Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework.

John Keeney, Vinny Cahill

Distributed Systems Group,
Department of Computer Science,
Trinity College,
Dublin 2,
Ireland.
{John.Keeney, Vinny.Cahill}@cs.tcd.ie

Abstract

We argue that the software user, the developer, the designer and indeed the application logic itself all possess invaluable intelligence to gear how software should adapt itself to changing requirements and changing context.

We present Chisel, an open framework for dynamic adaptation of services using reflection in a policy-driven, context-aware manner. The system is based on decomposing the particular aspects of a service object that do not provide its core functionality into multiple possible behaviours. As the execution environment, user context and application context change, the service object will be adapted to use different behaviours, driven by a human-readable declarative adaptation policy script.

To demonstrate this framework we will provide a dynamically adaptive middleware for mobile computing. The framework will allow users and applications to make mobile-aware dynamic changes to the behaviour of various services of the middleware, and allow the addition of new unanticipated behaviours at run-time, without changing or stopping the middleware or an application that may be using it.

This is achieved by implementing the behaviours as metatypes in Iguana/J, which supports non-invasive dynamic associations of metatypes to service objects without any requirement to interrupt, change or access the object's source code.

1. Introduction

The principal aim of the Chisel project is to build a framework supporting unanticipated dynamic adaptation that will take account of contextual information from as many sources as possible. These sources include low-level information about the changing nature of the execution environment, but also include high-level knowledge and intelligence from the application being adapted and the user using the application. Traditional systems [2, 14, 15, 21, 22, 31-33] have failed to take into account the intelligence of the user and the application to drive dynamic adaptation, as the execution environment, the application resources and demands, the users' resources and requirements all change, possibly in an unpredictable and erratic manner. It is unrealistic to expect an adaptation framework using a "black box" approach to its adaptation intelligence to perform adequately in a generalised manner.

When an application needs to adapt it is usually not because the core problem domain of the application has changed, but rather that a non-functional requirement or behaviour of some object within the application needs to change. For example, it should not be necessary to recompile a distributed application in order to adapt to use a different network communications protocol. By separating out the aspects of the application that do not provide the core functionality of the application into multiple non-functional behaviours, the application can be adapted by changing these behaviours, without changing the application itself. This can be achieved using the concept of metatypes and reflection to implement the adaptation mechanism.

This paper describes the design of the Chisel dynamic adaptation framework. Section 2 describes why the dynamic association of metatypes with base level service objects was chosen as the adaptation mechanism. Section 3 gives an overview of the design of the Chisel framework and how adaptations are driven by a declarative policy rule script, which also describes high-level contextual information influencing how the framework should adapt base-level service objects. Section 4 describes an adaptive middleware system based on ALICE [2, 14, 15, 32] to demonstrate the Chisel adaptation framework. Section 5 describes related research, with conclusions and planned future work described in Section 6.

2. Using reflection to dynamically inspect and adapt software systems

This section describes the use of reflection as a dynamic adaptation technique, and how metatypes can be used to describe non-functional behaviours of service objects. The Iguana/J reflective architecture is also described.

2.1. What is reflection?

A reflective computational system is a computational system that reasons about and acts on itself. Reflection can be used to dynamically inspect and introspect on a computational system. Maes [19] defines a *meta-system* as a computational system that stores data (*metadata*) that represents some part of another computational system (*base-system* or *object system*), i.e. a part of its domain is another computational process. This metadata is *causally connected* to the part of the object-system that it represents, i.e. if the object-system changes then the metadata changes accordingly, or if the metadata is changed then the object system must change or adapt in a corresponding manner. A *reflective system* is a meta-system with itself as its base system. So a reflective computational system is one that contains data (metadata) representing some part of itself, data representing its functional application or domain (object data), and a program of execution to manipulate these data (both object data and metadata). This metadata can be inspected to describe some part of the system, and changed to adapt the system.

Structural reflection provides structural information about the system by providing a concrete representation of (*reifying*) structural parts of the base level as metadata (e.g. data structures in the base-level, data types used, inheritance, interfaces implemented etc). Behavioural reflection is the ability to reify and change the representation of a system and so adapt the computation and behaviour of that system. Changing the structure of the base-level system can also be used to change the behaviour of that base-level system. Similarly changing the computation or behaviour of the base-level usually involves changing some part the base-level structure of the system. Therefore it is difficult to draw a clear separation between structural reflection and behavioural reflection.

The architecture of a software system may be defined as the system's overall structure as an organised collection of interacting components [7]. It is described by the components that make up the system and how these are inter-connected. Architectural reflection is defined as computation performed by a system about its own architecture [7]. In an architecturally reflective system, the system architecture is usually reified as a data structure that is causally connected to the actual architecture of the system [11]. This can be used to dynamically examine the architecture of a system at run-time in a structurally reflective manner, but it can also be used to dynamically adapt the architecture of the system as behavioural reflection.

In an Object Oriented programming language, a meta-object is an object that stores information about the implementation and interpretation of some object [19]. The set of meta-objects that represent a particular object is that object's meta-level [30]. The set of meta-objects that represent all of the base-objects in an application make up that application's meta-level since an application is a collection of objects.

A black-box approach to system design means that implementation of a system is hidden behind a strict interface whereby the system user has no information on the internal make-up of that system. "Open Implementation" [17], based on reflection, gives the system a second "meta interface", which is separate to the traditional (base) interface, to have the system adjust its own implementation. The communications between the base level and meta-level takes place through a set of well-defined interfaces. These interfaces together are referred to as the meta-object protocol or MOP for short [18]. A MOP allows the user to incrementally modify the implementation and behaviour of a programming language [18]. In an OO programming language, a MOP can be seen as an extension to the language's object model, as it specifies which parts of the object model may be reified and possibly changed.

2.2. Metatypes as an adaptation mechanism

An object's type will describe the functional behaviours that are directly related to the part of the core application domain being modelled by that object. Schäfer [30] introduced the concept of a metatype as a characterisation of an object's own object model, and as such its non-functional behaviour and structure. Examples of metatypes include: verbosity, remote accessibility, persistence, debugability, fault tolerance or optimisation (see figure 1). An object's metatype may be orthogonal to its type since the behaviours described in metatypes are not those inherent behaviours of the entity being modelled by the object, i.e. behaviours that are not directly related to the part of the core application domain being modelled by that object. A metatype can be implemented using meta-objects to implement a non-functional behaviour. An example involves adding persistence behaviour to an object by associating that object with a set of meta-objects that implement persistence, regardless of the functional data, interface or behaviour of the object. Objects of a single type may have multiple metatypes associated with them. Several objects of different types may have the same metatype associated with them. Ideally, this association of metatypes should occur transparently to the objects, so the objects can be written in a manner completely unaware of any metatype that may be

applied to it, with no changes to the object or its code, and without interrupting any current operation of the object.

The Iguana reflective programming model [13, 26, 27, 30], is a reflective programming extension for object-oriented languages. Iguana provides a framework to allow metatypes to be defined, implemented as meta-objects, and associated with objects without changing those objects' type. It was introduced [13] as a language independent model to incorporate meta-object protocols into high level programming languages. It was later refined into Iguana/C++ [30], then support for unanticipated adaptation with Java was added in Iguana/J [26, 27].

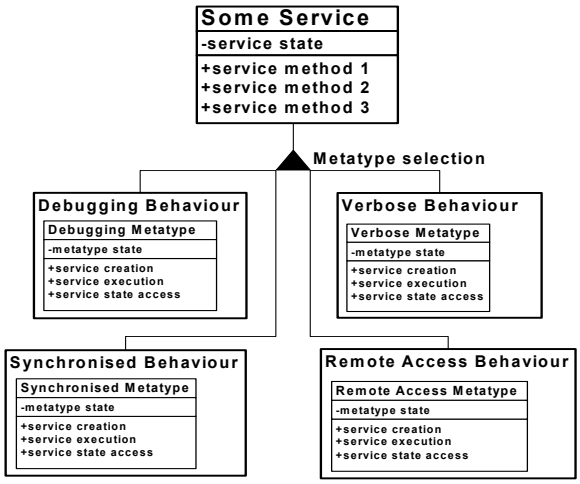

Figure 1. Example service object with four possible behaviours

Iguana supports the definition of multiple MOPs by providing a framework to allow the meta-level programmer to choose which parts of an object's object model to reify (see "reification categories" in [13, 26, 30]). Each part of the object model that is reified is represented by a meta-object, which is an instance of a meta-object class. In Iguana, a MOP is the selection of which parts of the object model to reify, and the association of a meta-object class to each of these reification categories.

In Iguana, metatypes are implemented by deciding which parts of the object model to reify, writing a set of meta-object classes for these reified elements that embeds the new metatype behaviour and then associating that MOP with an object, class, or interface. The term "metatype selection" is used to refer to this association of MOP implementations to objects. Iguana supplies the framework to dynamically instantiate these meta-objects to reify the object model and correctly order and compose them if more than one metatype is selected. Another novel contribution of Iguana is the ability to have objects select new metatypes at run time, thereby dynamically adding new non-functional behaviours to the system, without changing the type of the object.

2.3. Iguana/J

Iguana/J [26, 27] implements the Iguana reflective architecture for the Java programming language. It supports runtime reflection, whereby meta-objects exist at runtime rather than compile-time, so reified operations are redirected to the appropriate meta-objects. The Iguana/J runtime operates by extending the Java JVM using the JIT interface, so meta-objects can be associated with base-level object operations at load-time or at run-time

The MOP is declared in a protocol declaration file, by declaring which parts of the object model will be reified:

```
protocol MyProtocol1{
        reify Creation:MyCreate1();
        reify Execution:MyExecute1();
}
```

The code for the meta-object classes is provided in Java, with each class extending the default class for that reification category:

```
import ie.tcd.iguana.MExecute;
class MyExecute1 extends MExecute {
  Object execute(Object o,Object [ ] args, Method m){
        System.out.print("Executing:"+m.getName());
        return m.invoke(o,args);
  }
}
```

The association of this MOP and its associated meta-objects classes (a metatype) can be made statically to any class or interface in a protocol selection file:

```
MyClass1==>MyProtocol1();
java.net.Socket==>MyProtocol1();
```

Metatype association can also be performed dynamically from within base-level or meta-level code, to any class, interface or object:

```
import ie.tcd.iguana.Meta;
MyClass1 myObject1 = new MyClass1();
Meta.associate(myObject1,"MyProtocol2",args);
```

With this mechanism Iguana/J maintains a high degree of separation of concerns since there is no tangling of meta-level code and application-level code. New meta-objects to implement new metatypes can be written at any time and dynamically associated with any class, interface or object.

The ability to dynamically associate metatypes with an application's objects allows the object model of that application to be completely changed in a manner that is transparent to that application since the type of any object that selects a metatype is unchanged. Iguana/J does not require any access to the application source code since metatype association occurs at load-time or runtime. Metatypes can even be associated with any third party application classes or objects. The application does not need

to be restarted or altered in any way since interception occurs every time the reified operation is performed.

While this has obvious advantages as a mechanism for dynamic adaptation in response to changing context, it ignores the fact that the application itself and the user are most knowledgeable about how an application should be adapted as its operational context changes.

3 The Chisel dynamic adaptation framework

Chisel is an adaptation framework that supports service object adaptation in a resource aware, application aware and user aware manner. This section describes policy-based control, the design and operation of the Chisel framework, and the policy language to be used to control context-aware adaptation in the Chisel framework.

3.1. Policy based control

A policy rule is defined as a rule governing the choices in behaviour of a managed system [8]. Management action policies are defined as persistent, positive or negative, imperatives or authorities for a set of policy subjects to achieve goals or actions on a set of target objects [8]. Informally, a policy rule can be regarded as an instruction or authority for a manager to execute actions on a managed target to achieve an objective or execute a change. An adaptation policy rule is usually made up of a trigger for the rule, which is often fired as a result of a monitoring operation, an action to perform in response to the trigger and a target for the action, which describes which managed part of the system to enforce the rule upon. Many policies will also contain some restrictions or guards confining the rule action to appropriate occasions.

Many traditional adaptable systems [2, 14, 15, 21, 22, 31, 32] are composed of a single adaptation manager that is responsible for the entire adaptation process; i.e. monitoring, adaptation selection intelligence and performing the actual adaptation. Since the intelligence to select appropriate adaptations and the mechanism to perform these adaptations in embedded directly within the adaptation manager, this type of system becomes inflexible and inappropriate for general use.

By decoupling the adaptation mechanism from the adaptation manager, and removing the intelligence mechanism to select or trigger adaptation, the adaptation manager becomes more scalable and flexible. Since the user and the application are often most enabled to make informed choices, which are based on high-level contextual or semantic information about how a system should adapt, then it is logical that the user and the application help drive the adaptation of the system.

Policy specifications maintain a very clean separation of concerns between the adaptations available, the decision process that determines when these adaptations are performed and the adaptation mechanism itself. Policy specification documents are persistent text-based declarative representations of policy rules, where the document can usually be edited then interpreted to support the addition of new rules. Policy declaration files can be read, understood and generated by users, programmers and applications.

In order for an adaptation to occur, the context changes that may trigger some adaptation must be monitored. The context manager should then leverage all available context knowledge and intelligence to determine if some adaptation is required. A separate adaptation mechanism, controlled by an adaptation manager can then perform this triggered adaptation as a response to an adaptation request.

3.2. Design of a context-aware, policy controlled adaptation framework

We propose the Chisel dynamic adaptation framework, which will adapt service objects in a context-aware policy-based manner, using metatypes. A policy-based approach was chosen to drive the adaptation mechanism by incorporating user and application specific semantic knowledge and intelligence, combined with low-level monitoring of execution environment.

The system is based on decomposing the aspects of the service objects that do not provide the core functionality of that service into multiple possible non-functional behaviours. These behaviours of the service objects will be implemented as Iguana/J metatypes that can be statically or dynamically associated and disassociated with the service object, in a completely dynamic manner, without stopping, overwriting or changing the application or object's code in any way. New service behaviours can be written and incorporated into the system at any time, even while the service objects are operating. This system will include in the policy document a-priori information for adaptation (*self adaptive system*[11]) in the form of default behaviours and known adaptations. To cope with adaptation requirements that were unprecedented when the service was designed and compiled, new reconfiguration intelligence (*adaptable system* [11]) can be incorporated at runtime by altering the policy declarations and the inclusion of new behaviours in the form of new metatypes.

The application and the service objects will be dynamically managed and adapted by a meta-level adaptation manager. This meta-level coordinator will have full access to the data in the application and so can adapt the application and service operation in a user-aware and application-aware manner by dynamically selecting different behaviours for the middleware services.

The adaptation manager can access information on local resources to trigger evaluation of the policy rules, which may force adaptation of the base-level objects. Rule evaluation can also be triggered by high-level contextual information passed via the policy script. New behaviours

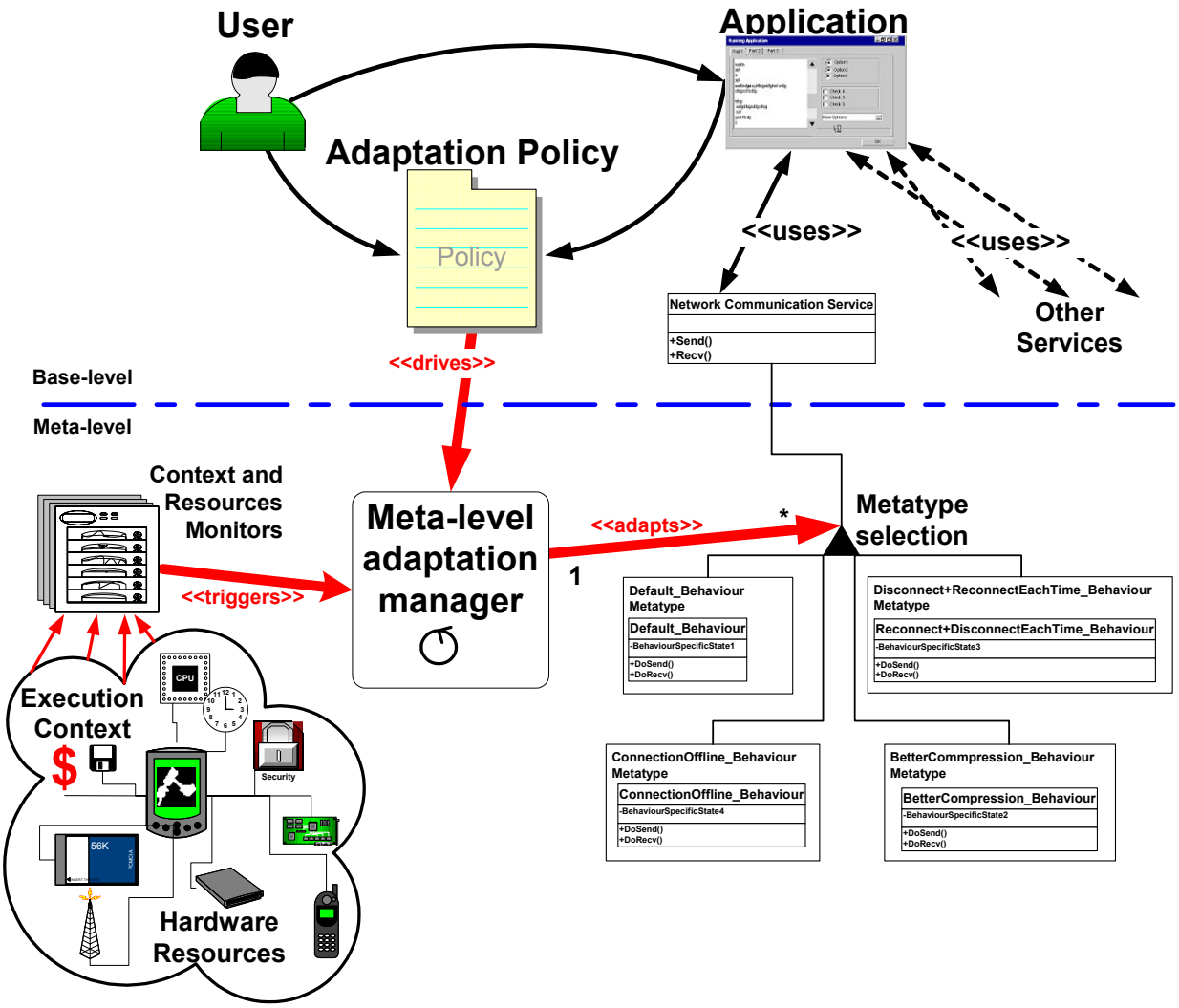

Figure 2. Design of a meta-level adaptation framework to support policy-driven context-aware adaptation of base-level service objects

can also be written and made available for selection dynamically.

As the execution environment, user context and application context change, the service objects will be adapted to use a different behaviour in a user-aware, application-aware and resource-aware manner driven by the human-readable declarative adaptation policy. The system designer, system developer, application and user can all control how the system will adapt through this adaptation policy document. The meta-level adaptation manager will interpret the adaptation policy while asynchronously or synchronously monitoring context changes that may trigger behaviour reselection for service objects.

3.3. Resource monitoring, event triggering and initiating adaptation

The metal-level adaptation manager will be broken into a further series of managers and services (see figure 3). In conjunction with an event service, events can be registered with each resource of interest, to be thrown if significant changes occur. Each resource can also be polled or operate a callback mechanism to inform listeners when significant changes occur in its context. Events can also be thrown when other environment contexts change. This includes user context, application context and execution environment context. Users and applications can trigger events using the policy declaration file. These user defined events can be triggered by other events, at certain times or dates, or directly by the user.

A context manager will monitor these resources for appropriate contextual changes. When context changes occur, the context manager, in conjunction with the policy rule manager, will check the policy rule set to identify relevant policy rules that may trigger adaptation relating to the specific context change.

The behaviour manager will be responsible for performing behaviour adaptation by performing a metatype reassociation on a managed service object, while keeping track of the set of managed services and available metatypes, and incorporating new metatypes dynamically into its behaviour set.

A policy manager will be responsible for tracking changes to the policy declaration file. This interpreted file will be incrementally parsed, with rule declarations translated into a series of rule objects, and user and application contextual information passed onto the context manager via the event service.

A rule manager will be responsible for evaluating the rules passed from the policy manager; in conjunction with the context information passed from the context manager, thereby triggering appropriate behaviour changes by the behaviour manager.

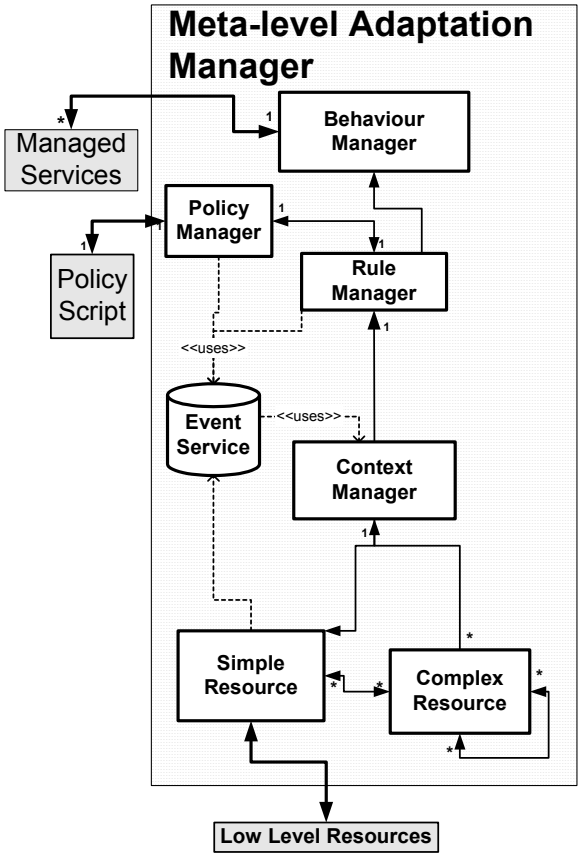

Figure 3 . Design of the meta-level adaptation manager

3.4. Chisel policy language

Because of the relatively uncomplicated nature of the adaptation policy directives needed for Chisel, it was decided to design a new policy declaration language. It was decided that there would be no appreciable benefit from incorporating a fully functional scripting or policy language such as Esterel [1], Jess [29] or Ponder [8].

There are two distinct parts to the policy language described in this section. The first part is the adaptation rules themselves:

```
ON Event:
    ManagedService.NewBehaviour
        IF Constraints allow
```

This part of the language is based on a simple event, condition action rule similar to Ponder [8]. When an event is triggered by the context manager in response to a significant change in a monitored resource, any appropriate rules will be evaluated to check if the selected managed services should be adapted to use the new behaviours specified. The new behaviour specifies which metatype the behaviour manager should associate with the managed service object. All monitored resources, the managed services themselves, and the triggered event can be queried for more information to evaluate a series of constraints or guard statements to focus the application of this adaptation policy rule. If an adaptation is necessary the behaviour manager will be requested to perform the adaptation.

This type of rule can also be used to perform more complex event operation such as filtering:

```
ON Event:
    Event.Operation
        IF Constraints allow
```

Here when an event is triggered it can be used to perform an operation on another event, such as TRIGGER, CLEAR, DISABLE, ENABLE to disable, re-enable or clear the triggered state of an event, coordinated through an event service.

The second part of the policy language supports the definition of new events dynamically.

```
NEW Event Trigger Condition
```

This demonstrates how a new event is declared that can be automatically triggered by the event service if some condition holds. Triggers include ON, EVERY, and AT, to have the event trigger at a certain time, or periodically, or WHEN to define a constraint to be calculated as above.

3.4.1. Specification of new rules

This section introduces a series of examples to illustrate how rules are specified:

```
ON WirelessDisconnect:
  NetworkConnectionService.WiredBehaviour
    IF NetworkConnectionService.WiredAvailable == True
          &&
          WirelessDisconnect.IsTemporary == False
```

Figure 4. Policy Rule 1:

Here *WirelessDisconnect* is an event, probably throwable by a network resources monitor, which signifies

that a wireless connection has become disconnected. It has a boolean field called *IsTemporary* that may signify that the resource monitor knows that this disconnection is of a temporary nature. If the *WirelessDisconnect* event triggers, this rule will be evaluated and executed. *NetworkConnectionService* is the target service, and *WiredBehaviour* is a possible behaviour for that service. The service *NetworkConnectionService* has a boolean field called *WiredAvailable* that can be queried.

When the event *WirelessDisconnect* triggers, the adaptation manager will select the *WiredBehaviour* metatype for the *NetworkConnectionService* service if the *WiredAvailable* boolean field has the value **True**.

```
ON MemoryLow:
    DisconnectedObjectAccess.LowMemoryCachingBehaviour
        IF True
```

Figure 5. Policy Rule 2

In this case *MemoryLow* is an event, probably throwable by memory resources monitor, which signifies that the system is running low on available memory. If the event triggers, the *DisconnectedObjectAccessService* service object will always have its metatype changed to that implementing a *LowMemoryCachingBehaviour*.

```
ON UserVeryInterested:
  ReplicatedDatabase.ResolveConflictBehaviour
      .AskUserIfNeededBehaviour
```

Figure 6. Policy Rule 3:

Here *UserVeryInterested* is an event that signifies that the user is knowledgeable and interested enough to manually resolve conflicts that may occur in a replicated database service called *ReplicatedDatabase*. This event is possibly throwable by an application from a slider bar on an advanced preferences control, or perhaps thrown in response to a statement in the adaptation policy by an advanced user. Since behaviours can themselves also be services, one of *ReplicatedDatabase's* behaviours is itself a service, *ResolveConflictBehaviour*, with a possible metatype called *AskUserIfNeededBehaviour* that presumably allows the user to help resolve conflicts.

```
ON UnluckyDay:
      ReallyUnluckyDay.TRIGGER
  IF TimeService.Today.Date.dd == 13
```

Figure 7. Policy Rule 4:

In this rule *UnluckyDay* is an event (see figures 8 and 9 below). When it is raised, event *ReallyUnluckyDay* will also be triggered if the *Today.Date.dd* integer stored in the *TimeService* service is equal to the value **13**.

3.4.2. Specification of new events

A series of examples illustrating how dynamic events are specified in this section:

```
NEW UnluckyDay Every Friday
```

Figure 8. Dynamic event definition 1

In this example the user is dynamically defining a new event called *UnluckyDay* that will be thrown every Friday. When the adaptation policy script is parsed by the policy manager, the new event will be added to the event service, with an instruction to trigger the event every Friday. This slightly extreme example shows the flexibility of the system to incorporate user-specific semantic information, which may be important to one user but silly to another.

```
NEW ReallyUnluckyDay
```

Figure 9. Dynamic event definition 2

The user has defined another dynamic event called *ReallyUnluckyDay* that will not be automatically thrown by the event service. This type of event will be typically thrown as a consequence of an adaptation policy rule. See figure 7 above.

4. Chisel in use: adaptive middleware for mobile computing

This section describes the use of the Chisel dynamic adaptation framework to implement context-aware dynamically adaptable middleware services for mobile computing, based on the ALICE framework [2, 14, 15, 32]. Section 4.1 introduces mobile computing and the need for context awareness in mobile computing. Section 4.2 introduces middleware for mobile computing, with an in depth description of the ALICE framework described in Section 4.3. Section 4.4 describes the current progress made with this implementation.

4.1. Motivation for context aware adaptation in mobile computing

"Mobile computing" can be considered an extension of distributed computing, whereby portable devices have access to (possibly remote) services regardless of their movement or physical location. Unfortunately, the ability to take distributed applications to a mobile computing domain comes with a very high price. Mobile devices are generally more fragile than stationary computers. They can be easily broken, reset or stolen. They are usually poor in resources such as memory, processing power or battery life. However the main difficulty with mobile computing is communications via a network connection. The characteristics of this connection can range from an inexpensive, very high bandwidth with low latency connection such as high-speed LAN, to a very expensive,

low bandwidth with high latency connection such as GSM or infrared. Even the network address of the machine can change. Mobile applications should also be able to handle periods of disconnection. The application and data characteristics, and the user's context requirements and limitations may all change dynamically. Any of these contextual conditions can change without warning and to values unknown and unprecedented by the application designer, thereby exacerbating the need for dynamic adaptation in mobile applications. Examples include when the device becomes out of range for wireless connections, when the user leaves work, or when a user suddenly disconnects the device from its synchronisation cradle to go to a meeting.

Many researchers have already focused on how adaptive applications can be adapted to improve performance for specific environment characteristics, especially network connection conditions (see Section 5). As an example we examine the adaptations possible to cope with an erratic network connection. In the presence of a high quality network connection it is possible to have a lot of network communication. One of the main indications of this is the promptness of consistency maintenance messages for replicated or cached data. However, when the quality of this connection plummets, it is necessary to maintain application functionality while using available network resources efficiently using varied techniques such as batching, filtering, compression, cache pre-fetching or protocol reselection.

Fortunately the application and the user already have a great deal of semantic information about the requirements of the application, possible performance improvements, future contextual changes and how to cope best with changes. In a mobile aware application, the application can tell a great deal about how these resources are being used and suggest ways to further improve the efficient use of these resources. The major disadvantage of mobile awareness is that it is up to the application developer to write much more than just the application specific code since the application must control the system adaptation to support changes in the environment.

4.2. Middleware for mobile computing

Middleware should shelter applications from the underlying environment, communication subsystems and distribution mechanisms, thereby providing a single view of the underlying environment as seen in systems such as COM+ [21] Java RMI [31] and CORBA [22]. A middleware system for mobile computing must be flexible to account for heterogeneous, erratic execution contexts.

It is desirable that an adaptable middleware for mobile computing is open, to allow the application and the user to inspect the execution environment and manipulate the application and middleware in a mobile aware manner, using application specific and user specific semantic knowledge. This open model for middleware is a break from the black-box model of traditional middleware, yet middleware for mobile computing should maintain a homogeneous interface and programming model for the application

4.3. ALICE

ALICE (Architecture for Location-Independent Computing Environments) [2, 14, 15, 32] is an architectural framework to support mobility by providing a range of application level client/server protocols (see figure 10). ALICE allows these protocols to provide their own support for location management, disconnected operation and connection management. In ALICE, Mobile Hosts (MH) are mobile devices with a connection to a fixed computer called a Fixed Host (FH). These connections are tunnelled through Mobility Gateways (MG), which are also fixed machines. The Mobile Host can become disconnected from a Mobile Gateway and later become reconnected to a different Mobility Gateway without interfering with the connection to the Fixed Host.

ALICE is made up of a series of layers. The Mobility Layer (ML) handles communications between devices by overriding socket function while hiding which communication interface is being used for the connection. The ML tracks available connections and picks one using a reconfigurable selection algorithm while providing performance statistics on the different available communication interfaces. The ML also manages connections between the Mobile Host and the Mobility Gateway in a mobile-aware manner using application callbacks to inform the layers above that a disconnection or reconnection has occurred. An application protocol specific Swizzling Layer resides above the ML and supports mobility of servers by translating server references and redirecting client connections to more up to date server references. ALICE has been implemented in the C and Java programming languages. Versions exist for Java RMI [31] and CORBA [22].

At present, ALICE does not provide support to force disconnections to select a different communication connection. In the current system a disconnection must occur before a new connection can be selected in a resource-aware and context-aware manner using a new reconnection algorithm. A Disconnected Operation layer has been designed for ALICE that will synchronously queue unsent data between Mobile Host and Mobility Gateways while the Mobile Host is disconnected. The main issue with ALICE is the relative difficulty to control which connection to use and how to incorporate more semantic information to make a more informed choice about how the ML should reconnect the Mobile Host and the Mobility Gateway.

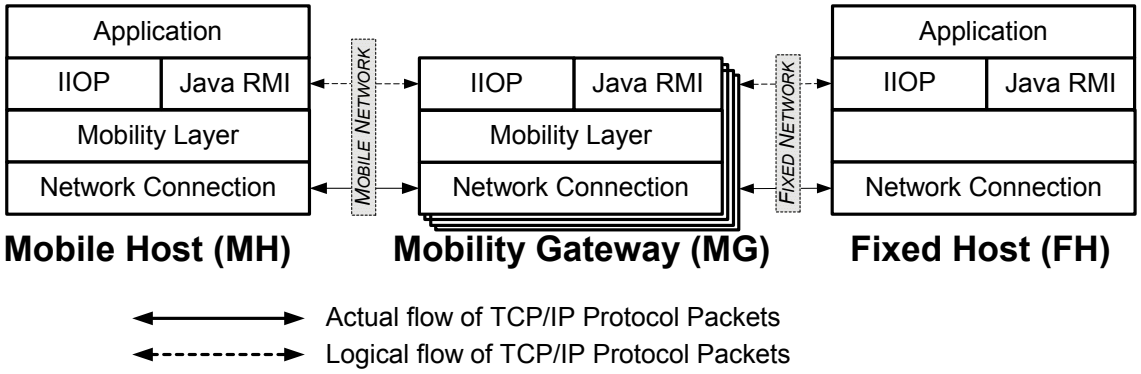

Figure 10. The ALICE framework

4.4. Implementation of a middleware based on ALICE using the Chisel framework

We have begun implementing open adaptive communication services for middleware for mobile computing based on the ALICE framework. The implementation in progress reimplements ALICE to use policy-driven dynamically adaptive service objects as described in the Chisel framework in Section 3. The user and the application will now drive the adaptation of ALICE in response to operation environment changes but also user and application context changes.

In conjunction with this, we will be starting the implementation of the policy and rule managers to interpret a policy declaration script that uses the policy language described in Section 3.4**.** Applications will then be able to function without any change or disruption as they operate on mobile devices in a user specific, application specific and mobile aware adaptive manner.

By statically or dynamically associating different metatypes with the standard *java.net.Socket* class [31] in a completely transparent manner, a simple Java chat client currently runs without disruption as a connection switches between direct connection to a server, to connection through a tunnelling gateway, to connection via a fully functional mobility layer using ALICE. Absolutely no changes were required to the application code and all adaptations can occur without stopping the client or server application. By performing this behaviour change the chat client application now supports long periods of disconnection and reconnection via a different network interface without breaking the socket connection in any way. Currently this adaptation occurs transparently to the application, but work is in progress to fully incorporate the adaptation policy mechanism to perform this adaptation in a completely context aware manner.

5. Related Work

This section describes a selection of related research in the fields of policy-based control, adaptive middleware systems, and context aware systems.

The Ponder policy language [8], developed at Imperial College London, is a declarative, object-oriented language for specifying security and management policies for distributed object systems. The policy language described in this document is loosely based on Ponder obligation policies. In this system, as events occur due to changes in context, the adaptation manager is obliged to adapt the behaviour of underlying system services. A fully functional policy language to specify security constraints is however outside the constraints of this research.

Also created in Imperial College London, GEM [20] is a Generalised Event Monitoring language used to program events and event monitors. It supports the generation, processing (merging, filtering, validation), dissemination (registration, distribution) and presentation (event abstractions or views) of events. The language used in the framework described in this document closely resembles the language used in GEM to define and respond to events.

The M3 [24] project from the University of Queensland have designed an adaptive middleware framework that supports adaptation in a context aware manner. This is achieved using a Mobile Enterprise Architecture Description Language (MEADL) script to dynamically re-configure how application and system components interact which each other. While this system has many similar design ideas to this project, the M3 system has some important drawbacks. The adaptation mechanisms prototyped (including filtering, object migration, interface restrictions and web content adaptation) all lack the generality and openness of a reflective mechanism like Iguana. The MEADL rules for a "role" can include contextual information by detecting if the role is currently operating in or out of a named context. This approach is extremely limiting however since context characteristics cannot often be measured as a boolean state,

so a ranged metric value should provide more expressiveness and accuracy.

Presented by the DistriNet research group in Katholieke Universiteit Leuven, Correlate [16, 28] is a concurrent object-oriented language based on C++ (and later Java) to support mobile agents. It has a flexible run-time engine to support migration and location independent inter-object communication. Each agent object has an associated meta-object that can intercept creation, deletion and all invocation messages for the object. This system allows non-functional aspects of the application to be separated from the application object, in a manner very similar to metatypes described in Iguana above. The meta-level system was initially used to implement load-balancing, real-time operation, security and persistence and later used to customise ORBs to use application specific preferences. Application specific information is included in high-level policies, which are consulted by the meta-level before using the non-functional aspects of the application. However this policy system is limited by imposing templates for these policies. These templates cannot be changed so the need for adaptation in response to unanticipated context cannot be fully handled so unanticipated forms of dynamic adaptation are very difficult to achieve in this architecture.

OpenORB [3,5] is a reflective middleware designed at Lancaster University. At load time, appropriate components are selected and composed as a middleware instance. Using reflection, components can also be changed or loaded at run-time. Every object is associated with a "meta-space" that can be accessed through one of the "meta-model" interfaces: "encapsulation", "composition" and "environment". This system was prototyped using the python programming language. Also described [3] is a mechanism for management components can be added dynamically to the component graph to both monitor and strategically adapt the middleware in a procedural, policy controlled manner. These event-based scripted and interpreted adaptation controllers can be dynamically changed to facilitate changing context and requirements. However a high level view of how the system should adapt is lacking here.

OpenORBv2 [4], a revision of OpenORB, is a component based reflective middleware also designed at Lancaster University. It is implemented using the OpenCOM component framework and provides a CORBA compliant interface. The OpenCOM component framework [23] is built on top of a subset of Microsoft's COM. OpenCOM provides low-level support for meta-models, using a series of interfaces to COM type objects that are encapsulated with the custom service being developed. These interfaces provide support for dynamic insertion of interception mechanisms, support for viewing dependencies of components, and access to the component graph for each component. Also included in OpenORBv2 is a "resources" metaspace to represent the resources needed and used by components.

DART (Distributed Adaptive Run-Time) [25] is a reflective run-time for distributed adaptation developed by Sony, Japan. A framework for reflective objects is provided using meta-level method implementation selection (adaptive methods) similar to the strategy pattern [12] to facilitate internal application adaptation. Also included a method interception system to call a set of meta-objects before and after invocation, (reflective methods), to facilitate adaptation of the application's environment in a manner similar to Iguana. The code to make an object reflective and adaptive is completely mangled with the application code so there is very little separation of concerns in this framework. How the system adapts is specified in global policy functions that register for adaptation events and can introspect on both the base level and meta-level code. Policies can be loaded and unloaded at runtime. DART also uses a description file that is used at load-time to configure the system. This file can be changed at any time to affect how the system will load in future invocations, however there doesn't seem to be support for adding new behaviours at run-time.

K-Components [10, 11] uses asynchronous architectural reflection to build context-adaptive software. The adaptation logic specifying adaptive behaviour is written as adaptation contracts in a declarative programming language (ACDL). Adaptation occurs in response to adaptation events raised by the application components or from the evaluation of adaptation rules themselves. If adaptation is required, a component can be removed from the system configuration graph and another component, exposing the same interface, can be swapped in. The main issue with K-Component in relation to this system is its inability to accept new types in the configuration graph since the configuration graph is a static representation of the architecture of the system. The system also requires that the adaptation event types are known to the configuration manager at design-time, so very little support is included to initiate adaptations in response to unanticipated or un-typed events, as will likely occur in a mobile or pervasive computing environment.

The CARISMA project [6] in UCL is a middleware system made up from adaptable services. It uses context-aware application-specific semantic information, such as resources required or location information, in an "Application Profile" encoded in XML. When used, the middleware checks the application profile document and compares with the current execution context to evaluate which behaviour or policy the service component should use when providing its service. Applications may change their profiles in a reflective at run time to adapt the system as application-specific and user-specific requirements of the application change dynamically. This system is based on the provision of multiple implementations of the same service but with different behaviours in a manner similar to the

strategy pattern [12] unlike this system described here that adapts the service itself.

RAM [9] from École des Mines de Nantes, France take the approach of completely separating functional and non-functional aspects of an application in a manner related to Aspect Oriented Programming. Using this separation of concerns approach, only the core application functionality is inserted into the application code, with all middleware services represented as non-functional concerns. The system is adapted at run-time by means of an adaptation engine, an application policy and a system policy. The system policy is a low level policy that contains adaptation rules for the system in an application transparent manner. The application policy is a higher-level policy that contains rules to adapt the system in an application-aware manner but does not contain low-level execution environment information. This system, while having the advantages of cleaner separation of concerns, will allow greater inconsistencies to occur, as the application policy may conflict with the operation of the system policy. The current system does not support dynamic changes to the policies, and so cannot support unanticipated adaptation, however this is planned for future versions.

Developed by Sony Computer Science Laboratory, Apertos [33] is a reflective object-oriented operating system. In Apertos, each object encapsulates state, methods and a virtual execution processor. Each object is associated with a set of meta-objects (metaspace) that defines the semantics and behaviours (object model) for the object. The metaspace also acts as the virtual processor that can be tailored for the objects associated with it and later adapt itself to provide optimal support for the object. In order to adapt an object at run-time, it is migrated to a different metaspace (group or hierarchy of meta-objects) that provides the new desired behaviour. Apertos is the first example of a reflective object oriented operating system that models operating system services as behaviours provided by an object's meta-level. The Apertos approach of modelling behaviours as adaptable, low-level operating system level entities is in some ways similar to approach described here. However, the framework described here is intended to run on top of a configurable network enable operating system so it will support adaptation at a higher level of abstraction than Apertos. Apertos also does not support dynamic addition of new behaviours by the dynamic creation of metaspaces, since metaspaces are compiled down to ordinary code to be used at run-time. Apertos also has no structured support for context-aware behavioural adaptation.

6. Conclusions and future work

This document describes a general-purpose adaptation framework, called Chisel, that gears its adaptation based on the changing contextual resources and requirements of the user, application and execution environment. In order to maintain the general nature of this framework it proved necessary to open up the adaptation system to allow external intelligence and contextual information to drive the adaptation decision process. A human readable declarative policy script was chosen as an easy to use, generalised and extensible solution to passing this data to the adaptation manger.

A context-aware, dynamically adaptable middleware for mobile computing was chosen as a prototype application of this framework. This middleware will provide adaptable services, such as network communications, for the applications residing above it, without interrupting or changing the code of the applications or the services in any way. The policy script will allow the application and the user to drive the adaptations in manner most appropriate to their own requirements and available high-level resources.

The implementation of the adaptation framework and the middleware services is currently underway with favourable initial results. We expect completion the system within the next few months.

7. References

[1] G. Berry, "The Foundations of Esterel", in Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling, and M. Tofte, Editors. MIT Press. 1998

[2] G. Biegel, V. Cahill, and M. Haahr. "A Dynamic Proxy-Based Architecture to Support Distributed Java Objects in Mobile Environments". in International Symposium of Distributed Objects and Applications, (DOA 2002). Irvine, CA. 2002

[3] G.S. Blair, et al., "Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform". IEE Proceedings - Software. 147(01): p. 13-21. 2000

[4] G.S. Blair, et al., "The Design and Implementation of Open ORB v2". IEEE Distributed Systems Online. 2(6). 2001

[5] G.S. Blair, et al. "An Architecture for Next Generation Middleware". in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98). Lake District, UK: Springer-Verlag. 1998

[6] L. Capra, W. Emmerich, and C. Mascolo. "Exploiting Reflection and Metadata to build Mobile Computing Middleware". in Workshop on Mobile Computing Middleware. Co-located with Middleware 2001. Heidelberg, Germany. 2001

[7] W. Cazzola, et al. "Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification." in proceedings of 6th Reengineering Forum (REF'98). Firenze, Italy: IEEE. 1998

[8] N. Damianou, et al. "The Ponder Specification Language". in Workshop on Policies for Distributed Systems and Networks (Policy2001). HP Labs Bristol. 2001

[9] P.-C. David and T. Ledoux. "An Infrastructure for Adaptable Middleware". in DOA'02. Irvine, California, USA,. 2002

[10] J. Dowling and V. Cahill, "Dynamic Software Evolution and the K-Component Model". Workshop on Software Evolution, OOPSLA. 2001

[11] J. Dowling and V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptive Software". Proceedings of Reflection 2001, LNCS 2192. 2001

[12] E. Gamma, et al., "Design Patterns: Elements of Reusable Object-Oriented Software": Addison Wesley. 416. 1994

[13] B. Gowing and V. Cahill, "Meta-Object Protocols for C++: The Iguana Approach". Reflection '96. p. 137-152. 1996

[14] M. Haahr, R. Cunningham, and V. Cahill. "Supporting CORBA Applications in a Mobile Environment." in MobiCom '99: 5th International Conference on Mobile Computing and Networking. Seattle. 1999

[15] M. Haahr, R. Cunningham, and V. Cahill. "Towards a Generic Architecture for Mobile Object-Oriented Applications". in SerP 2000: Workshop on Service Portability. San Francisco. 2000

[16] B.N. Joergensen, et al. "Customization of Object Request Brokers by Application Specific Policies". in Middleware'2000 conference. New York, USA. 2000

[17] G. Kiczales, "Beyond the Black Box: Open Implementation", in IEEE Software. p. 8-11. 1996

[18] G. Kiczales, J.d. Rivieres, and D. Bobrow, "The Art of the Metaobject Protocol": MIT Press. 1991

[19] P. Maes, "Computational Reflection", PhD, Vrije Universiteit Brussels, 1987

[20] M. Mansouri-Samani, "Monitoring of Distributed Systems", PhD, Department of Computing, Imperial College, London, 173. 1995

[21] Microsoft_Corporation, "COM+ (http://www.microsoft.com/com/tech/COMPlus.asp)". 1999

[22] Object_Management_Group, "The Common Object Request Broker: Architecture and Specification (OMG Document formal/02-06-01)", formal/02-06-01, 2002

[23] N. Parlavantzas, et al. "Towards a Reflective Component Based Middleware Architecture". in Workshop on Reflection and Metalevel Architectures. Sophia Antipolis and Cannes, France. 2000

[24] A. Rakotonirainy, et al., "Middleware for Reactive Components: An Integrated Use of Context, Roles and Event Based Coordination". IFIP/ACM International Conf on Distributed Systems Platforms, Middleware 01. (LNCS Vol. 2218): p. 77-98. 2001

[25] P.-G. Raverdy and R. Lea. "DART: A distributed adaptive run-time". in IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98). 1998

[26] B. Redmond and V. Cahill, "Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java", in Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object Oriented Programming (ECOOP),: Cannes, France. 2000

[27] B. Redmond and V. Cahill, "Supporting Unanticipated Dynamic Adaptation of Application Behaviour". ECOOP '02. 2002

[28] B. Robben, et al. "Non-Functional Policies". in Proceedings of the Second International Conference on Metalevel Architectures and Reflection. Saint-Malo, France: Springer-Verlag. 1999

[29] Sandia_National_Laboratories, "Jess, the Rule Engine for the Java Platform (http://herzberg.ca.sandia.gov/jess/)", 2003

[30] T. Schäfer, "Supporting Metatypes in a compiled, reflective programming language", PhD thesis, Dept. of Computer Science, Trinity College Dublin, Dublin, 131. 2001

[31] Sun_Microsystems_Inc., "Java 2 Platform, Standard Edition (J2SE)". http://java.sun.com/j2se/. 2002

[32] T. Wall, "Mobility and Java RMI", MSc, Computer Science, Trinity College Dublin, Dublin, 70. 2000

[33] Y. Yokote, "The Apertos reflective operating system: The concept and its implementation". Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). 27(10): p. 414-434. 1992