Designing and Constructing Modifiable Middleware using Component Frameworks

Nikos Parlavantzas, Geoff Coulson

Computing Department,

Lancaster University, UK

[parlavan, geoff]@comp.lancs.ac.uk

Abstract

Because of the increasingly diverse and dynamic environments in which they must operate, modern middleware platforms need to explicitly support ***modifiability****. Modifiability should encompass change that is both static and dynamic; small scale and large scale. Also, the process of modification should be flexible, easy to perform, and consistency-preserving. To address these needs, this paper proposes a generic component-based modifiability approach, and then uses this approach to build a highly-modifiable middleware framework. The modifiability approach provides design support for building component frameworks—i.e., reusable and extensible component architectures that are targeted at specific domains. In the approach, component frameworks build upon a minimal, technology-independent component model, and can be recursively assembled into more complex frameworks. Our middleware framework—an instantiation of our proposed approach—takes the form of a specific assembly of component frameworks, each of which addresses a distinct middleware-related concern. Our middleware framework supports two styles of modification: First, '****architectural modification****' enables large-scale, static, changes, such as customizing the framework to a new application domain or underlying infrastructure. Second, '****system modification****' enables changes that are based on specific customisations of the framework; these changes are* *smaller in scope (e.g. replacing protocol implementations) but are applicable at both deploy-time and run-time. A prototype implementation demonstrates the feasibility of our approach and framework, and demonstrates a sufficient degree of supported modifiability.*

1. Introduction

Middleware platforms are well established as an essential element of large-scale distributed software systems. But increasing the applicability, and lengthening the lifespan, of platforms requires that they accommodate the vast diversity and fluidity that increasingly characterises their deployment environments. Environmental variations that need to be accommodated range from large-scale, slow-rate variations (e.g. supporting diverse application domains like real-time, mobile, or multimedia applications in varied deployment environments like desktop computers, PDAs, or mobile phones), to small-scale, fast-rate variations (e.g. applying patches, requesting different qualities in transferring continuous media, or adapting to dynamic fluctuations in resource availability).

To achieve such accommodation, middleware platforms must explicitly support *modifiability*. The modifiability requirement can be refined into three lower-level requirements: *flexibility*, *ease of modification*, and *consistency maintenance*. Flexibility relates to the range of possible changes that can be supported by a platform. Flexibility can in turn be refined into the requirements that the platform supports: i) static and dynamic modification; ii) extension with new functionality; and iii) large-scale modification (i.e., support for changes affecting large areas of functionality). Ease of modification relates to the effort of performing required changes; and consistency maintenance relates to the possibility that modifications may introduce inconsistency. As well as being modifiable[1], middleware platforms must also be *efficient*. This relates to the resource overhead induced by the platform—that is, any resource

[1] We use the term 'modifiable middleware' rather than 'adaptive middleware' because it is a more general term; in particular, it covers large-scale, static modification, not typically addressed by adaptive middleware.

usage that does not contribute directly to meeting the needs of middleware users. Unfortunately, efficiency typically conflicts with, and must be balanced against, modifiability.

No existing middleware platform satisfies all these requirements in a balanced way. Large-scale modification, in particular, remains largely unaddressed, thus making it difficult or impossible to customise platforms to different application domains and underlying infrastructures. Mainstream platforms, such as CORBA and Java EE, suffer from serious limitations in terms of flexibility [Kon02, Kordon05]. Modification in these platforms mainly consists in statically selecting from a fixed or minimally-extensible set of options. Research platforms, such as FlexiNet [Hayton99] and DynamicTAO [Kon00], as well as Microsoft .Net [Microsoft05] provide enhanced support for dynamic extension and modification, but lack any support for large-scale modification. Platforms that address this issue are OpenORB [Blair98], UIC [Roman01], and ExORB [Roman04]. OpenORB requires a large amount of effort for performing modifications, and provides weak support for consistency maintenance and efficiency. UIC allows large-scale changes, but provides no explicit mechanism to support them. ExORB defines a software construction approach that enables large-scale changes, but the approach relies on an uncommon programming model that imposes complexity on middleware developers.

To address the modifiability-related requirements in a balanced and principled way, this paper first proposes a generic, component-based, modifiability approach, and then uses this approach to build a highly-modifiable component-based middleware framework, called O_2. The modifiability approach provides concepts and rules for designing *component frameworks*; that is, domain-specific component architectures that enable a variety of run-time component configurations. The approach also includes a pattern for supporting dynamic modification in a uniform way across component frameworks. Component frameworks can be recursively assembled into more complex frameworks. The O_2 framework is itself constructed as an assembly of component frameworks, each addressing different concerns. O_2 inherently supports modifiability as follows: i) it addresses large-scale, static, changes through integrating new component frameworks and producing different *middleware architectures*; and ii) it supports smaller-scale changes, both static and dynamic, through managing the configuration of plug-in components. This work formalises and extends our previous work on OpenORB v2 [Coulson02a]. Specifically, it provides formal support for constructing and assembling component frameworks, and adds the capabilities to perform large-scale architectural modifications and to target multiple component technologies.

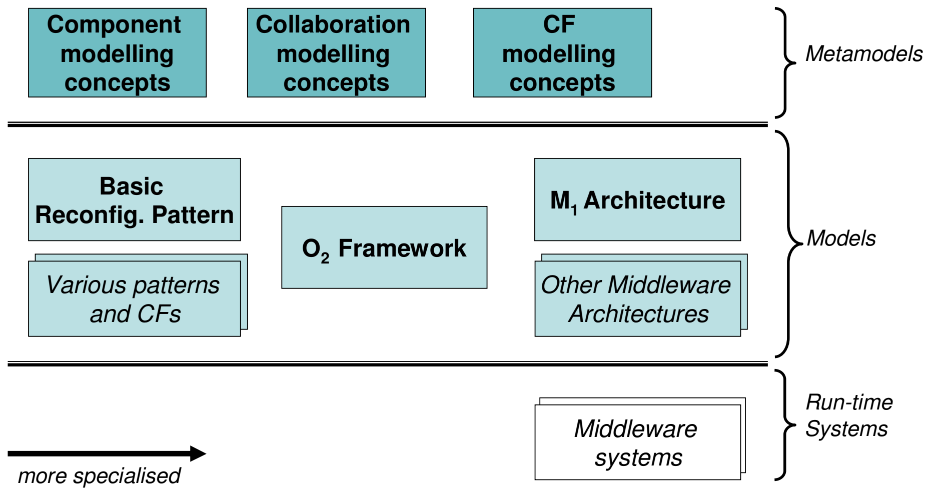

Figure 1 –The O_2 framework and other elements

Figure 1 depicts the logical relations between the main elements of our work. The vertical dimension organises the elements into the three traditional layers of metamodels, models, and run-time systems. The horizontal dimension organises the elements in order of increasing specialisation. The approach comprises the three categories of concepts in the metamodels layer (sections 2.1, 2.2, and 2.3) and the basic reconfiguration pattern in the models layer (section 2.4). The O_2 middleware framework builds on this pattern and a small number of basic frameworks (section 3). Middleware architectures build on O_2 and other frameworks. M_1

is a representative middleware architecture that forms the basis of our proof-of-concept implementation (section 4). We have performed on evaluation of O_2 with respect to modifiability and efficiency (section 5), and compared O_2 with related work (section 6).

2. A Component-based Approach to Modifiability

In outline, our proposed approach to the design of modifiable component systems is to build such systems in terms of component frameworks (hereafter, CFs) and to provide concepts for designing and expressing these CFs. The concepts that we propose for designing CFs fall into three categories:

- *component modelling* which defines fundamental concepts (e.g., the component concept) of the component model upon which CFs are built;
- *collaboration modelling* which defines concepts related to *collaborations*, the main constituents of CFs;
- *CF modelling* which, building on the above two sets of concepts, defines concepts specifically related to CFs themselves.

Apart from these concepts, we propose a *basic reconfiguration pattern* to facilitate the design of CFs that must support dynamic modification. We also specify how the various concepts are represented in terms of UML elements, thus providing a *concrete notation* for specifying CFs. This notation enables the use of existing UML tools to support the development and maintenance of CF models[2]. The three categories of concepts, along with their UML representations, are discussed in sections 2.1, 2.2, and 2.3 respectively. The reconfiguration pattern is then presented in section 2.4.

[2] ———Naturally, representing our concepts in terms of other modelling languages is also possible.

2.1 Component modelling concepts

To specify and build component systems, one needs a component model that defines what components are, how they communicate, and what services the supporting execution environment should offer. Our component model expresses the essential properties of component technologies that are compatible with our approach (one such technology is discussed in section 4).

In defining the component model, two main requirements were identified:

- The component model should be *minimal*. Specifically, it should only address the following core issues: i) component deployment and interoperation within a common address space; ii) component and interface naming; and iii) support for meta-information. The motivation here is to maximise the applicability of the component model to different types of systems (e.g., both application and infrastructure software), and to allow it to be used as a stable foundation for diverse, domain-specific models; that is, CFs.
- The component model should be *abstract* in the sense that it should hide technology-specific details such as binary interoperability standards or underlying virtual machines. The motivation here is to increase the ease of designing CFs and to enhance their reusability across different target technologies.

The main concepts of the component model (see Figure 2) are as follows: *Components* are units of implementation that are capable of being integrated into systems without modification. Components offer one or more distinct *interfaces*, that is, coherent sets of operations together with supporting specifications (e.g., pre- and post-conditions). Both interfaces and components are identified with globally unique identifiers. Components are represented at run-time by so-called *component objects* which are created as a result of component instantiation.

Components and component objects support *introspection*; that is, they can be probed for meta-information; this corresponds to the information captured in Figure 2 (e.g., information on the interfaces offered by a component). In addition, components can be associated with *extended metadata*; i.e., name-value pairs that form generic placeholders for additional self-descriptive information. The meaning of extended metadata is not prescribed by the component model; it depends on the needs of the specific context. For example, metadata could contain formal specifications of a component's behaviour, its contextual dependencies, its quality properties and requirements, requirements on infrastructure services, or licensing information. The metadata can be accessed through introspection both before and after the component is instantiated.

Components may be independently deployable or not. A component that is not independently deployable is termed a *subordinate* component. Each subordinate component is related to some other single independently deployable component, and can be instantiated only by this component, or by another of its subordinate components. Subordinate components are not separately replaceable. On the other hand, independently deployable components can be instantiated directly by any external party; they are separately replaceable units which contain all their subordinate components. The motivation for subordinate components is to simplify the programming models of component systems. Specifically, exporting a functionality unit as an independently deployable and replaceable component incurs a performance overhead, thus practically restricting the granularity of such components. Subordinate components enable exporting fine-grained units as components, thus providing for uniformity in programming models.

Finally, a supporting *execution environment* enables interoperation between component objects within a common address space and realises dynamic loading and unloading of components. It also provides facilities for component instantiation and dynamic interface discovery (i.e., obtaining a reference to an identified interface given any interface reference to a component object).

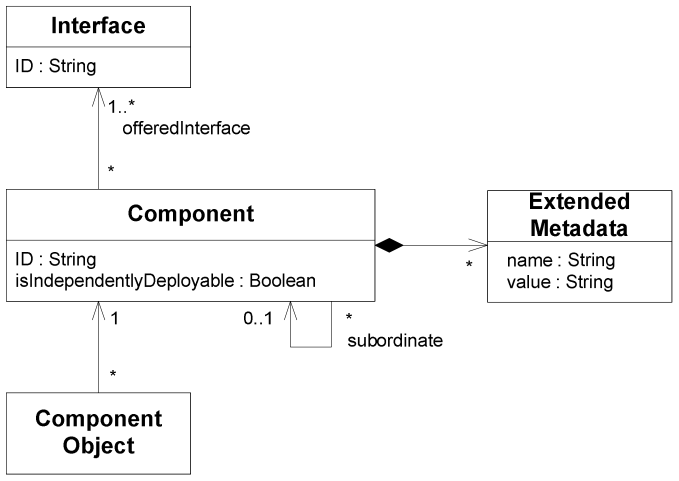

Figure 2 - Component modelling concepts

UML representation

In UML terms, components and interfaces are represented by the corresponding UML concepts. The supporting specification information for interfaces can be expressed in various notations depending on the required level of precision. For example, one might use OCL [OMG05b], UML interaction and state diagrams, or specialised quality of service (QoS) notations (e.g., that proposed in [Aagedal02]).

2.2 Collaboration modeling concepts

A collaboration describes an ensemble of interacting objects. The structure diagram in Figure 3 illustrates our collaboration modeling concepts and shows how they are related.

Before expanding on the collaboration concept, we discuss the central concepts of *component type* and *component relationship*. A component type is an abstract component description. It defines a set of interfaces that are *offered* by a component as well as a set of interfaces that are *used* by the component. The component type incorporates the specifications of its interfaces and possibly adds further information, such as constraints that relate interface specifications with each other. For example, such a constraint may prescribe that invoking a specific operation in some offered interface has the effect of invoking some operation in some used interface.

A component relationship is then a description of links between component objects of particular component types. Three frequently used component relationships are *usage*, *creation*, and *composition*, which respectively signify that a component object invokes, creates, or is composed of another object. In line with UML semantics, composition requires that a part object belongs to at most one composite at a time. Composition relationships simplify component system design because they define hierarchical structures of objects; they also play an important role in other aspects of our approach as will be seen in sections 2.3 and 2.4.

Building on the above definitions, a *collaboration* is a description of how a collection of component objects cooperate to achieve a joint goal. Collaborations comprise both structural and behavioural information. First, the structure of a collaboration comprises a set of component types and a set of relationships between them. Types and relationships defined within a collaboration are termed *roles* and *connectors* respectively: roles specify the properties that objects must exhibit to be able to participate in the collaboration; and connectors specify the properties of the links between participating objects. A collaboration definition may also contain multiplicity constraints (e.g., only a single object can play a given role) and specialisation constraints (e.g., some roles are specialisations of a given component type). Second, the behaviour of a collaboration contains a set of *interactions*, which define the exchange of messages between different roles over connectors.

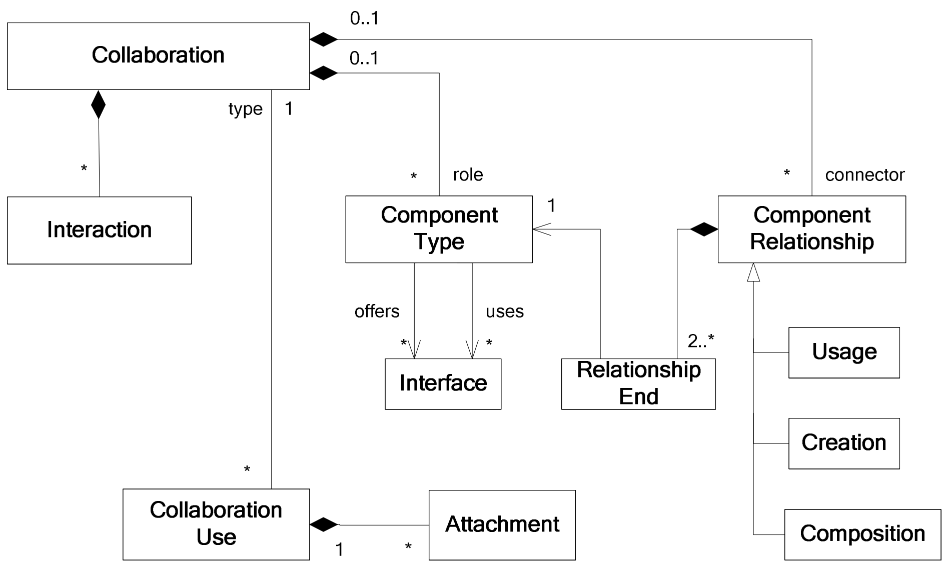

Figure 3 – Collaboration modelling concepts

Crucially, existing collaborations can be employed to build new collaborations through the *specialisation* and *aggregation* mechanisms. A collaboration *specialises* a base collaboration if its definition draws on and refines that of the base collaboration. Thus, a specialised collaboration contains all the roles and connectors of the base collaboration, or specialisations of them; and may optionally contain additional roles and connectors. A collaboration *aggregates* another collaboration if the definition of the former (the aggregate collaboration) contains a *use* of the latter (the aggregated collaboration). Specifically, a *collaboration use* represents a particular application of the aggregated collaboration to the aggregate collaboration by defining a set of *attachments* between roles/connectors of the two collaborations. These attachments indicate which elements of the aggregate collaboration correspond to and refine which elements of the aggregated collaboration—a single element of the former may correspond to multiple elements of the latter. Note that collaboration specialisation can be seen as a special case of aggregation, in which the specialised collaboration both contains a use of the base collaboration and refines the base collaboration. The next subsection contains an example that illustrates all of these concepts.

UML representation

Collaborations, collaboration uses, and interactions are represented as the corresponding UML concepts. As regards these concepts, our metamodel (see Figure 3) follows the general outline of the UML metamodel, but is significantly simplified for our purposes. Component types and relationships are respectively represented as UML classifiers and associations. Usage relationships are represented as «uses» associations, composition relationships are represented as UML compositions (graphically shown using the filled diamond or the nesting notation), and creation relationships are directly represented as «creates» associations. Similarly to interfaces, component types may be annotated with supporting specification information that is expressed in OCL, state diagrams, or any other convenient notation.

Example

To illustrate the notation for collaborations, consider a 'lookup' collaboration between client, server, service, and registry roles with the goal of enabling dynamic service discovery (see Figure 4). In this collaboration, clients use the registry to locate a service that they need, and servers use the registry to register services. In addition, consider a 'simple printing' collaboration involving user and printer roles with the goal of carrying out a printing task. As seen in Figure 4, lookup and simple printing can be combined to define a 'printing' collaboration that supports the dynamic discovery and use of printers. Specifically, 'printing' specialises 'simple printing' and aggregates 'lookup'. The collaboration use notation shows the attachments between roles in 'lookup' and 'printing', which also determines the attachments of connectors. Naturally, interactions between roles in 'lookup' apply to their corresponding roles in 'printing'. Note that the 'printer service' role is attached to both the 'server' and 'service' roles, which indicates that in the 'printing' collaboration, printers are assumed to register themselves with the registry.

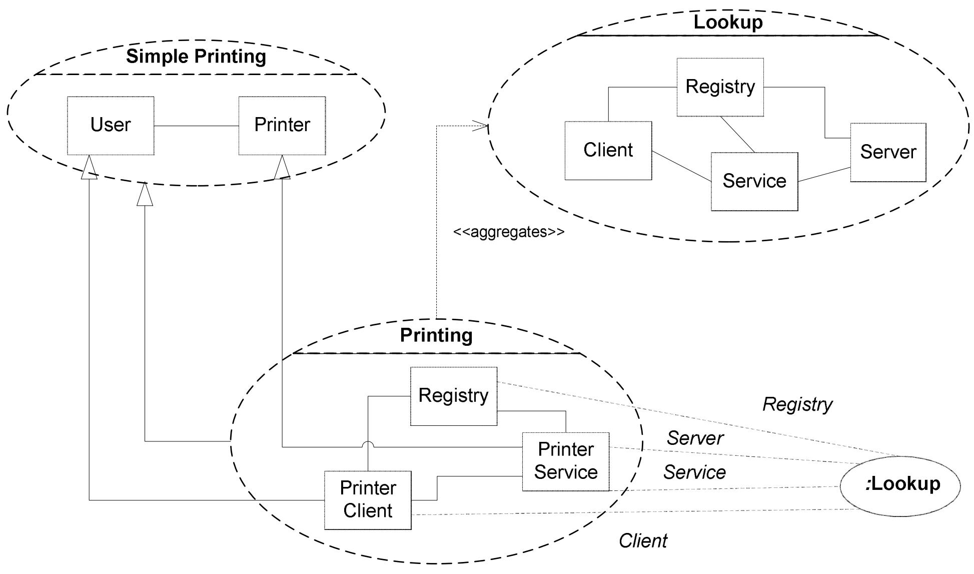

Figure 4 – Example of collaboration specialisation and aggregation

2.3 CF modelling concepts

CFs are reusable architectures for specific domains, and are designed to be instantiated in terms of components. Specifically, a CF defines a set of design rules that constrain the external characteristics of components, their relationships, and the interactions among them. Instantiating a CF involves implementing and integrating components according to the prescribed rules. In addition, a CF may include software that supports or enforces the rules. The main motivation for CFs—as, indeed, for all kinds of frameworks—is design and code reuse. Moreover, since CFs are architectures, they provide a means of ensuring that systems maintain desired architectural properties. One property that is supported to some degree by all CFs is modifiability; CFs enable modifiability by definition since they can be instantiated in multiple ways, forming multiple run-time component configurations.

Formally, a CF is a grouping of collaborations, components, and supporting documentation (see Figure 5). A CF minimally contains a *primary collaboration* which may be defined as a specialisation or aggregation of other collaborations. The structure and behaviour of its collaborations express the design rules of the CF. As well as collaborations, a CF may contain a set of components that realise roles defined in its collaborations. These components represent *CF-provided software*, which implements frequently-used or always-required functionality, such as functionality to support and/or enforce CF rules. Finally, the CF contains supporting documentation that is necessary for understanding, using and evolving the CF. This typically comprises the following: a discussion of the problem domain and the goals that the CF addresses; conceptual models at high abstraction level; a framework overview; design constraints and rationale; and examples of using the CF.

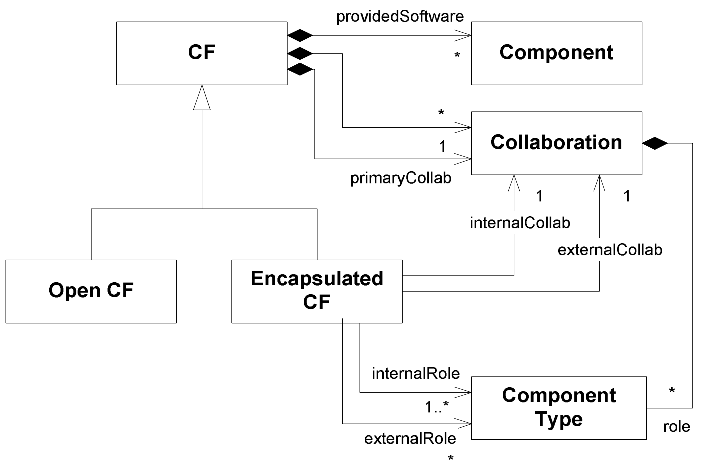

Figure 5 - CF modelling concepts

As with collaborations, CFs can be assembled into more complex CFs through specialisation and aggregation. A CF specialises or aggregates another CF if their respective primary collaborations are related by specialisation or aggregation respectively. Assembling CFs is a key means of managing the complexity of designing, understanding, and evolving component architectures.

Clearly, CF specifications that are expressed using the abstract component modelling concepts discussed above are not, in themselves, sufficient to support the implementation of executable component systems. To provide this capability, CFs must be refined with details that are specific to a target component technology that supports software execution. This refinement is typically supported by a mapping from the abstract component model to a concrete component technology. Using such a mapping, one can straightforwardly concretise a technology-independent CF to a technology-specific CF. Of course, the transformation will generally require additional input, such as engineering decisions associated with the particular technology.

Encapsulated versus open CFs

As seen in Figure 5, we distinguish two types of CFs, namely, *encapsulated CFs* and *open CFs*. Encapsulated CFs integrate component objects to form encapsulated systems; that is, groups of objects that are treated as a single behavioural unit at a higher abstraction level.

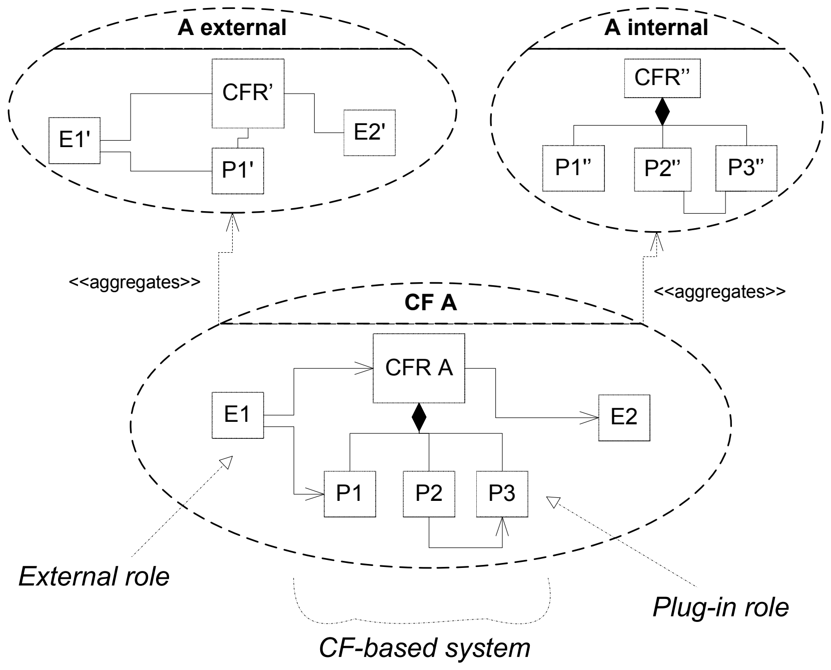

Figure 6 - Example of encapsulated CF

An encapsulated CF describes both how its constituent component objects cooperate to realise the target system (termed the *CF-based system*) and how this system interacts with its environment. The roles of an encapsulated CF are therefore classified as either *external* or *internal* roles depending on whether or not the conforming objects are part of the CF-based system. The system is itself represented by a distinguished internal role, called *CFR* (for CF-based system representative), which is transitively composed of (i.e., has a composition relationship to) all the other internal roles, which are termed *plug-in* roles. In analogy to the above-mentioned external/internal role separation, the primary CF collaboration is decomposed into external and internal collaborations; that is, the primary collaboration is defined as an aggregation of these two collaborations. Figure 6 shows an example of an encapsulated CF (CF A) with two contained collaborations (A external and A internal).

Encapsulated CFs are useful for modelling self-contained, modifiable component systems as well as their subsystems; that is, traditional units of functional decomposition. Aggregating encapsulated CFs is facilitated by their separation into two collaborations (external and internal). This is because the designer of an aggregate CF does not need to know about the internal collaborations of the aggregate CF's constituent CFs. Importantly, encapsulated CFs are essential elements of our reconfiguration approach, which is discussed in section 2.4.

We now turn to *open* CFs. Open CFs constrain object integration for various purposes *without* explicitly defining an encapsulation boundary. They are useful for capturing component integration protocols. For example, open CFs can capture interactions with shared services, such as logging, security, or inter-address space communication. Moreover, open CFs can capture commonly used interaction styles (e.g., variations of the observer pattern) or domain-specific composability standards (e.g., data exchange standards in process control applications). Naturally, open CFs may be aggregated by an encapsulated CF in order to describe different aspects of the CF-based system. An example of an open CF—O_2's *resource CF*—is given in section 3.2.

UML representation

A CF is represented as a stereotype of a UML package that collects UML representations of its contained collaborations and components. The primary collaboration is given the same name as the CF, and its graphic notation (i.e., the dashed ellipse) is commonly used to represent the whole CF. The CF's supporting documentation is represented as a collection of UML artifacts owned by the CF package and is expressed in a combination of formal and informal notations.

2.4 The basic reconfiguration pattern

CFs allow developers to select independently-deployable components that will populate the CF instantiation at run-time, and thus they inherently support static modification of component systems. However, CFs do not inherently support dynamic modification. To address this, we define a *basic reconfiguration pattern*, which provides a simple and uniform means of supporting both static and dynamic modification. Modification is achieved through changing the run-time configuration of CF-based systems comprising component objects and the links between them.

The basic reconfiguration pattern is modelled as a simple, pre-defined, encapsulated CF (i.e., it itself is an instance of the CF concept) with the following roles: *reconfiguration manager*, *managed part*, and *configurator*. The reconfiguration manager is composed of the managed parts and is responsible for establishing their initial configuration and for maintaining and managing their dynamic reconfiguration. This responsibility is reflected in offered reconfiguration services, which are used by the configurator. The manager accomplishes its responsibility by interacting with the managed parts. This may involve creating or deleting managed parts or using management services provided by them. The basic reconfiguration pattern is specialised by encapsulated CFs that need to support modification, as seen in Figure 7. Specifically, the encapsulated CF's CFR role specialises the reconfiguration manager role; an external role specialises the configurator role; and plug-in roles specialise the managed part role.

The basic reconfiguration pattern is intentionally defined at a high abstraction level; it neither restricts the reconfiguration services exposed by the CFR, nor does it prescribe how these services are realised by cooperating with plug-ins. CF designers make these decisions by considering actual modifiability requirements and by exploiting domain-specific knowledge and built-in constraints associated with the CF. For example, in a multimedia streaming CF, a designer may exploit a constraint that plug-ins are to be arranged in a pipe-and-filter architectural style, in order to include a service for inserting a filter between two other filters without interrupting the data flow. As another example, the designer may exploit a constraint that there must be a single object of a particular plug-in type in order to include a service for dynamically replacing the object, but not for removing or adding objects of this type.

Despite considerable variability in the detail of reconfiguration management across different CFs, the basic reconfiguration pattern does define a small set of *generic interfaces* (see [Parlavantzas05] for the full detail), which must be offered along with a reconfigurable CF's CF-specific interfaces (see Figure 7). Specifically, the reconfiguration manager (i.e., the CFR) must offer an IPartConfiguration interface, which exposes the configuration of parts as a modifiable collection of named objects. This interface has operations to retrieve, add, remove, and replace managed parts; one can add both existing objects and new objects that will be instantiated by the manager. The interface also has operations to subscribe and unsubscribe to events that report changes in the configuration; these events are realised as invocations on an associated IPartEvents interface. Managed parts (i.e., plug-ins) must offer another interface called IManagedPart which has operations for initialisation and termination that are called when a part is added or removed.

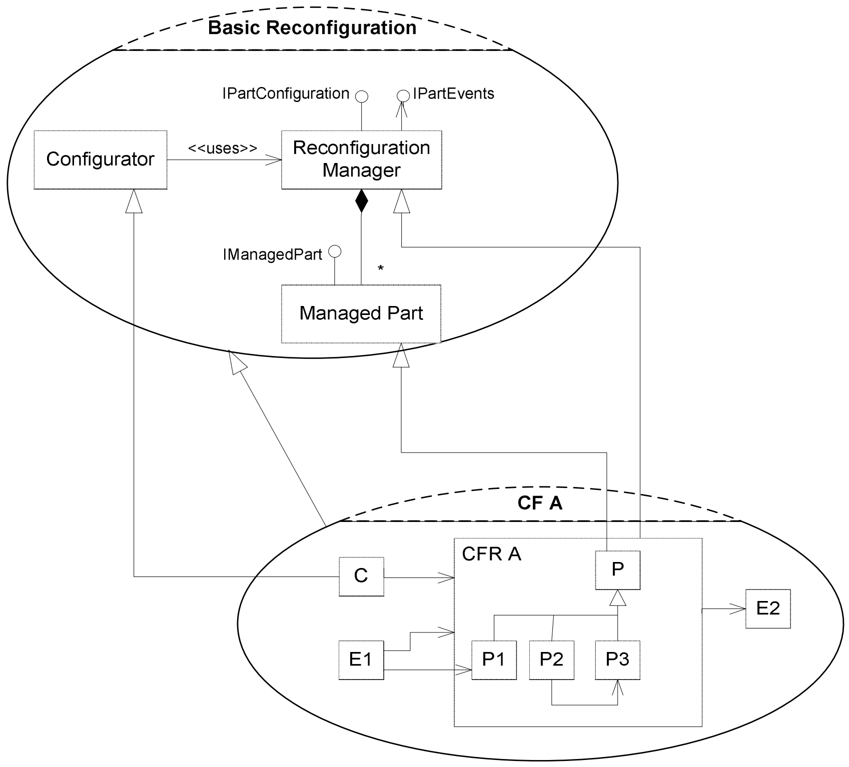

Figure 7 – The basic reconfiguration pattern and an example of its application to a CF

The behavioural specifications of these generic interfaces are abstract and must be refined within specialised CFs. For instance, a CF will typically constrain the types of objects that are eligible to be added and the conditions under which part removal or replacement is permitted. Moreover, for consistency reasons, the syntactic form of the generic interfaces must be followed by CF-specific interfaces when similar functionality is exposed (e.g., part addition).

Three main benefits accrue from the use of the basic reconfiguration pattern:

- The pattern supports CF designers by providing a general model of reconfiguration management that is applicable to all encapsulated CFs.
- The pattern allows designers to exploit CF-specific knowledge in order to achieve a desired level of modifiability. For example, designers can provide high-level reconfiguration services that rely on domain-specific abstractions, thus facilitating

modification. Similarly, designers can provide reconfiguration services that validate changes using CF rules and invariants, thus preventing inconsistencies.

- Providing generic reconfiguration-related interfaces reduces the complexity imposed on developers due to the uniformity in syntax and semantics. Moreover, generic interfaces promote composability with components that have no built-in knowledge of the CF, such as automated management tools.

The main liability of the basic reconfiguration pattern is the resource overhead for realising reconfiguration services. However, apart from the overhead associated with supporting the simple generic interfaces, this is CF-specific and can be appropriately traded off against other modifiability-related requirements, such as the desired flexibility level.

Finally, note that our modifiability approach can naturally be extended with more prescriptive, domain-specific, forms of the basic reconfiguration pattern (i.e., specialisations of the basic reconfiguration CF). These can provide enhanced support to designers at the cost of reduced generality. For example, one could define a specialisation that assumes that parts interact with each other only through connections initiated and controlled by the manager using specific operations on parts. The reconfiguration interface would then treat the part configuration as a graph and provide, for example, operations for connecting and disconnecting parts.

3. The O_2 framework

Having discussed the basic concepts of our CF-based modifiability approach, we now present a concrete instantiation of our approach: the O_2 middleware framework. O_2 is designed as an encapsulated CF that combines multiple simpler CFs, each of which addresses different sets of middleware-related concerns, and supports multiple run-time component configurations. However, a fixed CF cannot accommodate the diverse and ever-changing requirements imposed by all possible middleware environments. For this reason, O_2 addresses only a basic core of general requirements, and it is explicitly designed to support specialisation, which opens up two styles of modification (see Figure 8):

- *Architectural modification* involves extending O_2 by integrating new CFs and adapting the result to a specific component technology. The product is an aggregate, technology-specific CF, which we call a *middleware architecture*. This modification style is intended to enable large-scale and static changes, such as changes to the API exposed to middleware users or modifiers. Different middleware architectures will typically be designed for different application domains and underlying infrastructures—e.g. multimedia applications on desktop computers, or mobile applications on PDAs.
- *System modification* then involves providing components that plug into a particular middleware architecture's CF(s), and managing their configuration through CF-provided facilities; these facilities minimally include reconfiguration support following the basic reconfiguration pattern. System modification enables smaller-scale changes to middleware systems both statically and dynamically. For example, it enables replacing resource management policy components at either deployment-time or run-time.

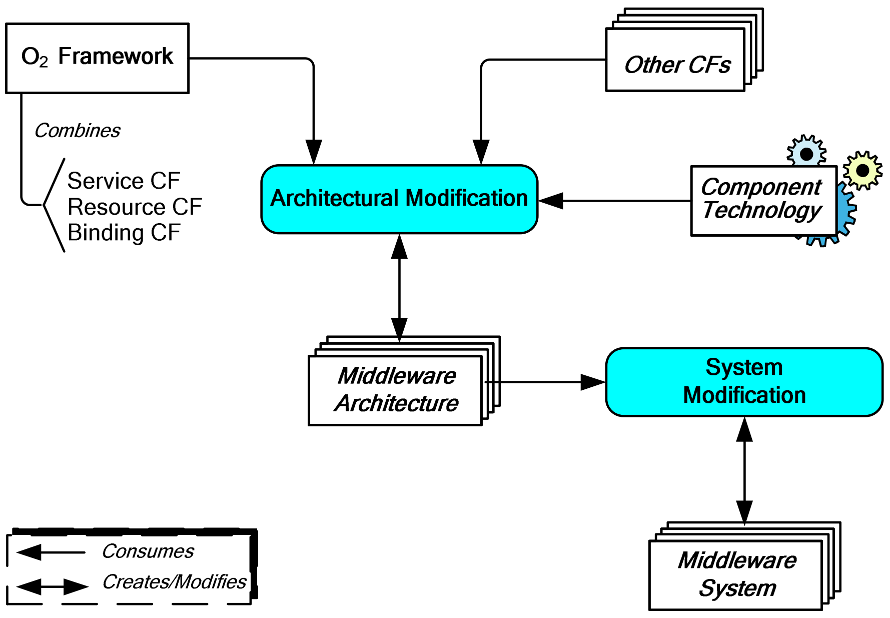

Figure 8 – Modification styles supported by O_2

The rest of this section is structured as follows. First, section 3.1 presents an overview of the structure of O_2 as an assembly of three basic CFs. Section 3.2 then discusses each of these basic CFs, and section 3.3 describes how they are integrated.

3.1 O_2 overview

O_2 is an aggregate CF that combines three constituent CFs: the *service CF*, the *resource CF*, and the *binding CF*. This combination of CFs is intended to address three common general areas of requirement in the design of middleware systems: i) requirements for supporting modifiability; ii) requirements for managing underlying resources; and iii) requirements for interconnecting application components (or, establishing *bindings* between them). More specialised requirements are satisfied at the level of particular middleware architectures, which may add further CFs. For example, a requirement to support modification of communication protocol functionality can be addressed by defining a middleware architecture that integrates a CF for composing protocol stacks.

In more detail, O_2 is defined as a specialisation of the service CF that aggregates the resource and binding CFs. The three basic CFs, a subset of their internal structure, and their relationship with O_2 are shown in Figure 9.

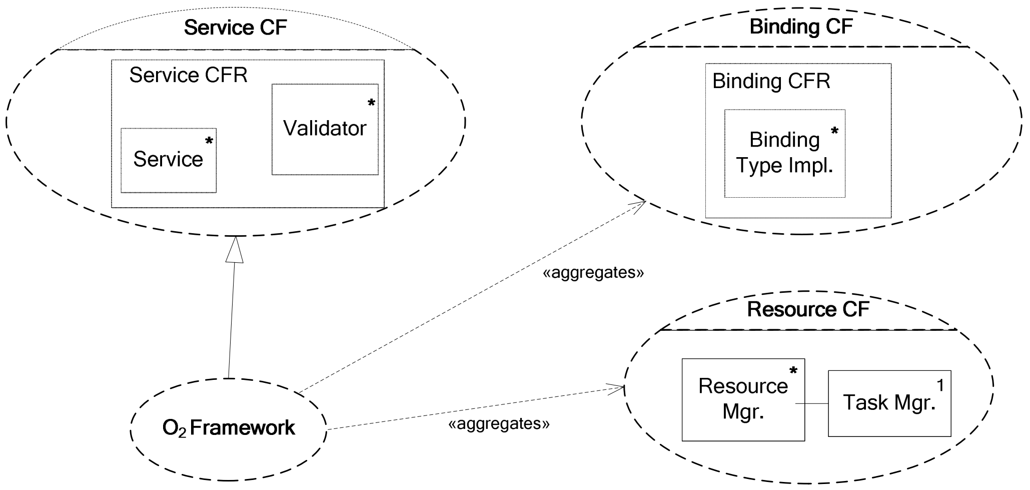

Figure 9 – The basic CFs and part of their structure

3.2 The basic CFs

- **Service CF**

The service CF is a lightweight, minimally-prescriptive CF that provides infrastructural support for assembling interdependent components. The CF relies on the abstraction of a *service*, a uniquely-identified and shared unit of functionality. Specifically, it is an encapsulated CF that accepts *service* plug-ins that realise one or more services and submit requests for other services at any point during their execution. The *service CFR* resolves such dynamic requests by consulting the current plug-in configuration and dynamically loading appropriate plug-ins, if necessary. Dynamically removing and replacing service plug-ins is also supported following the basic reconfiguration pattern. To facilitate robust reconfiguration, the CF employs a simple notification mechanism whereby registered service users are notified when a service is to be removed or replaced. The service CF also supports plug-ins called *validators*. These decide whether proposed changes to the service configuration are to be allowed. Their decisions are based on the current configuration, and on extended metadata obtained from to-be-added plug-ins, which capture the latters' resource and service requirements.

The service CF plays two key roles within O_2. First, it forms a basis for the specification of the basic structure of middleware architectures. This relies on the capability of the service CF to be easily specialised by adding constraints on the set of services that must be available. Second, it facilitates large-scale, static and dynamic modification of middleware systems by enabling changes in the service configuration.

- **Resource CF**

The resource CF [Parlavantzas03b] facilitates resource adaptation: that is, monitoring and controlling the resource usage of activities within a running middleware system. The resource CF is an open CF that relies on the abstractions of *tasks* and *resources*. Tasks are units of computation to which resources are allocated and resource usage is charged. Tasks are organised into a dynamic hierarchy that serves to delineate computations that contribute to the same goal. Each resource is associated with a single task, but the association is adjustable. For example, a thread (a resource) can be transferred to a different task to reflect the fact that it performs work contributing to a different goal.

Resources of a given type are managed by *resource managers*, and tasks are managed by the *task manager*. Resource managers enforce task-based resource allocation, accounting, and control, and the task manager acts as the access point for obtaining resource allocation information. By assigning most of the responsibilities to resource managers, this design can accommodate diverse types of resources at different levels of abstraction (e.g., buffers, threads, virtual processors). Importantly, the CF defines low-level adaptation primitives that expose a high degree of flexibility to users. The primitives include support for inspecting the resources allocated to a task, obtaining/ modifying the properties of individual resources, navigating the task hierarchy, transferring resources between tasks, and terminating tasks. The role of the CF within O_2 is to support adaptation with respect to the resources provided by the underlying infrastructure. Resource adaptation is essential for managing QoS requirements relating, for instance, to response time or throughput. As an example, a middleware system might adapt the CPU time allocated to processing requests in order to sustain response times in the face of fluctuating network delays. Resource adaptation is also useful in many other situations, such as ringfencing the resource usage of independently-developed and dynamically-plugged components.

- **Binding CF**

The binding CF [Parlavantzas03a] supports the development and integration of so-called *binding types*, and exposes a simple and consistent programming model for using binding types. Binding types (BTs) are middleware-provided interaction paradigms, such as remote method invocation, messaging queuing, or group communication, which are modelled as open CFs. For instance, a publish/subscribe BT defines publisher, subscriber, and event channel roles and specifies that publisher invocations on the channel are propagated to subsribers with specific delivery semantics (e.g., at-most-once). The binding CF is an encapsulated CF that accepts *binding type implementation* plug-ins. Its external collaboration captures commonalities across BTs and constrains their form, thus providing both guidance for BT designers and consistency for binding users. The external collaboration contains, for example, a standard binding establishment collaboration, which constrains the process of binding publishers and subscribers to event channels. The internal collaboration of the CF provides mechanisms to support implementing BTs, such as mechanisms to resolve dependencies on other BTs and low-level services. It also supports managing the plug-in configuration following the basic reconfiguration pattern.

The binding CF can accommodate a wide variety of BTs since it is based on a general binding model, which allows distributed or local, multiple-participant, explicit or implicit, and first- or third-party bindings. Moreover, the CF enables dynamic inspection and adaptation of bindings since bindings are created explicitly and represented as objects. The role of the binding CF within O_2 is to support extension with respect to supported interaction paradigms, thus increasing the applicability and value of the middleware framework. By accommodating high-level BTs that closely match application needs (e.g., auction or voting BTs), the CF also contributes to increasing the usability of O_2.

Full details on the basic CFs are given in [Parlavantzas05].

3.3 Integration of the basic CFs

Having described the three basic CFs, we are now in a position to present the full picture: *O_2 is a specialisation of the service CF that adds a number of roles and constraints that are associated with the resource and binding CFs*. Specifically, O_2 defines the following added roles: binding service, task service, resource service, and communication service (see Figure 10). The first three roles are defined as services that respectively refine the CFR role of the binding CF, and the task manager and resource manager roles of the resource CF. The communication service role captures services that can be used by the binding service; communication services are optional. Note that O_2's service roles are separated into three layers based on an “allowed to use” relationship: a role in a given layer is allowed to use roles in any lower layer. The resource layer contains the task and resource services; the communication layer contains communication services; and the binding layer contains the binding service.

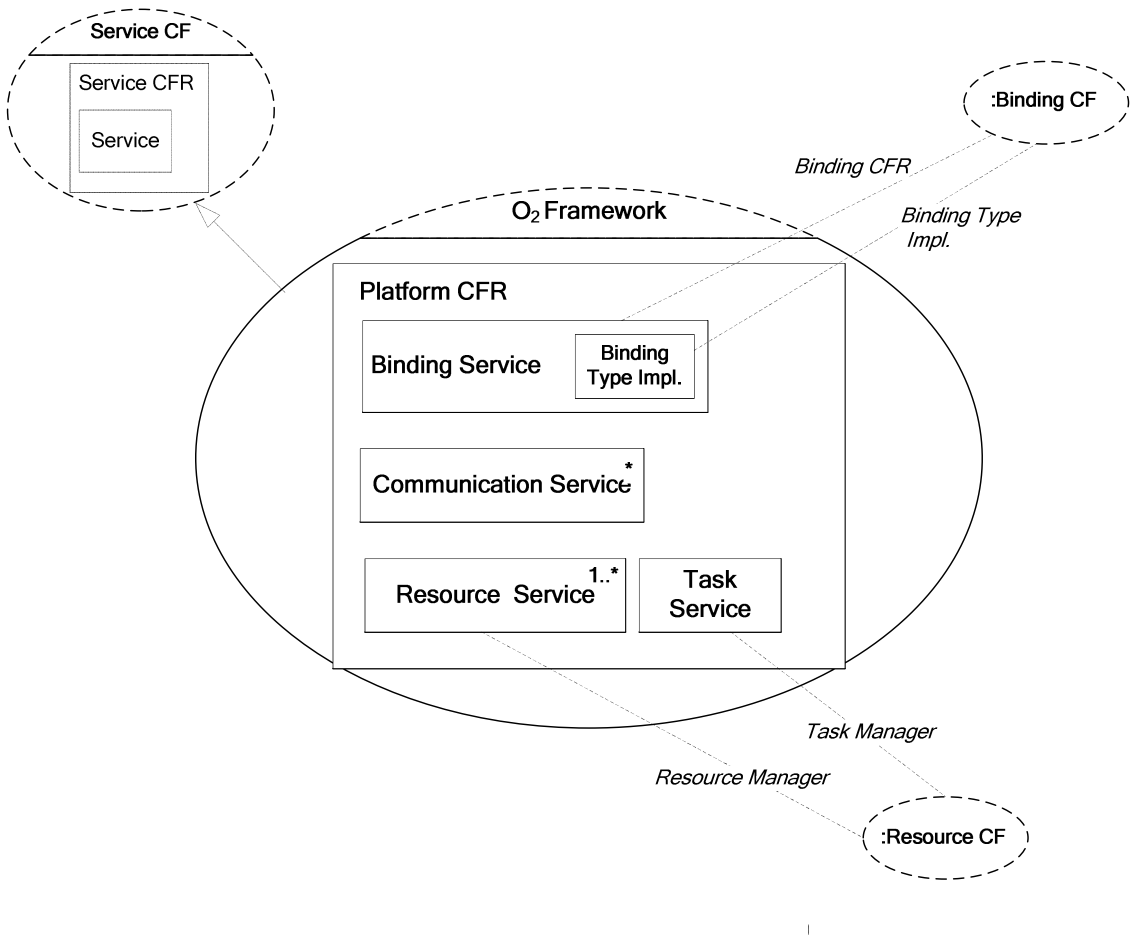

Figure 10 - Structure of O_2

Finally, O_2 defines the following two constraints on its roles. First, all services must use resource services provided by the resource CF rather than raw resources provided by the underlying infrastructure. For example, if there is a resource service for managing threads, this must be used instead of native OS or pthread calls. Conformance to this constraint is necessary for taking advantage of the resource adaptation facility provided by the resource CF. Second, the O_2 CFR must create and maintain a unique task for each service. The motivation for this constraint is to isolate service implementations and to enable accurate tracking of their resource usage.

4. Using O_2

O_2 is used to design middleware architectures, that is, aggregate, technology-specific, CFs which are published to application and middleware developers, and which form a blueprint for using, developing and modifying middleware systems. Producing a middleware architecture using O_2 involves two activities, not necessarily performed in sequence. First, it involves specialising O_2 by adding service roles residing at the resource layer, and, optionally, at the communication and binding layers. Such added roles are typically associated with new CFs, which are thus integrated into the middleware architecture. Second, it involves refining the middleware architecture's CFs with details specific to a target component technology. As mentioned in the previous section, this is typically guided by a mapping between the abstract component model and the concrete component model.

To demonstrate the use of O_2, we outline a representative middleware architecture, called M_1[3] which forms the basis of our current proof-of-concept implementation. The structure of M_1 is depicted in Figure 11. M_1 specifies the following resource services: *memory service*, *thread service*, and *transport service*, which are attached to the CFR roles of corresponding encapsulated CFs that (respectively) accept as plug-ins memory allocation policies, thread schedulers and transport protocols. Moreover, M_1 specifies the following communication services: *protocol service* and *multimedia streaming service*, which are attached to factory roles that create protocol stacks and media filter graphs respectively, and belong to corresponding encapsulated CFs that accept protocol and media filter plug-ins respectively. Finally, M_1 specifies a simple validator (*fixed resource validator*) that simply forbids the dynamic removal of the resource services. The design of the CFs introduced by M_1—that is, the thread management, memory management, transport management, protocol, and multimedia streaming CFs—draws significantly on previous work, mainly the GOPI platform [Coulson02b].

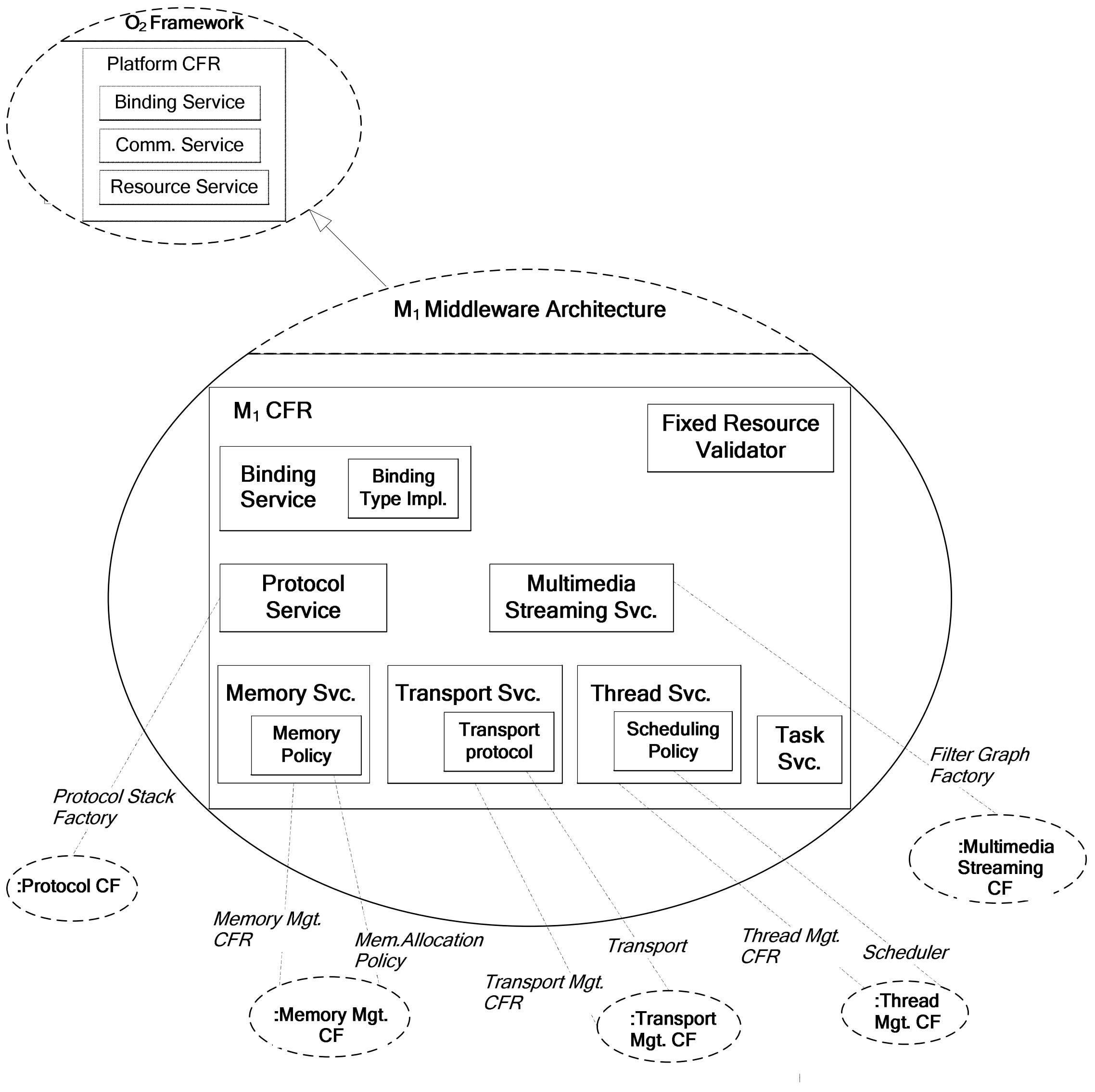

Figure 11 - Structure of the M_1 middleware architecture

The M_1 middleware architecture builds on *OpenCOM* [Clarke01], [Coulson02a], which is a concrete component-based programming technology, the concepts of which correspond closely to the abstract component model discussed in section 2.1. OpenCOM is based on Microsoft's COM [Microsoft04] and inherits COM's main benefits; namely, language independence and efficient in-memory interoperation between components. However, OpenCOM relies only on the core elements of COM (mainly, the binary-level interoperability standard, and the special *IUnknown* interface for dynamic interface discovery) and explicitly excludes the remaining, higher-level COM/COM+ elements, such as inter-process communication and transaction handling. OpenCOM adds a set of reflective facilities to this core subset of COM—notably, introspection support—which enable it to be applied in building CFs according to the proposed modifiability approach.

To use OpenCOM as a concrete technology for building CFs, one needs a mapping from elements of our minimal, abstract component model to OpenCOM elements. This mapping enables one to transform technology-independent CFs to OpenCOM-specific CFs, and it is is particularly straightforward. Component and interface identifiers are mapped to OpenCOM's 128-bit globally unique identifiers (referred to in COM parlance as CLSIDs and IIDs respectively); and independently deployable components and subordinate components are mapped to creatable and non-creatable OpenCOM classes respectively. The facilities for component instantiation and introspection are mapped to corresponding OpenCOM facilities. The facility for dynamic interface discovery is mapped to *IUnknown* operations. Note that OpenCOM includes a reconfiguration facility in which the runtime environment maintains and manipulates connections between objects. This facility is unnecessary for realising the abstract component model and is not used in M_1, which applies CF-managed reconfiguration. The simplicity of the abstract component model means that it can be easily mapped to a wide range of component technologies beyond OpenCOM. For example, in the case of .Net ([Microsoft05]), an independently deployable component is mapped to an assembly containing a special, annotated class whose instances represent the component instances. Subordinate components are mapped to other classes contained in an assembly. Component instantiation and introspection are realised using the .Net reflection service.

The proof-of-concept M_1 implementation comprises a set of components that realise M_1 roles and populate its multiple run-time configurations. The implementation consists of approximately forty OpenCOM creatable components that collectively comprise about 60,000 lines of C++. Implemented components include the following:

- components that implement BTs, such as remote method invocation, publish/subscribe, group communication, message queuing, and an 'e-auction' BT;
- protocol components, such as components that fragment and reassemble messages, components that implement reliable and unreliable multicast protocols, and an implementation of CORBA GIOP;
- scheduler components that realise priority-based and earliest deadline first thread scheduling policies;
- memory allocation policy components that implement first-fit, best-fit, and binary buddy allocation schemes; and
- transport components that support TCP, UDP, and IP multicast.

The wide range of implemented components demonstrates that the M_1–based middleware system can be extended along multiple dimensions with multiple, commonly used variants of middleware functionality.

5. Evaluation

This section presents an evaluation of O_2 with respect to the requirements discussed in section 1: namely, *flexibility*, *ease of modification*, *consistency maintenance*, and *efficiency*.

5.1 Flexibility

Flexibility is evaluated in terms of the three constraints identified earlier, which consider support for: i) static (i.e., at design, implementation, and deployment time) and dynamic (i.e., at operating-time) modification; ii) extension; and iii) large-scale modification.

O_2 supports *static modification* by supporting the design of different middleware architectures as O_2 specialisations. The flexibility available in designing architectures is particularly high since O_2 imposes only minimal constraints on the resource, communication, and binding-layer services (see section 3.3). In fact, apart from the constraints related to resource management, O_2 imposes no constraints on the primary functionality of these services, thus allowing the creation of a wide range of middleware architectures.

O_2 supports *static* as well as *dynamic modification* through the application of the basic reconfiguration pattern by all encapsulated CFs. The actual degree of flexibility exposed by each pattern application depends on the specific CF. For example, the multimedia streaming CF exposes operations to configure a filter graph, i.e., operations for adding, removing, replacing, connecting, and disconnecting filter plug-ins. The filter configuration can be modified dynamically, while the graph is actively streaming data. Similarly, the protocol CF exposes operations to configure a protocol stack, i.e., operations for inserting protocols in specified locations, removing, and replacing protocols. However, this CF disallows changing the stack after the stack is activated. As another example, the memory management CF allows replacing allocation policies at any time since they are stateless. In contrast, the binding CF allows replacing BT implementations only when they are not being used.

O_2 supports *extension* by enabling the incorporation of multiple CFs, each of which supports extension with respect to some specific aspect of middleware functionality. For example, the resource CF supports extension with respect to resource types, and the multimedia streaming CF in M_1 supports extension with respect to media filters.

Finally, O_2 supports *large-scale modification* in two ways, both based on the service CF. First, it supports customisation of middleware architectures by varying the set of available services, which represent coarse-grained, shared units of middleware functionality. For example, M_1 was customised by adding an event-based communication service and a power management resource service, thus forming a new platform for mobile computing environments [Parlavantzas05]. Second, O_2 supports changing the configuration of service implementations both statically and dynamically. For instance, it supports statically selecting the service implementation components that will realize the architecture-defined services, dynamically replacing these components with enhanced or modified versions, dynamically removing unused components to reduce memory footprint, or dynamically adding implementations of new services to satisfy unanticipated requirements.

5.2 Ease of modification

The ease of modification supported by O_2 is evaluated in terms of the two identified styles of modification: namely, architectural modification and system modification. First, architectural modification is facilitated mainly by the ability to derive middleware architectures as assemblies of existing CFs, which is a prominent feature of the general modifiability approach discussed in section 2. More specifically, architectural modification is facilitated by the ability to specialise O_2 by adding service roles and validator roles. Added service roles are typically attached to roles in other CFs, which are thus integrated into the middleware architecture. Another feature that facilitates architectural modification is the ability to express CFs in technology-independent terms, thus allowing CFs to be adapted to different component technologies. Finally, the layering of services in O_2 facilitates modifying architectures while structuring and reducing the impact of changes.

Second, system modification relies on the modification facilities provided by the basic O_2 CFs plus other CFs that are potentially integrated into middleware architectures. In particular, the application of the basic reconfiguration pattern by all encapsulated CFs enhances the ease of system modification for the reasons given in the list in section 2.4. In addition, the pattern promotes a separation between reconfiguring and using the middleware system by localising the reconfiguration management responsibility to CFR objects that offer and use well-known, generic interfaces.

5.3 Consistency maintenance

Inconsistencies may potentially be introduced by either of the two O_2-supported modification styles (i.e., architectural modification or system modification). As an example of the former, an architecture role may be defined as a refinement of both an O_2 role and a role of an aggregated CF, and these two roles may have conflicting constraints. As an example of the latter, dynamically replacing a plug-in that is engaged in interactions with other middleware parts may cause a system failure. O_2 currently offers no support for avoiding inconsistencies in middleware architectures; such inconsistencies are managed manually or in a semi-automated way, using consistency management facilities provided by modelling tools.

O_2 does, however, provide support for avoiding inconsistencies in *middleware systems*. This relies on the application of the basic reconfiguration pattern by all encapsulated CFs. Specifically, the pattern has three benefits with respect to consistency:

- It allows designers to provide consistency maintenance support that exploits CF-specific knowledge. For example, when adding a new service plug-in, the service CFR validates the CF-specific rule that only one instance of a service can be active in the system. As another example, when connecting two filters, the media streaming CFR validates that their connection points support a common media type, which describes the data that they will exchange.

- Since the pattern imposes that every object is associated with at most one reconfiguration manager (the manager has a composition relationship to its managed parts), the consistency management functionality does not need to account for cases in which inconsistencies are introduced through interactions with objects outside the CF (e.g., other reconfiguration managers). As a consequence, consistency management is significantly simplified. For example, consistency management is unnecessary for constraints that the reconfiguration manager enforces by construction.
- Using the pattern constrains the effects of reconfiguration to a single encapsulated system and its dependents, and thus reduces the impact of potential inconsistencies.

5.4 Efficiency

Our evaluation of O_2 with respect to efficiency is divided into two parts: an in-principle analysis of the overall overhead introduced by O_2; and an empirical performance comparison between the implemented system and two other relevant middleware platforms (GOPI and Orbacus). These two parts are presented in turn in the following sub-sections.

5.4.1 Overhead analysis

The overhead analysis centres on two types of overhead owing to i) incidental dependencies on inefficient underlying technologies; and ii) the application of the basic reconfiguration pattern by all encapsulated CFs.

First, O_2 has no direct dependencies on underlying technologies, depending instead on an abstract, minimal, component model definition that specifies only a small set of basic features and leaves open how these are implemented. As a result, the component model does not inherently induce any unnecessary overhead and permits efficient concrete realisations, such as OpenCOM as used in the current implementation [Coulson04b]. OpenCOM supports native code components that share a minimum runtime environment providing only component instantiation and introspection. Importantly, the runtime is not involved with invocations between components, which have the cost of C++ virtual method invocations.

Second, due to its non-prescriptive nature, the overhead of the basic reconfiguration pattern is largely CF-specific. Moreover, the pattern allows designers to exploit CF-specific knowledge to provide optimisations and to reduce the reconfiguration overhead. For example, since policy plug-ins maintain no state in the memory management CF, plug-in replacement can be realised without concern for state migration. Moreover, designers can make CF-specific trade-offs between efficiency and other modifiability-related requirements, such as consistency maintenance and flexibility. For example, most of the M_1 CFs employ a small number of consistency checks in order to reduce the reconfiguration overhead.

5.4.2 Performance evaluation

To examine further the potential efficiency of the O_2 approach, the performance of our M_1-based system—configured as a CORBA platform—was compared with that of two other CORBA systems, namely GOPI v1.2 and Orbacus 3.3.4 (C++ version). GOPI is a modular CORBA platform written in C and implemented in a single library. GOPI provides a useful point of comparison because a large part of its source code was reused by the M_1-based implementation. Orbacus is well known as one of the fastest and most mature CORBA-compliant commercial ORBs available.

The performance tests measured method invocations per second (over the loopback interface) between a client and a server that both reside on the same machine[4]. The configuration of the M_1-based system used in the tests contained our implementation of a remote method invocation BT underpinned by the CORBA GIOP protocol. An interface with a single

[4] Tests were performed on a Dell Precision 410MT workstation equipped with 256Mb RAM and an Intel Pentium III processor rated at 550Mhz. The operating system used was Microsoft's Windows 2000.

operation was employed that takes as its argument an array of octets and returns an array of the same size. The implementation of the operation at the server side was empty.

The results of timing a large number of round-trip invocations using this setup are shown in Figure 12.

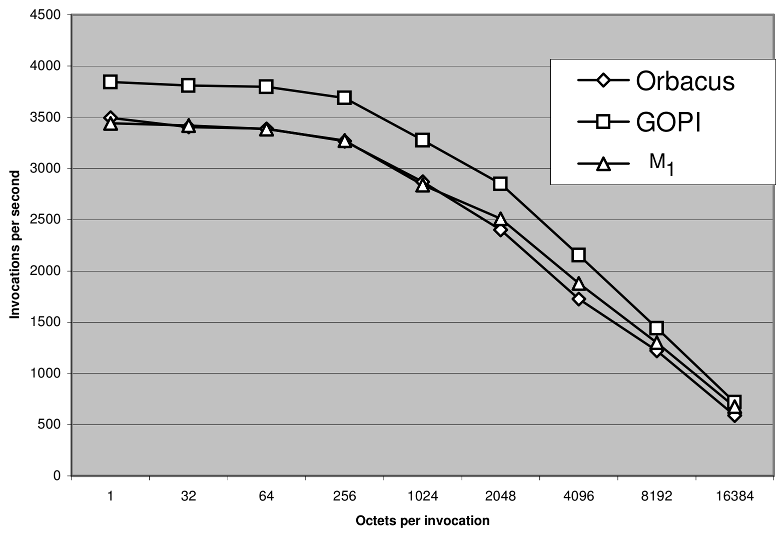

Figure 12 - Performance comparison between Orbacus, GOPI, and M_1-based system

It can be seen that for packets of less than 1024 octets, the M_1-based system performs about the same as Orbacus, with GOPI running around 12% faster. As packet size increases, the difference between all three systems diminishes—this is presumably because the overhead of data copying begins to outweigh the cost of invocation processing. Since GOPI and the used M_1 configuration share a significant part of code and design, the performance difference between them can be attributed largely to two factors: i) the generic O_2 overhead that was analysed previously; and ii) the use of the OpenCOM component model. The results show that the performance of the M_1-based system is entirely comparable to that of GOPI and Orbacus, even though these systems do not provide a comparable level of modifiability.

5.5 Evaluation summary

The preceding evaluation demonstrates that O_2 sufficiently satisfies the modifiability-related requirements we identified in section 1. In particular, it provides a high degree of flexibility with respect to designing new architectures and performing large-scale changes to middleware systems, and a variable degree of flexibility with respect to other modification facilities, which is adaptable to the needs of specific middleware architectures. The ability to integrate CFs facilitates architectural modification, and the factoring of the target system into CFs, combined with the basic reconfiguration pattern, facilitates system modification. O_2 also supports consistency maintenance because of the basic reconfiguration pattern. Clearly, application of the pattern does not guarantee consistency, but it does facilitate the design of CF-specific consistency maintenance support. Finally, O_2 satisfies the efficiency requirement as its CFs can be mapped straightforwardly to efficient component technologies, and the application of the basic reconfiguration pattern overhead can be adjusted depending on specific needs. The performance comparison provides further evidence of O_2's potential efficiency by showing that a non-trivial O_2-based middleware system can perform as well as less modifiable, non-component-based equivalents.

6. Related work

We first consider work related to our general modifiability approach and then work related to the O_2 framework. The component framework concept was introduced by [Szyperski98], but this work did not address the problem of designing or representing CFs. Our collaboration and CF modelling concepts draw on previous role-modelling approaches, such as those of [D'Souza98], [Riehle98], and [Reenskaug96], as well as UML 2.0 [OMG05a]. However, our modifiability approach specialises these concepts to explicitly target component-based development, and packages them in an accessible way as a simple abstract language. The transformation from a technology-independent CF to a technology-specific CF is analogous to the transformation from a PIM to a PSM in the MDA approach [OMG03]. However, PIMs and PSMs typically describe complete software systems, not frameworks, and the abstraction gap between them is typically larger than the gap between technology-independent and technology-specific CFs.

Our basic reconfiguration pattern has similar goals with ‘component configurators’ in [Kon00]. Unlike configurators, our pattern exploits the hierarchical structures induced by composition relationships among components, which helps simplify reconfiguration management and reduce the impact of changes. Specifically, a configurator maintains and manipulates the dependencies between a certain component and other components. In contrast, our reconfiguration manager maintains and manipulates the parts of an encapsulated system and the links among them. Our pattern is also similar to the reconfiguration support in Fractal [Bruneton02], a hierarchical component model that has been used to develop infrastructure software such as operating systems [Fassino02] and message-oriented middleware [Leclercq05]. This work, however, has not addressed design support for building and composing Fractal-based component architectures and is thus largely complementary to our work.

We now consider a selection of commercial and research middleware platforms, and assess their modifiability. Most commercial, container-based, component technologies, such as CCM and EJB, suffer from limited flexibility since they support a predefined set of services from which a fixed set of configurations is selected statically. A notable exception is .Net [Microsoft05] which provides extensible container-provided services. Web services standards are increasingly being embraced by commercial vendors. The ever-growing number of such standards has motivated the adoption of extensibility mechanisms, mainly based on configurable chains of interceptors (e.g., handler chains in Apache Axis [Apache05]). Such mechanisms, however, provide only low-level, generic support for *developing* middleware logic. In contrast, higher-level, domain-specific support is provided by CFs in O_2-based systems. Large scale modification remains generally unaddressed in commercial platforms. For example, although large-scale changes for accommodating specialised operational environments are performed regularly in the CORBA world (e.g., deriving real-time CORBA from basic CORBA), CORBA defines no systematic approach for performing such modifications.

Turning now to research platforms, FlexiNet [Hayton99] and Jonathan [ObjectWeb02] are Java-based platforms that are structured as white-box object-oriented frameworks. FlexiNet concentrates on assembling protocol stacks and supports consistency maintenance by enabling the association of constraints with stacks (e.g., constraints on possible transport protocols). Jonathan enables large-scale variations in the form of different ‘personalities’ (e.g., a CORBA or Java RMI personality) but lacks support for dynamic modification. A general limitation of platforms based on object-oriented frameworks is that they tend to embody dependencies on *implementations* (i.e., classes) rather than interfaces. This complicates performing large-scale changes in the structure and behaviour of the frameworks themselves.

OpenORB [Blair98, Costa00, Blair01] represents the first generation of reflective middleware developed at Lancaster; it features multiple ‘reflective meta-models’ for inspecting and adapting various aspects of components and bindings. OpenORB exposes a high degree of flexibility, but performing changes is difficult and error-prone since the meta-models provide only low-level primitives (e.g., component replacement). Moreover, OpenORB provides no effective support for consistency management, and the reflective facilities incur a substantial resource overhead that cannot be avoided or scaled down. The second generation of reflective middleware, OpenORB v2, uses CFs and builds on OpenCOM [Coulson02a]. As mentioned previously, our current work formalises and extends OpenORB v2, adding support for defining multiple related architectures. The OpenORB v2 architecture is essentially equivalent to the M_1 architecture.

DynamicTAO and the Universally Interoperable Core (UIC) [Kon00, Roman01] are reflective ORBs that support dynamic change by means of component configurators, mentioned earlier. Similarly to O_2-derived architectures, dynamicTAO and UIC support consistency maintenance by allowing customized implementations of configurators that exploit context-specific knowledge to validate reconfiguration requests. Moreover, flexibility and resource overheads can be adjusted by changing the number of employed configurators. Large-scale modification is addressed by UIC, but not dynamicTAO. However, UIC addresses this concern by simply proposing a 'skeleton' of abstract components that can be specialised through inserting concrete components. Unlike our work, UIC provides no design support for defining or changing such skeletons.

[Jørgensen00] presents a component-based middleware platform that supports customisation of non-functional application requirements. Specifically, customisation is realised through the dynamic selection of alternative component implementations, driven by declarative, application-specific, policies (e.g., the expected deadline associated with invocations). Customizing the platform is thus very easy for developers; but flexibility is restricted to switching between instances in a fixed run-time structure with fixed connections. Large-scale change is allowed through changing the component architecture, but this is not particularly supported. Moreover, the platform introduces a high performance overhead since policies are interpreted at the time of each method invocation.

DPRS [Roman04] is an approach to constructing dynamically programmable middleware services that relies on 'architecture externalization'; that is, exporting the structure, logic, and state of the service so that they can be dynamically inspected and modified. The approach was used to build a flexible, multi-protocol ORB, called ExORB. ExORB supports a high degree of flexibility as virtually every aspect of the system is available for inspection and adaptation. ExORB also supports consistency maintenance because the approach adopts an execution model with well-defined reconfiguration-safe states. However, ExORB cannot prevent inconsistencies that stem from violating higher-level, middleware service-specific constraints. Importantly, the approach mandates an uncommon programming model that separates state, functional units, and execution sequences of those units, thus imposing extra complexity to middleware developers. Our approach does not mandate any specialised programming model, but it can clearly accommodate them as specific CFs, if necessary.

Middleware platforms in the form of extensible containers have recently attracted both commercial and research interest. The .Net container-based technology mentioned earlier is one example. JBoss [Fleury03] is an extensible application server, which, similarly to .Net, uses interceptors to realise custom services. AspectJ2EE [Cohen04] is an aspect-oriented programming language geared towards the generalised implementation of J2EE application servers. Middleware services are implemented as aspects that are woven with enterprise beans at deploy-time. Similarly, Alice [Eichberg04] supports implementing services as aspects and relies on Java annotations to provide meta-information about components and aspect joinpoints. Such work on container/aspect-based middleware investigates primitive mechanisms (e.g., interception, metadata, aspects) that remove the need for application logic to access middleware services. This work, however, provides little or no support for implementing actual infrastructure services. Moreover, there is little support for minimising the possibility of interference between independently-developed services, which compromises the consistency of such systems. Finally, dynamic reconfiguration of services is typically lacking. For example, in .Net, the set of services provided to objects and their properties cannot be changed after object instantiation.

7. Conclusions

This paper first presented an approach to the construction of modifiable component systems as component frameworks. This approach offers a set of design tools for building CFs—namely, a set of abstract concepts expressed using a UML-based notation—and has three main features. First, it employs a minimal and abstract component model, resulting in wide applicability to various application domains and underlying component technologies. Second, it provides principled mechanisms for assembling CFs into larger ones, thus helping manage the complexity of understanding, designing, and evolving large component architectures. Third, it provides a general reconfiguration pattern which helps in designing CFs that expose easy to use and consistency-preserving facilities for dynamic reconfiguration.

Following that, the paper presented the O_2 middleware framework, our proposed solution to the requirement for middleware modifiability. Based on the generic approach to modifiability discussed previously, O_2 is designed as an assembly of basic CFs and supports two styles of modification: architectural modification, which enables large-scale, static changes, such as customizing O_2 to different application domains and underlying infrastructures; and system modification which enables smaller-scale changes, both static and dynamic, such as replacing protocol implementations. O_2's feasibility has been evaluated by providing an implementation based on a representative middleware architecture called M_1. The paper has also offered qualitative and quantitative evidence that O_2 satisfies adequately and in a balanced way the identified modifiability-related requirements of flexibility, ease of modification, consistency maintenance, and efficiency.

The three main directions for future work are: i) to expand the set of plug-in components, CFs, and middleware architectures based on O_2; ii) to provide tool support for assembling CFs, validating the well-formedness of CF models, and transforming them to component-technology specific models; and iii) to apply the modifiability approach to different domains, covering both application and infrastructure software. We have already, in recent work, successfully used a subset of the approach (namely, the idea of applying CFs that build on a minimal component technology) to address the domains of programmable networking [Coulson03] and Grid middleware [Coulson04a].

Finally, we are convinced that the key to mastering the ever-increasing complexity and variability that characterises middleware development is raising the level of abstraction. The middleware community has so far paid little attention to higher abstraction levels and powerful abstraction mechanisms, such as models, modelling languages, and frameworks. By demonstrating the benefits of our approach in enhancing middleware modifiability, we hope that this work will accelerate the adoption of such mechanisms by the community.

References

[Aagedal02] Aagedal, J.Ø., Ecklund, E., "Modelling QoS: Towards a UML Profile", *UML 2002*, Springer LNCS 2460, Dresden, Germany, September 20 - October 4, 2002, pp. 275-289.

[Apache05] Apache, Web Services - Axis, Oct. 2005, http://ws.apache.org/axis/

[**Blair98**] Blair G.S., Coulson G., Robin P. and Papathomas M., "An Architecture for Next Generation Middleware", *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (Middleware'98), Lake District, UK, Springer-Verlag, Sept. 15-18, 1998, pp. 191-206.

[**Blair01**] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K., "The Design and Implementation of OpenORB v2", *IEEE Distributed Systems Online*, Special Issue on Reflective Middleware, Vol. 2, No. 6, 2001.

[**Bruneton02**] Bruneton, E., Coupaye, T., Stefani, J.B., "Recursive and Dynamic Software Composition with Sharing'. Seventh International Workshop on Component-Oriented Programming (*WCOP02*), Monday, June 10, 2002.

[**Clarke01**] Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N., "An Efficient Component Model for the Construction of Adaptive Middleware", *Proceedings of the IFIP / ACM International Conference on Distributed Systems Platforms* (Middleware'2001), LNCS 2218, Heidelberg, Germany, November 2001, pp. 160.

[**Cohen04**] Cohen, T., Gil, J.Y., "AspectJ2EE = AOP + J2EE Towards an Aspect Based, Programmable and Extensible Middleware Framework". In: Proceedings of the 18th European Conference on Object-Oriented Programming (*ECOOP 2004*), LNCS Vol. 3086 / 2004, Oslo, Norway, June 14-18, 2004, p 221.

[**Costa00**] Costa, F., Duran-Limon, H., Parlavantzas, N., Saikoski, K., Blair, G.S., Coulson, G., "The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications". In W. Cazzola, R. J. Stroud and F. Tisato., editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science No. 1826, Springer-Verlag, Heidelberg, Germany, June 2000, pp 79-99.

[**Coulson02a**] Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N., "The Design of a Highly Configurable and Reconfigurable Middleware Platform", *ACM Distributed Computing Journal*, Vol 15, No 2, April 2002, pp 109-126.

[Coulson02b] Coulson, G., Baichoo, S., Moonian, O., "A Retrospective on the Design of the GOPI Middleware Platform", *Multimedia Systems*, Volume 8, Issue 5, Springer-Verlag, New York, December 2002, pp 340 - 352.

[Coulson03] Coulson, G., Blair, G.S., Hutchison, D., Joolia, A., Lee, K., Ueyama, J., Gomes, A.T., Ye, Y., "NETKIT: A Software Component-Based Approach to Programmable

Networking", *ACM SIGCOMM Computer Communications Review (CCR)*, Vol 33, No 5, October 2003, pp 55-66.

[Coulson04a] Coulson, G., Blair, G., Parlavantzas, N., Yeung, W.K., Cai, W., "Applying the Reflective Middleware Approach in Grid Computing", *Concurrency and Computation: Practice and Experience*, Vol 16, No 5, 25 April 2004, pp 433-440.

[Coulson04b] Coulson, G., Blair, G.S., Grace, P., "On the Performance of Reflective Systems Software", Proc. International Workshop on Middleware Performance (MP 2004), April, 2004, Phoenix, Arizona; Satellite workshop of the IEEE International Performance, Computing and Communications Conference (IPCCC 2004), pp 763-771, 2004.

[**D'Souza98]** D'Souza, D. and Wills, A., "*Objects, Components, and Frameworks with UML - the Catalysis Approach*", Addison-Wesley, ISBN: 0201310120, 1998.

[**Eichberg04**] Eichberg, M., and Mezini, M., "Alice: Modularization of Middleware using Aspect-Oriented Programming", Software Engineering and Middleware (*SEM 2004*), Linz, Austria, 20-21 September 2004.

[**Fassino02**] Fassino, J.P., Stefani, J.B., Lawall, J., Muller, G., "Think: A Software Framework for Component-based Operating System Kernels", In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, USA June 10-15, 2002.

[**Fleury03**] Fleury, M. and Reverbel. R., "The JBoss Extensible Server". ACM/IFIP/USENIX *International Middleware Conference*, LNCS Volume 2672, Rio de Janeiro, Brazil, June 2003, pp 344—373.

[**Hayton99**] Hayton, R. and ANSA Team, *FlexiNet Architecture*, ANSA Architecture Report, Citrix Systems Ltd., Cambridge, UK, February 1999. Available at: http://www.ansa.co.uk

[**Jørgensen00**] Jørgensen, B.N., Truyen, E., Matthijs, F., and Joosen, W., "Customization of

Object Request Brokers by Application Specific Policies". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (*Middleware'2000*). New York. April 3-7, 2000.

[**Kon00**] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., and Campbell, R.H., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (*Middleware'2000*). New York. April 3-7, 2000.

[Kon02] Kon, F., Costa, F., Campbell, R., Blair, G., "The Case for Reflective Middleware". *Communications of the ACM*. Vol. 45, No. 6, pp. 33-38. June, 2002.

[Kordon05] Kordon, F., Pautet, L., "Toward Next-Generation Middleware?", *IEEE Distributed Systems Online*, 5(1), 2005.

[**Leclercq05**] Leclercq, M., Quema, V., Stefani, J.B., "DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware", IEEE Distributed Systems Online, Vol. 6, Issue 9, Sept. 2005.

[**Microsoft04**]Microsoft, COM Component Object Model Technologies, Retrieved: Dec. 2004, http://www.microsoft.com/com/default.mspx

[**Microsoft05**] Microsoft, .Net Home Page, Retrieved: Oct. 2005, http://www.microsoft.com/net

[ObjectWeb02] ObjectWeb Consortium, Jonathan v3.0 alpha 10, Oct. 2002, http://jonathan.objectweb.org/

[**OMG03**] Object Management Group, MDA Guide V1.0.1, OMG Document omg/03-06-01

[**OMG05a**] Object Management Group, UML UML 2.0 Superstructure Specification, OMG Document formal/05-07-04.

[**OMG05b**] Object Management Group, OCL 2.0 Specification, OMG Document ptc/05-06-06.

[Parlavantzas03a] Parlavantzas, N., Coulson, G., Blair, G.S., "An Extensible Binding Framework for Component-Based Middleware", *Proceeding of the 7th IEEE International Enterprise Distributed Object Computing Conference* (EDOC 2003), Brisbane, Australia, September 16-19, 2003, pp 252-263.

[Parlavantzas03b] Parlavantzas, N., Coulson, G., Blair, G.S., "A Resource Adaptation Framework For Reflective Middleware", Proc. 2nd Intl. Workshop on Reflective and Adaptive Middleware (located with ACM/IFIP/USENIX Middleware 2003), Rio de Janeiro, Brazil, June, 2003.

[Parlavantzas05] Parlavantzas, N., "*Constructing Modifiable Middleware with Component Frameworks*", Lancaster University Thesis (Ph.D.), 2005.

[Reenskaug96] Reenskaug, T., Wold, P., and Lehne, O., “Working with Objects: The OORAM Software Engineering Method”. Manning/Prentice Hall, 1996.

[Riehle98] Riehle, D., and Gross, T., "Role Model Based Framework Design and Integration." In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '98). ACM Press, 1998. pp 117-133.

[**Roman01**] Roman, M., Kon, F., and Campbell R., “Reflective Middleware: From Your Desk to Your Hand”, *IEEE Distributed Systems Online*, Vol. 2, No. 5, July 2001.

[**Roman04**] Roman, M., Islam, N., “Dynamically Programmable and Reconfigurable Middleware Services”, Proc of *ACM/IFIP/USENIX 5th International Middleware Conference*, ISBN:3-540-23428-4, Toronto, Canada, October 2004, pp 372 - 396.

[**Szyperski98**] Szyperski, C., Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.