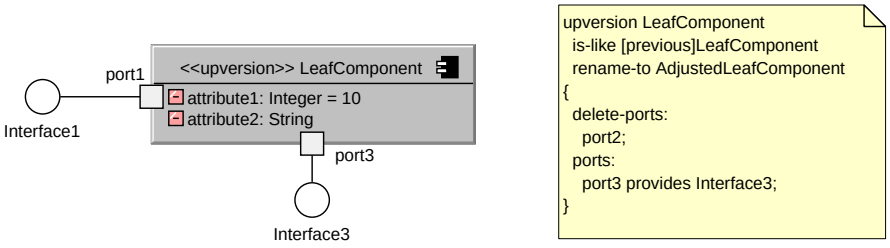


Figure 3.8: Upversioning a leaf component

Interfaces, leaf components and composite components may also be *supplanted*. Supplanting replaces the logical Backbone identity of the name with another definition. A leaf or composite component can be supplanted with either type of component. The old definition still exists to describe existing code-related implementations, but is replaced by the new definition from the perspective of the Backbone description. It is still possible to refer to the previous definition via *[previous]*, in order to delegate to it.

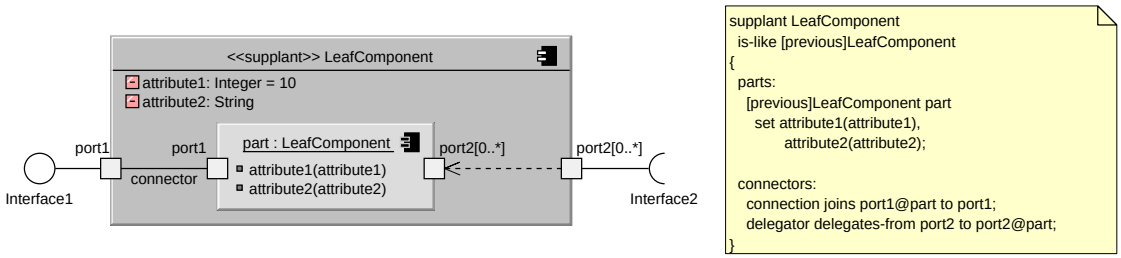


Figure 3.9: Supplanting a leaf component and making it composite

As no code related artifacts are changed, supplanting an interface can result in a break between the implementation of a component and its Backbone definition. Backbone tracks these dependencies and requires that affected components must be upversioned or provided with adapters.

The difference between the upversion and supplant constructs is that upversion models changes to the code-level implementation and updates the Backbone definition to match. Supplant models

changes to the Backbone definition of a construct only, and is a way of altering the meaning of a logical Backbone name.

3.7 Goal-Level Extension Properties

The form of these properties has not been specified yet. They may be represented as sequence diagrams showing traces that must be able to be generated by the system. The aim of these properties is to ensure that the goals of the extension are preserved, and that unwanted interaction or traces do not occur. Goals can also be expressed for the base architecture.

3.8 Advanced Features

Backbone has a number of other advanced features which are not described here in detail, in order to simplify the presentation. Dynamic component instantiation is possible via factory components [19]. This is supported by a navigability model, describing how references to one component may be acquired from another. The passing of component references as parameters of an interface call is strictly controlled. Ideally, a system will not allow this and the connection model will show all possible component interactions. This makes static analysis of the architecture far more feasible.

Finally, a form of type inferencing will be supported in order to allow the creation of more flexible components [16]. We would like to be able to omit explicit definition of provided and required interfaces for some ports, where it is desired that the interfaces be discerned from the connectors of the enclosing composite structure.

3.9 Summary

Backbone is an ADL designed to model component-based architectures and extensions to an existing base. It provides a set of constructs for representing changes to the implementation of the leaf components, as well as for modelling changes to the Backbone definitions. The use of the is-like construct enables a component to be described in terms of structural deltas from another component.

A full analysis of the limitations of Backbone must be conducted, from the perspective of the types of code-level implementation changes allowed to a system in an extension. It is currently understood that more comprehensive support for the evolution of interfaces must be provided, as the evolution of value objects passed as parameters of an interface are not currently tracked. In addition, the Backbone approach to system implementation implies that there are restrictions on how interfaces visible to Backbone can be used internally in the implementation of a component. If a component does not directly expose a Backbone visible interface as being provided or required, the implementation of that component may not use the interface internally. Otherwise, the interpreter cannot track the need to upversion the implementation of the component when the interface is upversioned.

Chapter 4

Analysing the Combination of Extensions

In order to analyse the combination of extensions, we must perform the following steps:

1. Unify the structural view after considering reconfigurations.
2. Check that component protocols in the unified architecture are observed for all leaf and composite components
3. Check that goal-level properties are preserved.

These steps correspond loosely to the representations and steps presented in [22] in order to handle conflicting runtime transitions and invariants in a telecommunications system with many independently developed features.

4.1 Forming a Unified Structural View

Each extension can potentially reconfigure the base architecture. Because of this, it is easy to arrive at a situation where each extension expects a different structure of the base, and if we are to combine the extensions we need to form a unified structural view.

Structural conflict can be detected by flattening the graph down to connected leaf components for each extension and comparing. The delta information provided by the reconfigurations is not sufficient alone to be able to resolve the extensions into a single view, however. In this case, we propose allowing deltas to be annotated to indicate why changes were made. A key theme that has emerged in the example systems modelled so far is one of sharing services: problems occur when a component which expects sole access to a service is forced to share with another component. It appears that much of the time, the reason for a structural change is to permit this form of sharing. Our approach is to expand on this to allow sharing components to be annotated with information representing sharing patterns, allowing the merge algorithm to deal with these situations automatically. If the architecture cannot be unified automatically, the developer will be guided through the process by a CASE tool.

When two components are forced to share a single service, they often together violate the intended protocol of the service. To detect this situation, we require protocol information to be specified for leaf components.

4.2 Modelling the Protocol of a Leaf Component

A leaf component's protocol is specified using a subset of UML2.0 sequence diagrams (see section 3.5). No variables or extended state are allowed, which allows the behaviour to be translated into an automaton with a (reasonably) manageable number of states. One of the aims is to construct a composite component's protocol from the composition of the protocols of its constituent parts, and limiting the states allows this to be accomplished in an acceptable time.

We translate the protocols into FSP [35], a process calculus intended for describing and reasoning about concurrency. The LTSA tool is used to translate an FSP programs into a labelled transition systems (LTS). An LTS is a non-deterministic finite automaton (NFA), as two or more transitions from one state are allowed to have the same action label. FSP also includes the ability to force an LTS to be deterministic. The deterministic form is needed for checking for bad activity errors[2], as will be demonstrated below.

As a working example, consider the translation of an example component. The TimeManager component is from the architecture of a digital clock. At the top of figure 4.1 is the composite structure view of the TimeManager leaf component, showing that it has two ports: mgr, and clock. The mgr port provides the ITimerClient interface (for receiving a notification every second) and requires IDisplay (which represents a display to show the new time on). The clock port is provided for use by a client to adjust the time.

The lower part of the figure shows the protocol. The actors are the component (in grey) and the interfaces for each of the ports. To the right is the Backbone textual equivalent.

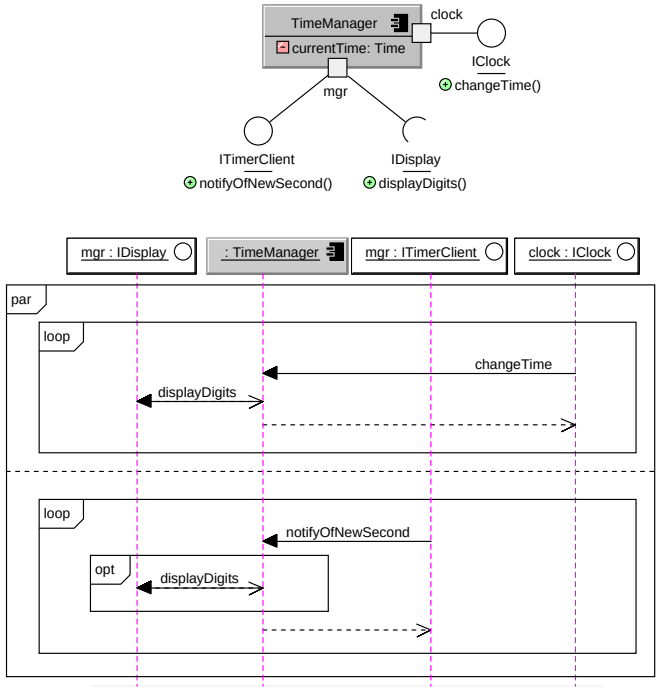
4.2.1 Sequential Operations

A sequential list of operations is modelled using a single FSP process consisting of actions to model the operations. In order to represent the differences between a client calling a service, a service receiving the call, the service sending the return and the client receiving the return, we use the following action prefixes:

- call.tx (the client calls a service)
- call.rx (the service receives or accepts the call)
- ret.tx (the service transmits the return)
- ret.rx (the client receives the return)

When two components' ports are connected together, the FSP from each protocol is composed together and the call.rx will be relabelled to call.tx and ret.tx will be relabelled to ret.rx. This ensures that a call from a client will always be handled by the service and that the return from the service will always be accepted by the client.

The list of sequential operations from the first loop of the protocol translates into the following FSP snippet. This will be combined into an FSP process at some point.




```
component *TimeManager
{
 properties:
  Long currentTime;

 ports:
  mgr provides ITimerClient requires IDisplay, ITimer;
  clock provides IClock;
}

protocol TimeManager
{
 actors:
  self: TimeManager;
  display: IDisplay of mgr; timerClient: ITimerClient of mgr;
  clock: IClock of clock;

 interactions:
  par
  {
   loop
   {
    clock -> self.changeTime;
    self <~~> display.displayDigits;
    clock <~ self.changeTime;
   }
   ----
   loop
   {
    timerClient -> self.notifyOfNewSecond;
    opt
    {
     self <~~> display.displayDigits;
    }
    timerClient <~ self.notifyOfNewSecond;
   }
  }
}
```

Figure 4.1: The TimerManager leaf component and protocol

```
clock.iclock.call.rx.changeTime -> mgr.idisplay.call.tx.displayDigits ->
mgr.idisplay.ret.rx.displayDigits -> clock.iclock.ret.tx.changeTime ->
```

Note that further prefixes of the port name and interface name have been added. This is necessary to prevent unwanted action synchronisation, in case multiple ports provide or require the same interface, or multiple interfaces on a port share the same action names. The port / interface combination is guaranteed to be unique within the scope of a component.

Threadsafe operations (Java synchronised methods) are modelled with a unique start and end action that surround the call. This does not model re-entrant locks but this situation is not considered to be common in protocol descriptions. Both the client and service must include these actions. In Java, the implementation of a method is synchronised, not the interface method. Backbone requires the interface method definition to be marked as threadsafe so that both the client and service know to include the start and end actions.

```
syncstart_1 -> clock.iclock.call.rx.changeTime -> syncend_1 ->
```

4.2.2 Par Operator

The par operator is translated by breaking out of the current FSP process and forming a process for each operand, which will be composed together with all other generated processes in the last stage of forming the protocol. Each operand starts with a shared synchronisation action, in order to guarantee that operations from all operands will start executing concurrently. Each operand ends with a shared synchronisation action, in order to guarantee that the operator is only exited when all the operands have completed.

The following text shows how the two parallel operands are translated into FSP processes.

```
Par1 = (_.start.a -> _.start.b -> _.end.b -> _.end.a -> Par1).
Par3 = (_.start.a -> _.start.c -> _.end.c -> _.end.a -> Par3).
```

The process that includes the par operator must synchronise on the start and end actions, in order to compose correctly:

```
TimeManager_internal = (_.start.a -> _.end.a -> TimeManager_end_loop).
```

Both Par1 and Par2 contain internal start and end symbols. These have been inserted to synchronise with the internal loop operators of each operand. If each par operand contained only sequential operations, these would be included directly in the Par1 and Par3 processes.

4.2.3 Infinite Loops

The loop operator (with no parameter) allows an infinite loop to be expressed. There are several constraints governing the use of infinite loops. For instance, it is not possible to nest them. No actions are allowed to follow an infinite loop, as these could never get executed.

An infinite loop is translated into two processes: the first process contains a synchronisation action, and the second process loops indefinitely. The second process must have the alphabet extended to include an end synchronisation action in order to compose correctly with its parent process. The first infinite loop in the protocol translates to:

```
Iloop4 = (_.start.c -> Iloop4_loop),
Iloop4_loop = (comp_4.mgr.itimerclient.call.rx.notifyOfNewSecond ->
_.start.d -> _.end.d ->
comp_4.mgr.itimerclient.ret.tx.notifyOfNewSecond -> Iloop4_loop) + {_.end.c}.
```

Several points are worth noting. First, the mgr.itimerclient.call.rx.notifyOfNewSecond action has had a further prefix added. This is referred to as a composition prefix and prevents the action from being shared with another parallel region. It models that operations from parallel operands of the same protocol are not synchronised.

4.2.4 Finite Loops

If the loop operator has a parameter, it is regarded as finite—i.e. the loop will execute one or more times. This is modelled using the FSP choice operator to either go back to the start of the loop or exit. For instance, if the first infinite loop were actually finite, it would be translated to:

```
Floop2 = (_.start.b -> Floop2_loop),
Floop2_loop = (comp_2.clock.iclock.call.rx.changeTime ->
comp_2.mgr.idisplay.call.tx.displayDigits ->
comp_2.mgr.idisplay.ret.rx.displayDigits ->
comp_2.clock.iclock.ret.tx.changeTime -> Floop2_exit),
Floop2_exit = (_.mytau.a -> Floop2_loop | _.end.b -> Floop2).
```

The mytau action represents a dummy action which will be hidden at a later stage.

4.2.5 Opt and Alt Operators

These operators represent choice, and are modelled using the FSP choice operator. The opt operator in the protocol translates to:

```
Opt5 = (_.start.d -> _.end.d -> Opt5 |_.start.d ->
comp_5.mgr.idisplay.call.tx.displayDigits ->
comp_5.mgr.idisplay.ret.rx.displayDigits ->
_.end.d -> Opt5).
```

Although the process loops, it will only start again when the parent process that it is composed with shares the _.start.d action.

A variant of the opt operator is opt (port) which will include the operand if the port is bound when used inside a composite component. If the port is not bound, then the operand is not included in the translation.

4.2.6 Composing the Processes

In the final step, the processes are composed together and any start and end synchronisation actions and dummy actions are hidden. Any composition prefixes are also suppressed. The final LTS of TimeManager contains 22 states, and the graphical representation is complex. The transition view of the LTS is somewhat more easy to read:

```
TimeManager = Q0,
Q0 = (c.clock.iclock.call.rx.changeTime -> Q1
|c.mgr.itimerclient.call.rx.notifyOfNewSecond -> Q2),
Q1 = (c.mgr.idisplay.call.tx.displayDigits -> Q3
|c.mgr.itimerclient.call.rx.notifyOfNewSecond -> Q4),
Q2 = (c.mgr.itimerclient.ret.tx.notifyOfNewSecond -> Q0
|c.clock.iclock.call.rx.changeTime -> Q4
...  (many lines omitted)
```

4.2.7 Multiple Protocols for a Single Leaf Component

The single protocol models the way that the component can handle parallel calls of changeTime and notifyOfNewSecond. However, analysing the LTS shows that the protocol produces traces where a displayDigits call is immediately followed by another displayDigits call without waiting for the first return to arrive. This does not reflect the implementation of the component, which is either in “set time mode” or “display time mode”.

To correct this, we can express the mode directly in the protocol using an alt operator, which involves a more complex description. Alternatively, we can express an additional protocol which will be composed with the first, allowing us to further limit traces. The FSP translation of this is:

```
Display_dependency = (c.disp.idisplay.call.rx.displayDigits ->
c.disp.idisplay.ret.tx.displayDigits -> Display_dependency).
```

Composing the main protocol with additional protocols allows us to express dependencies which are not possible or simple with a single protocol.

4.3 Modelling the Protocol of a Composite Component

By composing the protocols of the internal parts of a composite component and hiding the internal actions, we can form an LTS for the composite. If any parts do not compose together well, we will get transitions to the error state (-1). Hiding all but the externally visible actions greatly simplifies the resultant LTS.

A goal is to turn the composite component's LTS back into a sequence diagram. Because the consent operator preserves regularity over composition[2], this is definitely possible to do although the resultant diagrams may not be particularly attractive or useful. Based on the example problems worked through so far, the composite LTS diagrams are very simple and only become complex when there are complex dependencies between externally visible actions.

In [2], three error types are shown to occur when composing protocols: no activity, bad activity and divergence. The first type can be detected by using FSP composition and analysing the resultant LTS for deadlock, as in [10]. The second type can be detected by using FSP composition of LTSs which have had additional error transitions added. The third type is applicable for runtime component updates, and is not discussed further here.

The following section considers how to form the protocol of the composite Clock component, which combines the TimeManager and Display components, as shown in figure 4.2.

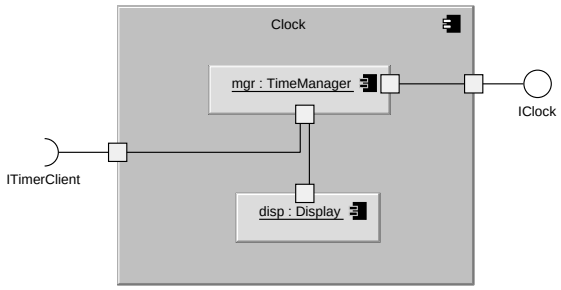


Figure 4.2: The composite Clock component

4.3.1 Detecting Bad Activity Errors

The protocol of the Display component (figure 4.3) accepts repeated calls to displayDigits, but does not allow these to be interleaved.

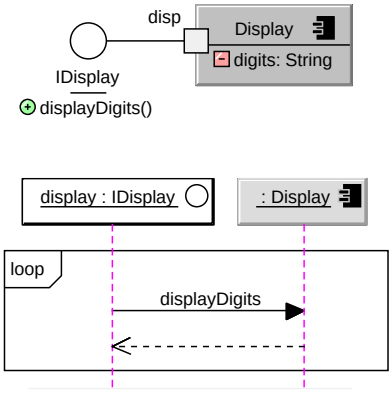


Figure 4.3: The Display component protocol

This can be expressed as:

```
Display = (c.disp.idisplay.call.rx.displayDigits ->
c.disp.idisplay.ret.tx.displayDigits -> Display).
```

However, there is a problem. This protocol will not detect errors in a client which can send in two displayDigits calls in succession, without waiting for the return. The protocol in figure 4.1 is capable of doing exactly that, and if this protocol is composed with the display protocol, then these erroneous transitions are omitted from the resultant LTS.

The solution is to augment the LTS with error transitions, for any input action (call.rx) which is handled elsewhere in the protocol but is not allowed from the current state. The augmented LTS for Display is shown in figure 4.4.

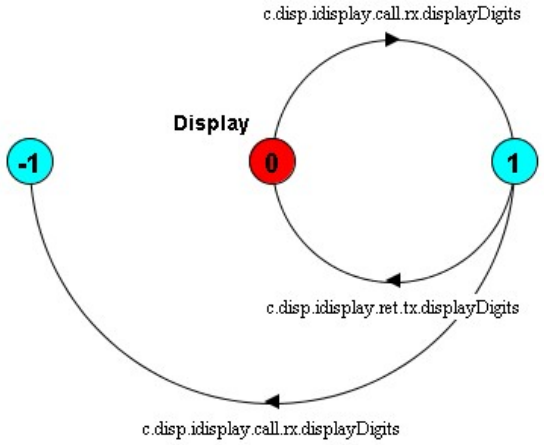


Figure 4.4: The augmented Display LTS

The FSP property construct can be used to construct error transitions, but it does not distinguish between input and output actions. This makes it unsuitable for this task. A way of restricting the alphabet used to construct error transitions for FSP properties would give the required effect, and it is envisaged that a facility similar to this will be added in the future to deal with this situation.

The LTS must be deterministic before adding error paths. To see why this is so, consider the the following non-deterministic FSP expression for a service:

```
Service = (a -> (b -> Service | c -> Service).
```

Translating this and adding error paths gives us the LTS in figure 4.5.

If the client FSP is

```
Client = (a -> b -> Client) + {c}.
```

and we compose both together, we get an LTS that reduces to the diagram shown in figure 4.6.

To correct this situation, we must only augment deterministic automatons. This leads to the assumption that if it is possible to compose the client and service together correctly, even in the presence of non-deterministic choice, then this is the correct action. This is not necessarily the case, as the actual implementation of the client may loop for 10 times calling a service, and the service may only accept the call 5 times. The protocols used are not powerful enough to express this distinction or catch this type of error.

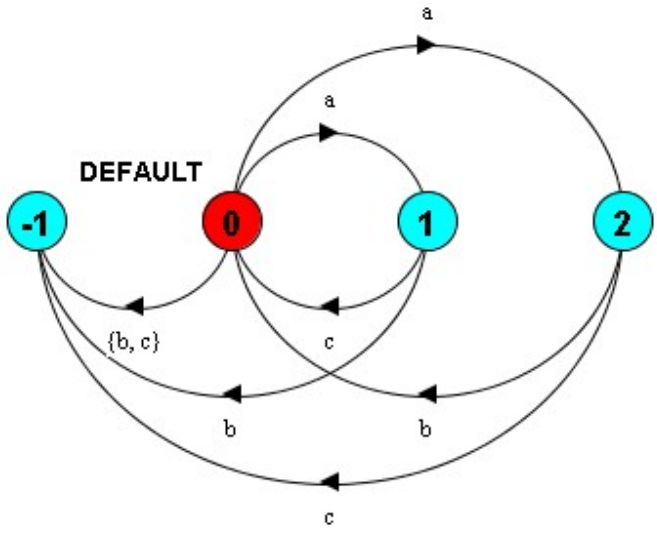


Figure 4.5: An augmented non-deterministic LTS

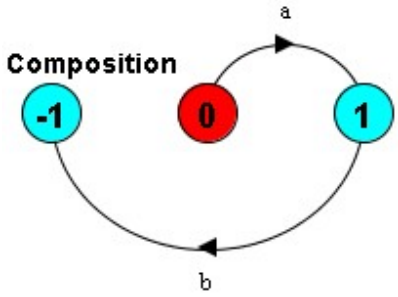


Figure 4.6: Composing a client and augmented non-deterministic service LTS

4.3.2 Composing the Constituent Protocols

To form the protocol of Clock, the protocols for the two parts are composed, and the internal events are hidden. We get the LTS in figure 4.7.

This reduces down to a protocol of a parallel operator with two operands: one for handling the changeTime calls, and one for handling notifyOfNewSecond calls. The LTS shown is slightly incorrect due to the lack of correct error transitions of the constituent protocols. The LTS currently accepts multiple call.rx.notifyOfNewSecond actions, which should be disallowed by an error transition.

By producing a sequence diagram for the composite, we can hide the complex internals and show back a convenient form to the developer. We can also use the generated sequence diagram as the protocol for this component when it forms an internal part of another composite component.

4.4 Protocol Handling of Indexed Ports

Indexed ports present a difficulty for protocol modelling. A request arriving at a component on input_port[5] may trigger a request being sent from output_port[10], for instance. Many other combinations are possible. This is difficult to model, because the relationships between the ports must be fully understood in order to look for protocol adherence.

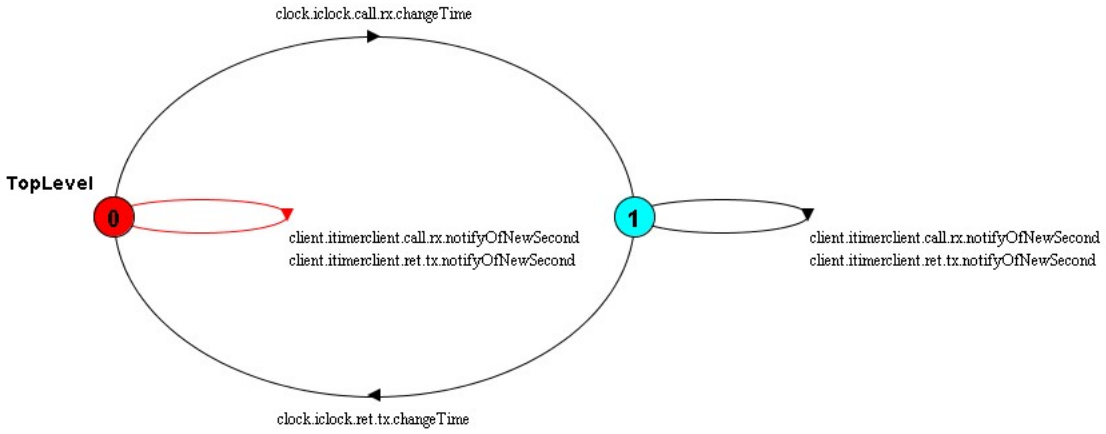


Figure 4.7: The Clock LTS

Rather than requiring the protocol to state the exact relationships between indexed ports, an analysis of the problems worked through so far show that primarily there are four relationships between indexed ports in a component:

1. Broadcast
   In this situation, the indexed ports represent clients who must all be notified of changes. We can model this effectively as a single port.
2. Mode
   In this situation, the component deals with a single index until it changes mode, whereupon it will deal with another index. This models the state pattern [19], and can be used to decompose a composite object into different constituent parts each representing a separate mode.
3. Per-call
   In this situation, each call from a client entails using a single index for the entire call.
4. One-to-one
   In this situation, a call received on an indexed port, can only communicate with other indexed ports with the same index.

Further analysis is required to determine if other modes exist, or if these modes are actually useful in practice. The current intention is for a component to indicate which one of the above situations apply, which will guide the protocol analysis.

4.5 Goal-Level Properties of Extensions

Some interference problems cannot be detected by protocols alone, and we intend to provide a way to express the goal of an extension as a set of properties that must be preserved after any transformation. An example where this occurs in one of the systems analysed so far is when an alarm extension and an hourly beep extension are added to the same clock architecture. In this case, although all the protocols are observed, it is possible for the alarm to be cancelled by a beep on the hour. The goal of the alarm is to sound for 20 minutes, and the other extension frustrates this aim.

We intend to allow any part of the architecture to specify a goal, possibly in terms of expected interactions that must be preserved. This is analogous to the invariants of [22] and may also take the form of additional FSP properties that must hold for the architecture to be considered to be correct.

Another possibility is the use of sequence diagrams to show the intended effects of an extension as a series of interactions between multiple components in the architecture. It would be necessary to show that the same effect was still preserved in a combined system. It is anticipated that this approach will lead to implied scenarios [60] which the specified interactions neither cover or rule out. These will require either validation or rejection by the developer.

Bibliography

[1] J. Adamek and F. Plasil. Partial bindings of components - any harm? In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04) - Volume 00*, pages 632–639. IEEE Computer Society, 2004.

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

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

[4] OSGi Alliance. About the osgi service platform. *Website*, http://www.osgi.org/documents/collateral/TechnicalWhitePaper2005osgi-sp-overview.pdf, 2005.

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

[6] Dorian Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, 2005.

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

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

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

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

[11] P.H. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, and A. van der Hoek. Differencing and merging within an evolving product line architecture. *Software Product-Family Engineering*, 3014:269–281, 2004.

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

[13] Mozilla Corporation. Firefox internet browser. *Website*, http://www.mozilla.com, July 2006.

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

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

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

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

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

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

[20] 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.

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

[22] J.D. Hay and J.M. Atlee. Composing features and resolving interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 110–119, New York, NY, USA, 2000. ACM Press.

[23] 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.

[24] D. Hou and J. Hoover. Towards specifying constraints for object-oriented frameworks. In *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, pages 5–, Toronto, Ontario, Canada, 2001. IBM Press.

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

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

[27] Object Technology International Inc. Eclipse plugin central. *Website*, http://www.eclipseplugincentral.com/, 2005.

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

[29] B.N. Jorgensen. Language support for incremental integration of independently developed components in java. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1316–1322, New York, NY, USA, 2004. ACM Press.

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

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

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

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

[34] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, San Francisco, California, United States, 1996. ACM Press.

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

[36] M. Mattsson and J. Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems*, pages 203–. IEEE Computer Society, 1997.

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

[38] N. Medvidovic. Adls and dynamic architecture changes. In *Joint proceedings of the second*

international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, pages 24–27, San Francisco, California, United States, 1996. ACM Press.

[39] N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 24–32, San Francisco, California, United States, 1996. ACM Press.

[40] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[56] M. Svahnberg and J. Bosch. Characterizing evolution in product line architectures. In *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*, 1999.

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

[58] 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.

[59] E. Truyen, B. Vanhaute, B.N. Jorgensen, W. Joosen, and P. Verbaeton. Dynamic and selective combination of extensions in component-based applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 233–242, Washington, DC, USA, 2001. IEEE Computer Society.

[60] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.

[61] 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.

[62] S. Yegge. When poymorphism fails. *Blog Entry*, http://www.cabochon.com/ stevey/blog-rants/polymorphism-fails.html, 2004.

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