Completely Unanticipated Dynamic Adaptation of Software

A thesis submitted to the
University of Dublin, Trinity College,
in fulfilment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

John Keeney

Distributed Systems Group,
Department of Computer Science,
Trinity College, University of Dublin.

October 2004

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

John Keeney

_____

May 2005

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

John Keeney

_____

May 2005

Acknowledgments

Firstly, I must thank my supervisor Prof. Vinny Cahill, who prised this thesis out of me with his subtle but effective prodding. Thanks for the support and guidance over the years while still letting me get on with my procrastination.

Thanks to Microsoft Research Ltd., for the happy days that comes from financial support, as Donal and myself ploughed the long furrow.

Thanks to the Coyote team, Tilman, Peter, Jim, and Barry who introduced me to the big bad world of reflection, and brought metatypes to us all. In particular Barry for Iguana/J, without which this thesis would not have been possible. In addition thanks to Mads for his help and guidance with ALICE, and for his useful and insightful comments to improve the thesis.

Thanks to the buckos in DSG for keeping me idle all these years, while I was keeping them idle too. I bet you guys won't read the rest of this thesis!

Thanks to the Girlies (that includes Owen and Paul) for the sanity.

Special thanks to my family for me, for the support, and the instilled desire to achieve.

Most of all, thanks to Niamh for the "minds". I promise I'll come home and shave now.

Summary

Dynamic adaptation of software behaviour refers to the act of changing the behaviour of some part of a software system as it executes, without stopping or restarting it. It is difficult to dynamically adapt software if the need for adaptation arises while the software is executing, and especially so if the program is compiled and the source code is unavailable. Ideally, it would be possible for adaptations to be applied to a running application without any anticipation of the adaptation itself, preparation of the location for that adaptation, or even anticipation of the need for some adaptation. Even with the best planning and foresight it is virtually impossible to anticipate at design and production stages all of the dynamic behaviour adaptations that may be required for a piece software, especially if the need for adaptation is triggered by unpredictable and erratic changes in the operating context, the application's resources and demands, and the users' requirements.

The need for dynamic adaptation arises in various circumstances, from the very simple desire to dynamically customise a piece of software to suit current needs, through to a necessity to continually evolve a long-running program as its requirements and operating context change. These adaptations may simply involve pre- or post-processing of operations, for example, to support consistency checking, through to dynamically adapting the core behaviours of an application as its operating context or requirements change, for example to support dynamic upgrading or repair of the system. While it may be necessary to adapt the core functional behaviours of an application, it may also be necessary to change or insert new non-functional behaviours that do not change what the software does, but rather how it does it. Examples here include dynamically inserting debugging or tracing statements, through to making some object in an application persistent or remotely accessible. To perform these changes it should not be necessary to restart the application, or indeed have access to the source code of the application since the core problem domain being modelled by the application has not changed.

If dynamic adaptation is to be completely unanticipated, the management and control of the adaptation process must also be dynamically adaptable. It is unrealistic to expect an adaptation framework using a hard-coded, static, or inflexible approach to adaptation management, to perform adequately in a generalised manner. Only by decoupling the adaptation mechanism from the adaptation control, and dynamically specifying and adapting the adaptation control strategies, can completely unforeseen dynamic adaptation of running software become a realistic goal. This thesis provides an in depth discussion of unanticipated dynamic adaptation, introduces the term “completely unanticipated dynamic adaptation” to refer to adaptations where all properties of the adaptation can remain unanticipated until during runtime, and identifies the set of requirements that must be met to achieve this.

This thesis presents the Chisel adaptation framework, and demonstrates that a general-purpose, context-aware dynamic adaptation framework is achievable. This system can be used to perform almost any unforeseen behavioural adaptation without stopping the application, and without changing the application itself. In this system a human-readable, dynamically updatable policy script was chosen as the favoured approach to drive the adaptation mechanism in a responsive manner by monitoring changes in the user, application, and environmental context. The Chisel framework also demonstrates that behavioural reflection, using the managed but unforeseen dynamic selection of Iguana/J metatypes, is a valid and powerful technique for completely unanticipated dynamic software adaptation. In addition, the Chisel framework provides a structured mechanism to allow a user to inspect and probe the internal operation of compiled software without access to the software's source code to allow that software to be adapted or extended as appropriate.

To evaluate Chisel and validate our claims, a number of examples and case studies are used, including the use of the Chisel framework to dynamically adapt an off the shelf network application, as it ran, to use ALICE, a middleware for mobile computing environments, and how, using an Iguana/J metatype to implement a snap-on non-functional behaviour to implement a naming mechanism for individual objects, those named objects can be individually adapted or queried as context sources.

Contents

List of Figures

List of Tables

"In the struggle for survival, the fittest win out at the expense of their rivals because they succeed in adapting themselves best to their environment."

Charles Darwin (1809 - 1882), *The Origin of Species, 1859*

Chapter 1
INTRODUCTION

This thesis is concerned with the study of unanticipated dynamic adaptation of software. In particular, it is concerned with completely unanticipated dynamic adaptation, whereby no part of the applied modification has been anticipated until after the software to be adapted has started executing. This thesis focuses on how this adaptation process can be controlled and managed at runtime.

Chisel is a software adaptation framework that allows completely unanticipated dynamic adaptation. It supports the application of unanticipated adaptations, which perform unanticipated actions, in unanticipated locations, at arbitrary times during execution, and all in a controlled and managed manner. The managed application of these adaptations is accomplished via an interpreted adaptation policy script, written in a human readable declarative language. This policy script can be changed dynamically to support requirements unanticipated even during runtime. The adaptations are implemented as metatypes [125, 139], an abstraction to describe "snap-on" software behaviours, which are effected using runtime behavioural reflection.

This Chapter begins by introducing the problem of unanticipated dynamic software adaptation. It continues by laying out the requirements that must be satisfied before a system can be specified as supporting completely unanticipated dynamic adaptation. The Chapter then introduces some concepts that are applied in this thesis, including behavioural reflection, the use of metatypes as a dynamic adaptation technique, and policy-based adaptation management. The Chisel framework is then introduced as a proposed solution to the problem of unanticipated dynamic adaptation, and one that fulfils the identified requirements. Finally, a roadmap for the remainder of this thesis is introduced.

1.1 Aims and objectives

This thesis aims to tackle the problematic requirement to anticipate software adaptations, even before the need for those adaptations becomes apparent. The objective of this thesis is to provide a structured way to perform managed adaptations of software, as the software runs, in a manner such that no aspects of any particular adaptation needs to have been anticipated prior to the requirement to perform those adaptations. The need to adapt software, especially at run time, usually only arises in response to a limited set of motivations. These may arise from the need to update or evolve a long-running system, the needs or resources of the user or higher-level software having changed, the needs or resources of the operating environment or supporting software having changed, or the desire to probe or debug the software in an experimental environment.

The ability to adapt a software module dynamically makes that module flexible and extensible. Software made up of these adaptable modules is then more malleable for use in unanticipated environments, and can support unanticipated use cases. The need to perform these adaptations dynamically is generally not present in stable software that executes in a stable and closed environment, where changes can be anticipated, and unanticipated requirements for changes are few. In this case, if changes are required, these changes can be applied off-line in an easy to control manner. However, in an environment where these factors do not apply it may be necessary to change the software as it runs, because the requirements, state, or resources of the application, operating environment, or user may have changed in a manner that requires immediate adaptation.

A mobile computing environment provides a prime scenario to show that this static model is not sufficient. In a mobile computing environment, the requirements and resources of the operating environment can change drastically, in an erratic and completely unanticipated fashion. In a mobile-computing environment, resources are usually limited, but it may be possible to dynamically add to or remove these resources on the fly. For example, a mobile computer may struggle with a very limited data connection, when without warning its user may hot-plug a network adapter, capable of massive bandwidth and very low latency. The user may just as suddenly remove this resource while it is being used. Standard adaptation models, e.g., the state and strategy patterns [55], whereby adaptations must be anticipated and embedded in the software's source code, cannot cope with such an environment. It is absolutely impossible to anticipate all possible adaptations to apply in a mobile computing environment. For this reason this thesis applies its results to the mobile computing

environment as a case study, but the reasoning remains the same for all adaptation scenarios where adaptations cannot be anticipated.

As a secondary objective, this thesis also aims to prove the capabilities and explore the limitations of metatypes for dynamic adaptation of running software. It is intended to show that this framework can use runtime behavioural reflection to adapt arbitrarily compiled software, even in ways that had not been foreseen at design time, compile time, or even after the application started executing. The Chisel policy-based management mechanism demonstrates how the unanticipated application of these adaptations can be dynamically managed in either a reactive or proactive manner. This is achieved using a high-level, user-readable, declarative policy script that can be dynamically updated, thereby allowing completely unforeseen personalisation, extension, or just examination of any arbitrary code.

1.2 Completely unanticipated dynamic software adaptation

This section provides some background into the area of software adaptation, and the varying degrees to which particular adaptations must be anticipated. This section concludes by introducing the concept of completely unanticipated dynamic adaptation.

1.2.1 Software adaptation and evolution

Before addressing the degrees to which software adaptation must be anticipated, software adaptation methods and software adaptations themselves must be discussed. "Adaptable systems" are defined as dynamically configurable systems, whereby the systems' configurations can be changed over time at runtime, whereas "adaptive systems" (called "self-adaptive systems" in [113]) are dynamically configurable systems that are responsible for changing their own configuration or behaviour during runtime [31]. Adaptive systems must decide how and when to execute reconfiguration operations on themselves. If the decision about when to perform adaptations are made by the systems themselves, it is not possible to adapt at runtime in an unforeseen manner because of the lack of a-priori knowledge about unanticipated changes. If the decision to adapt and the adaptation itself come from outside the system (adaptable system) it must be possible to modify at runtime the adaptation management logic, thereby handling unforeseen adaptation requirements [45]. A system can be both a self-adaptive system and an adaptable system if it performs self-adaptation but this adaptation process can be altered, or new individual adaptations can

be added at runtime, by some external source, for example, the system user [45]. Dynamically configurable systems can be either behaviourally closed dynamic systems, whereby all behaviour changes are built into the system, or behaviourally open dynamic systems, whereby new behavioural changes can be added [112]. If a system is to provide a mechanism to support the dynamic application of unanticipated adaptations, in response to unanticipated requirements at runtime, it must be a behaviourally open adaptable system, with an interface to support external changes to the adaptation logic.

A proposed taxonomy is introduced in [18], describing software adaptation tools and techniques based on four categorising features, *when*, *where*, *what*, and *how*. "*When*" describes the temporal properties of the changes supported, including when the adaptations are applied. "*Where*" describes the location where adaptations can be applied, which parts of the target system are changed, and what mechanisms are needed to apply the adaptations at those locations. "*What*" describes the types of adaptations that are supported and what attributes of the target system are adapted as the adaptation is applied. "*How*" includes a description of the control and management support for the adaptations.

This thesis will use the *when*, *where*, *what*, and *how* categorisation above, not just to describe adaptation tools and methods, but to describe individual adaptations or modifications themselves. If a particular adaptation is to be applied to a software system, that adaptation can be described in terms of when it is designed and when it is applied, where it is applied and the mechanism used, what is the nature of the adaptation and what aspect of the system is changed, and the management support and control mechanisms that handle and guide the application of that individual adaptation.

For example, a static adaptation of adding a new behaviour to an application by changing the source code of an application and recompiling it, can be classified according to the *when*, *where*, *what*, and *how* categorisation. *When* the adaptation is applied is given as design time, or pre-compile time. *Where* the adaptation is applied is defined by where in the source code the change was made and what was affected. *What* the adaptation does is defined by the change that was made by the new source code. *How* the change is managed is described by the control logic embedded in the application that controls the use of that newly added behaviour, and by stating that it was the programmer who made the logical decision to change the code to add the new behaviour in response to some requirement.

1.2.2 Anticipation of the adaptation's attributes

If a particular software adaptation can be described in terms of the four properties above, then the degree to which that adaptation is anticipated refers to the anticipation of the nature of the adaptation (*what*), the anticipation of the location of the adaptation (*where*), the anticipation of when the adaptation is written and applied (*when*), and the anticipation of what control and supports manage the application of the adaptation (*how*), as described in table 1.1. To date it has proved difficult to define what exactly "*unanticipated*" refers to in terms of software adaptation, reconfiguration, or evolution, since all adaptations must be anticipated at some time, and conversely, all adaptations must remain unanticipated until some point [87]. A popular understanding of "*unanticipated*" is that which has not been foreseen at design time [18, 99]. "*Unanticipated*" software adaptation can also be understood to mean software adaptations that are not anticipated until the execution of that software is started [125]. Anticipation of adaptation can also be linked to preparation for adaptation by stating that a change is unanticipated if that change is "possible without [the] encoding of proper hooks in prior versions of the changed software" [87]. Anticipated software change is caused by expected changes in requirements, so the adaptation can be considered and prepared before it is needed, whereas, unanticipated software change is caused by unexpected requirements change, so it is impossible to predict or prepare the adaptation or its location before the requirement for the change occurs [99]. Instead of settling for one definition of anticipation, this thesis will discuss the degrees to which the four characteristics of individual adaptations described above can each remain unanticipated until different stages in the lifetime of a software entity. The different stages discussed are: the design and production stages, compile time, start of runtime, load time, and during runtime.

- When must the contents of the particular adaptation be anticipated or prepared? (*anticipation of what the adaptation does*)
- When must the timing of the application of the particular adaptation be anticipated or stated? (*anticipation of when the adaptation is applied*)
- When must the location at which the particular adaptation is applied be anticipated or prepared? (*anticipation of where the adaptation is applied*)
- When must the management of the application of the particular adaptation be anticipated or prepared? (*anticipation of how the application of the adaptation is managed*)

Table 1.1: Categorisation of anticipation of an individual adaptation

While the categorisation given in table 1.1 above was chosen because of the relative independence from each other, there may appear to be some overlap between the different categories of anticipation. For example, when an adaptation is applied is generally dependent on the contents of adaptation control directives, but the separation between "how" and "when" becomes clear when considering when the control directives must be anticipated or prepared and the need to anticipate exactly when an adaptation is applied. In addition, if the control directives are specified in a manner to be dependent on some combination of unanticipated context, then the time at which that adaptation is applied is itself unanticipated. In a similar manner, the location at which an adaptation is applied can be tightly bound to what the adaptation does. This is particularly the case where the adaptation is prepared to address a particular functional requirement and so must be positioned where that requirement can be addressed. However, in the case of a general-purpose or non-functional adaptation, what the adaptation does and where it is applied may be independent, and so the need to anticipate what adaptation does should be independent of any anticipation of where that adaptation will eventually be applied.

Adaptation anticipated at design and production stage

Adaptation anticipation at design and production stages occurs if the adaptation, the location of the adaptation, or its timing or control mechanism is prepared as the code of the software is designed and written. Here the adaptation or some preparation for that adaptation is embedded directly in the application design. For example, if the location for a possible adaptation must be prepared, even if the adaptation that will be applied at that location can remain unanticipated, or, a system anticipates a number of adaptations or strategies, but the selection of which adaptation will be applied is left until some later stage, then these systems do not support unanticipated adaptation. Examples include the State or Strategy design patterns [55], where the ability to dynamically adapt the software, and the adaptation strategies themselves, are designed into the application software.

Adaptation anticipated at compile-time

The requirement for some aspect of an adaptation to be anticipated at compile-time arises where access to the source code of the software is required. This arises where the already written source code must be viewed or recompiled either to include the adaptation, or to support the adaptation. Examples here include systems that pre-process source code or require language extensions to be used. Several adaptation systems, as described later in Chapter 2, incorporate some form of adaptation anticipation at this stage. Example systems include IguanaC++ [139], AspectJ [162], and Correlate [73, 74, 128, 129, 154].

Adaptation anticipated at the start of runtime

If some aspect of an adaptation remains unanticipated until just before runtime, that property of the adaptation can be specified at the start of program execution. For example, this is the case where the behaviour of an application can be adapted to individual circumstances each time it is executed, for example as seen with applications that use command-line options. Examples of systems that require anticipation of aspects of adaptations at this stage include DynamicTAO [90, 91, 132], static metatype association in Iguana/J [125-127], and Kava [159, 161].

In this situation, unless the adaptation aspects are specified for later use by a runtime adaptation manager, most of the properties of the adaptation will necessarily have been anticipated and prepared at an earlier stage and have already been bound into the application source code or binary structure.

Adaptation anticipated at load-time

Anticipation of adaptation at load-time refers to where the adaptation, its location, or its timing or control mechanisms are prepared as some part of the program is loaded into a runtime system. In these cases, it may be the location of the adaptation is prepared by adding hooks for later binding, or the adaptation itself is statically bound into the program code. In some cases the application to be adapted may already be executing for some time, as with delayed class-loading in Java or explicit loading of dynamically linked libraries (DLLs), but load-time may also occur just as the application starts, as happens with some system classes in Java or with implicitly loaded DLLs. The adaptation binding mechanism may also operate at this time as seen with customised classloaders as used in Kava [159, 161] and Javassist [24, 26], dynamic linking of DLLs as used in K-Components [44, 45], or by making use of the JIT compiler interface as seen in Iguana/J [125-127].

Typically, adaptations, or aspects of the adaptations, that are applied at this stage are of a static nature, as seen in Kava [159, 161] and Javassist [24, 26]. This arises because once loading of a software module has occurred; it is difficult to unload or reload that module without consistency and state maintenance issues becoming a major concern. To do so would require that all links or handles to the loaded module be monitored so that the module can only be reloaded when no handles exist, or handles or links to the old module are changed to those of the newly loaded module in a managed and consistent manner.

Adaptation anticipation during execution

In this case, some aspect of the adaptation to be performed remains unanticipated until the target software is running. Many systems that support dynamic adaptation, and all systems supporting unanticipated dynamic adaptation, leave some property of the adaptation unanticipated until during the execution of the application to be adapted. As can be seen from the adaptation systems described in Chapter 2, all systems that support unanticipated dynamic adaptation in this manner must have some runtime controlling mechanism or metalevel manager linked into the application or its execution environment. This is necessary to support the unanticipated characteristics of the adaptation. Typically the unanticipated characteristics of the adaptation must be specified from outside the application, since if it was specified from within the application, it would necessarily have been anticipated and have support for that aspect of the adaptation already built into the application.

Summary of anticipation of an adaptation's characteristics

This thesis will use the categorisations described above to classify the degrees to which individual adaptations are anticipated. Table 1.2 below provides a summary of the degrees to which the location that individual adaptations are applied must be anticipated.

Design / production time	Hooks for the particular adaptation must be designed into the application at particular locations.
Compile / link time	Hooks to support adaptations at particular locations are added by processing the application source code at or before compile time.
At or before start of execution	Before the application is executed, hooks are inserted into the application at particular locations at a binary level to support adaptations.
Loadtime	As a module is loading into the executable system, hooks are inserted at set locations to support adaptations.
During runtime	Hooks are added dynamically at runtime so the locations of possible hooks need not be specified beforehand, or hooks have already been added at a very large number of locations.

Table 1.2: Anticipation of the locations where particular adaptations will be applied

Table 1.3 below outlines how the timing of the adaptation's application can be anticipated.

Design / production time	Code defining the timing of when an adaptation is applied must be programmed into the application at the production stage.
Compile / link time	Source code is required or reworked to define when an adaptation will be applied.
At or before start of execution	When the adaptation is applied must be specified before or at runtime, perhaps by changing the application executable itself, or as a command line option.
Loadtime	When the adaptation is applied must be specified as the target module is loaded into the executable system.
During runtime	An adaptation can be applied at any time as the target application is executing.

Table 1.3: Anticipation of when particular adaptations will be applied

Table 1.4 follows with a summary of how the control logic that is used to manage the application of an adaptation can be anticipated.

Design / production time	The logic controlling the application of particular adaptations is designed into the target application itself.
Compile / link time	The target application's source code is required to manage the application of an adaptation, or the source code is preprocessed to contain the adaptation's control logic at compile time.
At or before start of execution	The control logic of how the adaptation is applied must be specified before or at runtime, perhaps using a static initialisation script or command line option.
Loadtime	The control logic of how the adaptation is applied must be specified before or as the target module is loaded into the executable system, perhaps by weaving adaptation logic into the target module itself.
During runtime	Adaptation management directives can be specified at any time as the target application is executing.

Table 1.4: Anticipation of the control logic managing how a particular adaptation is applied

A synopsis of when the contents of an adaptation must be anticipated or prepared is given in table 1.5 below.

Design / production time	The adaptation must be hard-coded into the target application.
Compile / link time	The adaptation is statically linked into the target application, or the source code of the adaptation target is required to write the adaptation module.
At or before start of execution	The adaptation must be specified before the target application begins executing, perhaps because the execution system cannot support adding new modules at runtime.
Loadtime	The adaptation must be specified before or as the target module is loaded into the execution system, perhaps because the adaptation is statically bound to the target module and new adaptations cannot be added.
During runtime	New adaptations can be added and bound at anytime as the target application is executing.

Table 1.5: Anticipation of what a particular adaptation will do

1.2.3 Completely unanticipated dynamic adaptation

The thesis argues that an individual adaptation is completely unanticipated only if the nature of what the adaptation does (*what*), the location where the adaptation is applied (*where*), the specification of when the adaptation is applied (*when*), and what control logic manages the application of the adaptation (*how*) can all remain unanticipated and unprepared until after the system being adapted has started executing. We introduce the term "*completely unanticipated dynamic adaptation*" to refer to dynamic software adaptation where all of these aspects of each adaptation to be applied can remain unanticipated until during the execution of the software to be adapted.

But completely unanticipated dynamic adaptation must itself be anticipated

Since all aspects of completely unanticipated dynamic adaptations remain unanticipated until during execution, the execution system will need some form of runtime support to

coordinate the application of these individual unanticipated adaptations. Only the methodology and tools to support individual completely unanticipated dynamic adaptations must be present, so only the *need* to support completely unanticipated dynamic adaptation at runtime must be anticipated at or before runtime. Therefore, this thesis argues that requiring a runtime adaptation controller, which may be combined or associated with the application at or before runtime, does not break the requirement of completely unanticipated dynamic adaptation, specifically that all aspects of the individual adaptations can remain unanticipated until during execution.

1.3 Motivation

Section 1.2 has discussed completely unanticipated dynamic adaptations in detail in terms of how a dynamic adaptation can be termed "completely unanticipated" if the location of the adaptation, what the adaptation does, where the adaptation is to be applied, and when the adaptation is applied, can all remain unanticipated until after the target software has started execution. However, the reasons why a user would want to apply such an adaptation have not been discussed. The primary motivation for completely unanticipated dynamic adaptations is where the need for an adaptation does not become apparent until after the target software has started execution, but stopping the software for repair or extension, and then restarting it, is undesirable or inappropriate. In these circumstances, all of the information about what a particular adaptation is required for, where it is required, how or when it should be applied, simply cannot be known or anticipated before runtime, or before the adaptation is needed.

Completely unanticipated dynamic adaptations are particularly useful where adaptations are required as hotfixes for long running applications, or in circumstances where high availability requirements dictate that the software must be adapted but restarting the application or applying adaptation other than during runtime is not appropriate. This may be the case where a running application has accumulated dynamic state that would be difficult to restore if the application was stopped for adaptation and then restarted. This is also the case for applications with difficult startup or shutdown procedures or would require restarting other software depending on the target software. Unanticipated dynamic adaptation can also be used for dynamic error recovery where shutdown, fix, and restart would be unavoidable.

In an environment with frequent context changes it may also be inappropriate to stop and restart the application, since this could be a frequent or erratic occurrence, so the adaptations must be applied dynamically. In such an environment it is likely that the required adaptations may be unanticipated as the need for the adaptation may be unanticipated. This is particularly true for mobile computing where the user's, application's and environment's state, context, and requirements can all change in unanticipated ways, to values which could not have been foreseen prior to execution.

This leads on to the motivation that that an unprepared piece of software could be made context-aware. While this would be beneficial for software that is already dynamically adaptable, this would be particularly beneficial for software where both dynamic adaptation and the context-aware triggering of those adaptations were unanticipated prior to runtime, thereby adding flexibility and value to that software. Such unanticipated dynamic adaptations could also be valuable to user by allowing the software to be dynamically customised in an unanticipated manner, thereby empowering the user.

While the need to inspect and probe a piece of running software is itself a key requirement for unanticipated dynamic adaptation, such probing and profiling behaviours can themselves be considered adaptations. Ideally such probing and profiling of the software should not actually disrupt the software being probed since that would affect the fidelity of the observations, but the application of such probing or profiling mechanisms to an unprepared piece of software is indeed a change to the designed behaviour of that software as a whole. If the profiling behaviours are treated as unanticipated adaptations, to be dynamically applied and removed in an unanticipated manner, then different parts of the software can be probed as the need arises. This would be useful for debugging, monitoring or auditing, or testing. It could be used during the development process of the software where testing, profiling, debugging, and fixing is performed in an iterative manner. It could also be used at runtime to locate bottlenecks or hotspots that require streamlining, particularly where accumulated state is of importance.

While completely unanticipated dynamic adaptation could be used to circumvent security measures built into the software, as mentioned in section 1.9 later in this chapter, completely unanticipated dynamic adaptations could also be used by an administrator to audit or control access to the software. For example, this could be achieved by dynamically allowing or disallowing certain method calls to certain users, or, checking the value of parameters or other contextual state before allowing an operation to proceed.

1.4 Dynamic adaptation using metatypes

A reflective computational system is one that reasons about its own computation [144]. This is achieved by the system maintaining a representation (metadata) of itself that is causally connected to its own operation, so that if the system changes its representation of itself, the system adapts [95]. With behavioural reflection in an object-oriented system [52], the reflective system reasons about and adapts its own behaviour by associating meta objects with the objects in the application. In a reflective system, the communications between the meta objects and base objects take place through a set of well-defined interfaces, referred to as that system's meta object protocol (MOP) [83]. The key advantage of runtime behavioural reflection is that this introspection and adaptation of the base-level behaviour can be performed dynamically at runtime.

1.4.1 What is a metatype

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. Object metatypes describe behaviours that are not directly related to the part of the core application domain being modelled by that object. Schäfer [139] introduced the concept of a metatype as a characterisation of an object's object model, and as such its non-functional behaviour. Redmond [125] defines a metatype by stating that "the metatype of a class or object represents some coherent internal behaviour change from its original source code behaviour", i.e., a behavioural change applied to the class or object. 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. 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.

Examples of metatypes include: verbosity or tracing, remote accessibility, persistence, fault tolerance, debugging, or optimisation. Adding persistence behaviour to an object, without regard to the object's data, interface or behaviour, does not change its type, but rather its metatype.

A metatype may change the functional behaviour of the base object with which it is associated, or may provide an additional non-functional behaviour that is associated with an object. To achieve this, the metatype may need to access application and type information about the object, or its operating environment, from the running system. For example, to

profile the operation of an object, its associated metatype will need to be informed when that object is accessed. It is preferable that this access happens transparently to the objects in the system, so the objects' class can be written in a manner completely unaware of any metatype that may be applied to it. Making use of a runtime reflective model, using a meta object protocol (MOP) is a prime candidate for that interface [139]. However, a runtime reflective model is not the only mechanism that could be used to implement metatypes. Chapter 2 describes a number of adaptive techniques that could be leveraged to implement the metatype model.

This thesis aims to discuss and evaluate how the association of new metatypes to application objects and classes can be exploited to adapt the behaviours of those classes and objects. In particular, this thesis will focus on the dynamic association of metatype with classes and objects in order to perform dynamic adaptation. This thesis will also demonstrate how these dynamic adaptations can be applied in a completely unanticipated manner, as requirements for these unanticipated adaptations become apparent during the execution of an application.

1.5 Policy-based management of adaptations

Many traditional adaptable systems 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 are 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. Policy specifications maintain a very clean separation of concerns between the adaptations available, the adaptation mechanism itself, and the decision process that determines when these adaptations are performed.

Policy specification documents are usually persistent text-based declarative representations of policy rules that ideally can be read, understood and generated by users, programmers and applications. A policy rule is defined as a rule governing the choices in behaviour of a managed system [35]. 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, in the form of a positive obligation policy rule, is usually made up of an event specification that triggers 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 object that is part of the managed system upon which that action is performed [35]. Many policies will also contain some restrictions or guards confining the rule action to appropriate occasions. This *event-condition-action* (ECA) format is standard for rule-based adaptation systems [33-35, 44, 45, 47, 67, 68, 78, 79, 97, 98, 104, 120, 143]. A policy-based adaptation management system is usually responsible for monitoring these events, evaluating the conditions and initiating the management action on the targeted managed object. In a policy-based dynamic adaptation system it should be possible to edit the rule set and have them re-interpreted to support the dynamic addition of new rules or changes in policy.

A policy-based adaptation management model, with dynamically updateable policy specifications, was seen as an ideal mechanism to drive a general-purpose dynamic adaptation framework. The use of policy-based management in the Chisel project is introduced below, and further detailed in Chapters 3 and 4.

1.6 The Chisel adaptation framework

Chisel is an adaptation framework that supports completely unanticipated dynamic adaptation of compiled Java applications. At runtime, an adaptation manager performs dynamic adaptations, implemented as dynamic metatype associations. This adaptation process is controlled by an interpreted, dynamically updateable, declarative policy script, which contains arbitrary rules about how the system should adapt. Controlled by these adaptation rules, adaptations can be applied proactively, or triggered in response to context changes. Here context refers to the state, resources, and requirements of the operating environment, the application, and the user. These changes, observable as events, are intercepted by the adaptation manager and used to drive the adaptation process. The policy script, which contains both proactive adaptation directives and arbitrarily complex reactive rules, supports the completely unanticipated specification of control directives to manage the application of dynamic adaptations to named objects, interfaces and classes, all without access to the application source code. The design and implementation to the Chisel dynamic adaptation framework is described in detail in the following chapters of this thesis.

1.7 General-purpose dynamic adaptation support

This thesis is focussed towards the study of unanticipated dynamic software adaptation in a "general-purpose" manner. The term "general-purpose" may be interpreted in numerous ways, but for this thesis it refers to adaptations are not limited in terms of applicability or function. Support for general-purpose adaptations can be discussed in terms of a number of dimensions: general-purpose application domains; general-purpose problem domains; independence from specific languages or runtime environments; and an absence of dependencies to either create, manage, or execute the adaptation. With respect to these discussion categories, adaptation for general-purpose application domains refers to adaptations that can be applied to a broad category of applications performing different tasks. Adaptation for general-purpose problem domains refers to adaptations that are not limited to solving a certain type of problem. General-purpose adaptations may be independent of the language or runtime environment for which the target software was produced, i.e., the adaptations can be written in different languages or operate in different runtime environments. To maintain generality, general-purpose adaptations should also have a small set of dependencies required by the adaptation mechanism or the adaptations themselves, whether at compile time, load time, or at the execution stage. These dependencies may refer to other software or systems required to create, support, manage, or execute the adaptation that is to be applied.

This thesis is particularly focused towards generality in terms of arbitrary problem domains and application domains. The previous section has introduced a large set of problems for different applications that could be at least partially solved by support for completely unanticipated dynamic adaptation in a context-aware manner. The Chisel framework, although implemented in Java, and dependent on the Iguana/J framework as a dynamic adaptation mechanism, was designed to be language independent where possible, and independent of the adaptation mechanism used.

Completely unanticipated dynamic adaptation can only be accomplished if the mechanisms supporting the specification of what the adaptations do, where they are applied, how and when they are applied, are kept separate for the mechanisms that applies the adaptations. This means that the framework must be independent of both the particular adaptations and the particular target applications. Therefore, by providing a framework that supports completely unanticipated dynamic adaptations, it can be said that such a framework is a

general-purpose adaptation framework in terms of generality of application and problem domains. Assuming a flexible adaptation mechanism is used, the generality of the framework will only be limited by: the types of adaptations possible; the set of locations that can be identified or created and at which an unanticipated adaptation can be applied; the ability to specify how and when the adaptations should be applied; and the set of restrictions inherent in the adaptation mechanism itself.

Chapters 3 and 4 of this thesis discusses the requirements that an adaptation framework must support in order to achieve support for completely unanticipated dynamic adaptations, and how meeting these requirements will affect the generality of the framework. Section 3.2 in particular discusses the use of Iguana/J and the metatype model in the Chisel framework, and discusses the consequential loss of generality in terms of the dependence on Iguana/J, the types of adaptations supported, and language and environment dependencies.

1.8 Contributions

This thesis provides an in depth study into the area of unanticipated dynamic adaptation of compiled software. This study introduces the term "completely unanticipated dynamic adaptation" and clearly defines the set of requirements that must be met before an adaptation framework can support completely unanticipated dynamic adaptation. A detailed discussion is also provided to discuss current techniques for and research into unanticipated dynamic adaptation, specifically with respect to their support for completely unanticipated dynamic adaptation.

As can be seen from the state of the art review in Chapter 2, there remains a lack of generalised support for completely unanticipated dynamic adaptation of arbitrary compiled software. A general-purpose solution, the Chisel framework, is provided, supporting completely unanticipated dynamic adaptation of general-purpose Java applications. An event-based mechanism to support context-aware adaptation is also provided. A declarative, human readable policy language is presented, which can used to create rules to dynamically drive the adaptation process. Using this policy language, specification of arbitrarily complex adaptation rules and event manipulation directives is supported, allowing unprecedented dynamic control and manipulation of software. Along with the policy script model, a programmatic interface is also provided. By use of the Chisel policy language and the Chisel programming interface dynamic and unanticipated contextual information can be exploited to both drive and constrain this adaptation process.

With Chisel comes the ability to inspect, profile, and debug the operation of any Java component as it runs, in a completely unanticipated manner even after the application has started executing. When combined with the ability to adapt the behaviour of almost any Java component using dynamic metatype association in an intelligent and reconfigurable manner, it is now possible, effective, and practical to dynamically apply unanticipated adaptations to any compiled Java software, without interrupting the application and without access to its source code. By using the Chisel framework the internal structure of third-party software can be examined, profiled, debugged, and adapted. The Chisel framework further challenges the "black-box" approach to software development by supporting the "open" manipulation of arbitrary software modules in a general-purpose manner. Such inspection and manipulation allows the combination, reuse, and tailoring of diverse software modules in ways unforeseen by their original designers.

This thesis continues current research on metatypes, and demonstrates that the dynamic association of metatypes with base-level objects, classes and interfaces is an effective method for performing dynamic adaptation, both to change the application's functional behaviours, and to add new non-functional behaviours. This thesis demonstrates that runtime behavioural reflection is an impressive and powerful technique for software adaptation using dynamic metatype association. A number of powerful uses of this technique are presented throughout the thesis, but especially the re-implementation of the ALICE middleware framework [7, 62, 63, 156] as a metatype. The ALICE middleware framework was used to add support for intermittent network disruption to an off-the-shelf application, allowing it to be used without modification in a mobile computing environment. As a further case study, an object naming mechanism is also implemented using metatypes, allowing the individual objects to be used as part of policy rules, both as targeted managed objects, and within the conditions block of any policy statement.

1.9 Orthogonal research topic

One of the major issues with unanticipated dynamic adaptation, composition or swapping of software components is the lack of evaluation of the consequences of such actions. In this thesis, no attempt is made to discuss the stability of the adapted software after the adaptation has been applied. The Chisel framework is a powerful enabling technology, but like almost all enabling technologies, it can be used to disable its target, either by imprudent use or in a malicious manner. This remains an active and challenging research topic, but was considered to be outside the scope of this research.

1.10 Thesis roadmap

The remainder of this thesis is organised as follows. Chapter 2 presents an in-depth state of the art review of dynamic software adaptation, classifying primarily the degrees of anticipation required by a representative sample of related research on software adaptation. The design of the Chisel dynamic adaptation is laid out in detail in Chapter 3, describing the key components of the framework. Chapter 4 describes a prototype implementation of the Chisel framework. Chapter 5 evaluates the Chisel framework by describing an in-depth case-study of the use of the Chisel framework to incorporate the ALICE middleware framework into a third party off-the-shelf application, adding support for intermittent network disconnection, thereby allowing it to be used successfully in a mobile-computing environment. Chapter 5 also describes how the Chisel framework was used to implement an extensive logging and profiling behaviour, and how this behaviour is used to support the dynamic naming of individual application objects in order that they can be dynamically adapted or probed. Finally, Chapter 6 provides some conclusions, summarises the contributions drawn from this research, and provides a list of interesting open issues that require further research.

Chapter 2
RELATED WORK ON ADAPTABLE SYSTEMS

This chapter provides some background on the research areas with which this thesis is concerned. The following sections analyse the most influential research in the area of dynamic software adaptation and adaptation management. Key areas of interest include dynamically adaptable software, unanticipated adaptation, reflective systems, and policy-based management of adaptable software. The key influential systems are summarised with their relevance to this thesis discussed. Since the area of adaptable software is very wide ranging, the set of systems described is the set of most influential systems but is not an exhaustive list.

The research discussed in this chapter is analysed specifically with respect to its relevance to the aims and objectives of this thesis. In this chapter, adaptable systems are primarily classified in terms of their support for completely unanticipated dynamic adaptation, and if the adaptation mechanism is implemented in an application-specific or general-purpose manner. Since this thesis is focused on the unanticipated dynamic adaptation of arbitrary software, this chapter also discusses support for adapting third-party compiled software without requiring access to the source code of that software. This chapter also discusses how the various adaptation mechanisms are managed and controlled, if the control logic can be dynamically specified, and the degree of separation between adaptation mechanisms and adaptation management. Since this thesis is not just concerned with unanticipated adaptation, but with unanticipated dynamic adaptation in particular, the dynamicity of the adaptation support provided is discussed in terms of when adaptations are bound into the target software and whether these adaptations can be removed or replaced.

A secondary objective of this thesis is to demonstrate the usefulness of metatypes as an adaptation technique. As previously discussed, a metatype is a behavioural change applied to a class or object. Metatype association allows an application to be modified by intercepting the operation of the application modules and redirecting those operations to an associated metatype, where those operations can be carried out under the control of the associated metatype. The associated metatype combines the desired behaviour of the application with the new behaviour embedded in the metatype, thereby adapting the application, without changing, replacing, or damaging any part of the application source code. Although introduced in the Iguana project as described later in this chapter, the metatype model can be implemented in many adaptation frameworks that support behaviour adaptation by association rather than replacement. The degree of support for the metatype model and adaptation by dynamic metatype association is discussed with respect to several of the adaptation mechanisms described in this chapter.

The systems covered in this chapter fall into a number of discussion categories: systems that must be discussed in terms of their seminal contributions to a particular area, systems that could be used as a basis for parts of the Chisel framework, and systems that offer functionality comparable to a part of or the entire Chisel framework. Each system will be discussed in terms of their relevance to one or more of these categories. The discussion categories used in this chapter are intended to form the basis of an informal descriptive framework rather than act as a formalised taxonomic framework. As a result, the topics discussed in this chapter are discussed to differing degrees, mainly in terms of their differences to related or similar work, or to the Chisel framework.

This chapter concludes with an overview of the systems described and a brief discussion on the open research questions that are tackled in this thesis.

2.1 Adaptation using reflective techniques

As discussed in Chapter 1, a reflective system is one that can inspect and adapt its own structure or operation. This section reviews the use of reflective techniques to adapt software. Of particular interest are systems that support some form of reification and adaptation of attributes of their operation or structure that affect the behaviour of the system, i.e., support behavioural reflection. This section includes a summary of relevant research on reflective systems and how it is related to or inspires the research described in this thesis. The Iguana architecture is discussed in terms of how it is used to form part of the Chisel

framework and how it influenced the design of Chisel. The Java HotSwap mechanism, Kava, and Javassist are discussed in terms of general purpose reflective adaptation tools and as background to other mechanisms which make use of them. DART, Guaraná, MetaXa, and K-Components are discussed as reflective tools that support dynamic adaptations where some parts of the adaptations can remain unanticipated. A number of other technologies could have been discussed in this section, e.g., the Byte Code Engineering Library (BCEL) [32], Open C++ [25], and OpenJava [152], but these systems would not have significantly contributed to the discussion of reflective techniques in terms of their differences to the systems that are discussed.

2.1.1 Iguana

The Iguana reflective programming architecture was developed at Trinity College Dublin. It is a reflective programming extension for object-oriented languages. It was introduced [59] as a method of incorporating meta object protocols into C++. It was later revised and simplified into Iguana/C++ [139]. Iguana/C++ was the first reflective framework to support dynamic metatype association in a compiled language (C++). Support for the unanticipated metatype association with Java was added in Iguana/J [126]. The Iguana model provides a mechanism to allow metatypes to be defined, implemented, and associated with objects without changing those objects' types. Most reflective OO programming languages and reflective systems have only one MOP and implementations of this MOP are used to change the object model of the resulting reflective systems. Iguana supports the definition of multiple MOPs. Iguana supplies the framework to allow the user to choose which parts of an object's object model to reify (see "reification categories" in [19, 59, 64, 125, 126, 139]). Each part of an object's object model that is reified is represented by a meta object, which is an instance of a meta object class.

Metatypes and Iguana

In Iguana, a MOP is the selection of which parts of the object model to reify, and the association of a meta object class with each of these reified parts. As discussed in Chapter 1, a metatype is a behaviour change that is applied to a class or object. In Iguana/C++ [139] and Iguana/J [126] metatypes are implemented using custom Iguana MOPs, i.e., by deciding which parts of the object model to reify, writing a set of meta object classes for these reifications to implement the new metatype behaviour, then associating that metatype implementation with an object or class. In the Iguana literature, the terms "metatype association" and "MOP selection" are similar and refer to this association of MOP

implementations to objects and classes. Iguana implementations supply the frameworks to instantiate these meta objects to reify the object model, and correctly order metatypes if more than one is selected. Another feature of Iguana/C++ and Iguana/J is the ability to have objects and classes select new metatypes at runtime, thereby changing the behaviour of the system, without changing the type of the objects or classes.

Iguana/J

Iguana/J [125-127] 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 at runtime.

The MOP is declared in a declaration file, which specifies which parts of the object model will be reified.

```
protocol Verbose {
  reify Creation: CreateVerbose();
  reify Execution: ExecuteVerbose();
}
```

Figure 2.1.1.1 Iguana/J: Example MOP declaration, the ProtocolVerbose MOP

Figure 2.1.1.1 above shows the declaration of a new metatype, *Verbose*. This metatype is implemented by reifying object creation and method execution with instances of meta object classes, *CreateVerbose* and *ExecuteVerbose* (See figure 2.1.1.2 below). This metatype is used to implement a verbose behaviour, that when associated with any object or class, will cause the creation and operation of that object or all objects of that class to be carried out in a verbose nature. This behavioural change (metatype) can be associated with any object or class, regardless of type, or any other metatype associated with the object or class, in a manner completely transparent to the object or class, and in a manner unanticipated by the class designer. Each meta object class is written in standard Java, with each class extending the default meta object class for the appropriate reification category.

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

Figure 2.1.1.2 Iguana/J: Example meta object class, the ExecuteVerbose class

The metatype declaration and the meta object classes are then passed to the Iguana/J compiler, which generates the metatype class that encapsulates the behaviour change. This metatype can then be associated with any object, class, or interface.

The association of a metatype and its associated meta object classes with any class or interface can be statically specified in a separate metatype association file, parsed at the start of execution of the application, as shown in figure 2.1.1.3, or dynamically from within base-level or meta-level source code, as shown in figure 2.1.1.4.

```
java.net.Socket ==> Verbose();
```

Figure 2.1.1.3 Iguana/J: Static MOP selection / metatype association

```
import ie.tcd.iguana.Meta;
java.net.Socket  mySocketObject = new java.net.Socket ();
Meta.associate ( mySocketObject, "Verbose", args);
```

Figure 2.1.1.4 Iguana/J: Dynamic MOP selection / metatype association

With the mechanism to statically initiate metatype association, Iguana/J maintains a high degree of separation between the adaptation mechanism and its target since there is no tangling of meta-level code and application-level code. However, this is not the case with dynamic metatype association since the code that performs the selection is hard-coded into the application. The ability to dynamically associate metatypes with an application's objects and classes 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.

The Iguana/J runtime component is implemented by making use of the JIT interface of the Java JVM (JVM version 1.3.1) [148]. This allows the Iguana/J runtime component to intercept class loading to insert hooks as the class is loaded. During execution, the Iguana/J runtime component makes use of these hooks in the class code to support metatype associations and the interception of reified operations for classes and objects. Iguana/J does not require any access to the source code of the adaptation target since support for metatype association occurs at load-time and runtime, and so metatypes can be associated with arbitrary third-party compiled classes and instances of those objects. The application does not need to be restarted or altered in any way since interception is checked every time the hooked operation is performed.

How metatypes and Iguana influence this research

This ability to use Iguana to dynamically associate metatypes with an application's objects and classes, thereby adapting that application without restarting or altering it in any way, provides a suitable mechanism to implement dynamic adaptation.

For any particular adaptation that is applied using Iguana/J, many aspects of the adaptation can remain unanticipated until after the target application has started executing. The design and coding of the metatype can be accomplished separately from the application since the meta object classes implementing the metatype, and the metatype itself, can remain unspecified until just before they are to be used at runtime. The statically initiated selection of metatypes, using the metatype association file, can remain completely unspecified and so unanticipated until the application is about to be run. Dynamic metatype association can be used to adapt the application at any time as it run, but the directive that performs the dynamic metatype associations, and so the control logic that governs the dynamic association of metatypes with arbitrary target objects or classes must be embedded in application code.

Iguana/C++ also supports dynamic metatype association, but the new MOP must be specified at compile time, and like Iguana/J the control logic for the adaptation must be specified in the application source code.

In these regards, Iguana systems do support some, but not all of the requirements for completely unanticipated dynamic adaptation. No Iguana mechanism currently exists that implements adaptation by metatype association where the location of the association, the nature of the associated metatype, and its association control and timing logic, can all remain unanticipated until after the application has started executing. In [125] a proposed solution to this problem is laid out for Iguana/J, whereby a `console` meta object class is designed which would continuously request the name of a class and the name of a meta object protocol, and perform the association dynamically in an unanticipated manner. However, this proposed Iguana/J solution is substantially less expressive and less flexible than the solution provided in this thesis. In addition, this proposed Iguana/J solution does not support the dynamic adaptation of individual objects, only classes.

While Iguana/C++, but especially Iguana/J have obvious advantages for unanticipated dynamic adaptation, both ignore the fact that the application and the user are most knowledgeable about changing high-level contextual requirements, and how they can be incorporated into intelligent control rules to drive the dynamic adaptation.

A summary of the adaptation characteristics of Iguana/C++ and Iguana/J is presented in table 2.1.1 below.

	Iguana/C++	Iguana/J
Anticipation of the adaptation contents	Compile time	During runtime
Anticipation of the adaptation location	Compile time	During runtime
Anticipation of the timing of the adaptation's application	Design time and compile time	Start of runtime or at compile time
Anticipation of the control logic for the adaptation's application	Design time and compile time	Start of runtime or at compile time
Adaptation binding time	Compile time and during runtime	Load time or during runtime
Adaptation binding mode	Dynamic	Dynamic

Table 2.1.1 Summary of the adaptation characteristics of Iguana/C++ and Iguana/J

As discussed in Chapter 1, this thesis aims to demonstrate the use of dynamic metatype association as a dynamic software adaptation technique. Iguana/J's support for dynamic metatype association, without requiring access to the source code of the object or class being adapted, and without the need to recompile the target application, has encouraged the design decision that Iguana/J should be used as the dynamic adaptation mechanism for the Chisel dynamic adaptation framework.

2.1.2 Java HotSwap

In version 1.4 of Java, support to HotSwap application classes at runtime [43, 146] was added, allowing a Java class within an application to be recompiled and replaced while the application is executing in debug mode. Current JVM implementations (version 1.4) support the use of the Java HotSwap mechanism to replace a Java class with another binary compatible version of that class [76], where the class structure and class interface can be reordered and augmented, but no part removed. In this version, only behavioural change by replacing or adding class methods is supported, but further support is planned. This HotSwap support was added to allow debugger tools to support a fix-and-continue strategy to code debugging. A tool, the HotSwap Client Tool [151] was developed to support the runtime management and replacement of Java class code.

Although this mechanism is not a reflective adaptation mechanism since no monitoring occurs and no causally connected metadata describes the behaviour or structure of the adapted system, this mechanism is included in this section because it allows behavioural and structural adaptation by manipulating the composition of classes and objects. As with other mechanisms that support the replacement of compiled software modules, if the original class source code is unavailable, the original class must be completely rewritten to include the adaptation being applied. This model also leads to an unfortunate loss of separation of concerns and tangling of adaptation and application code.

According to the classifications set out in this thesis, this tool can be used to implement completely unanticipated dynamic adaptation, since the adaptation, the target of the adaptation, the timing of the application of the adaptation, and the management logic supporting the adaptation's application can all remain unanticipated until during execution. The HotSwap Client Tool as presented does not provide a mechanism to control the adaptation in an automatic or context-aware manner since each adaptation must be individually applied by the user, at the exact moment when the adaptation is required. In addition, this tool only supports the adaptation of classes, not individual objects. While this does address the aim of this thesis to allow the user to influence the adaptation process, it does not provide support for automatic or default adaptation policies for users who do not wish to be directly burdened with this control.

	Java HotSwap
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	During runtime
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.1.2 Summary of the adaptation characteristics of the Java Hotswap mechanism

2.1.3 Javassist

Javassist [24, 26] is a post compile-time or load-time structural reflection framework for Java. It is used to edit bytecode in Java classes, allowing users to modify Java class files before runtime, or change the class at load-time as it is loaded by using a specialised classloader. Javassist allows users to create new classes, add fields or methods, or change the bodies of methods. Once the class is loaded, and all adaptations made to it, no further adaptation of the class is possible, i.e., Javassist can only perform static adaptation. However Javassist can be used to insert hooks to support dynamic adaptation for use by other adaptation frameworks, for example Wool and RAM, as discussed later in this chapter.

Knowledge of Java bytecode is not required as Javassist provides a high-level structural reflection library allowing source-level abstractions of the bytecode additions. Since it operates on Java bytecode, access to the source code of the adaptation target is not required. Again Javassist has only a programming interface so all adaptations must be specified in the application code. This means that all characteristics of all adaptations must be specified before the entire application is compiled, so unanticipated adaptation is not supported.

	Javassist
Anticipation of the adaptation contents	Compile time
Anticipation of the adaptation location	Compile time
Anticipation of the timing of the adaptation's application	Compile time
Anticipation of the control logic for the adaptation's application	Compile time
Adaptation binding time	Load time
Adaptation binding mode	Static

Table 2.1.3 Summary of the adaptation characteristics of Javassist

2.1.4 DART

DART (Distributed Adaptive Runtime) [121-123] is a runtime reflective framework for distributed adaptation, developed by Sony. DART builds upon some of the techniques used in the Aperios/Apertos reflective operating system [70, 164, 165]. A framework for reflective objects is provided to support functional behaviour adaptation of the application, which operates by allowing alternative method implementations (adaptive methods) to be selected via "selectors", in a manner similar to the Strategy design pattern [55]. Also included is a method interception system (reflective methods) for non-functional behaviour adaptation in response to environmental changes. Using this approach, intercepted method calls are redirected to a set of meta objects before and after invocation using a "reflector", which manages these meta-objects. Adaptive methods and reflective methods are shown in figure 2.1.4.1 (taken from [121]).

A runtime manager is instantiated for each application as it starts up. Adaptation policy functions, written in C, register for adaptation events and can introspect on both the base-level and meta-level code. When these adaptation events occur, the runtime manager then adapts the system using the policy functions that are registered for those adaptation events. These policy functions perform the adaptations at runtime by accessing individual base objects using the runtime manager and manipulating their selectors and reflectors. Policy functions can also be loaded into and unloaded from the runtime manager at runtime. Base-level objects can also access and change these selectors and reflectors via the runtime manager, thereby allowing an application to adapt itself, but these operations must be embedded in the application source code.

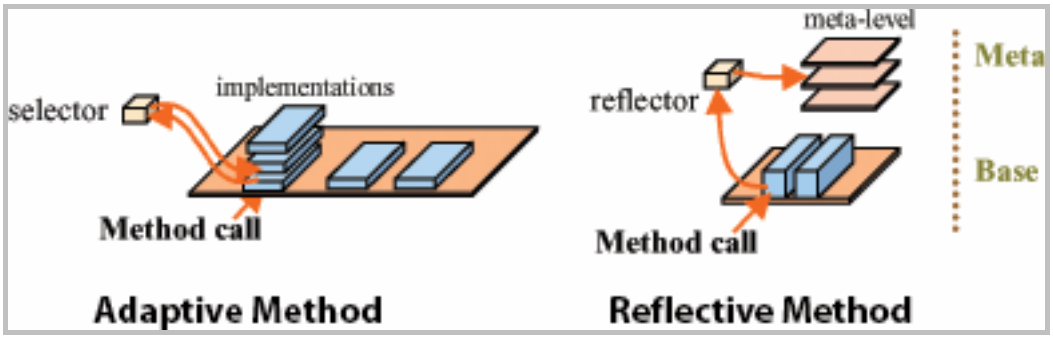

Figure 2.1.4.1 Adaptive methods and reflective methods in DART

Any individual class can be marked as adaptive in the application source code by having a default policy associated with it. In the source code, individual methods can also be annotated to be reflective methods or adaptive methods. OpenC++ [25] is used to create and bind the meta-level objects to the base-level objects at compile time. Since the code for reflection and adaptation is completely tangled with the application code, there is very little separation of concerns in this framework.

A collection of adaptable service libraries containing default adaptation policies and a set of generic events are also included with the framework, allowing applications developers to build complete adaptive distributed applications. These policies and events can also be extended by developers, but again these are compiled into the application. All default policy associations are also placed in 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, e.g., by adding new adaptive method implementations, or changing default meta objects used by reflective methods. However, there does not appear to be any mechanism to add new unanticipated policies or behaviours, or dynamically change methods to be adaptive or reflective, either at or before runtime.

DART does not support the dynamic specification of new adaptations, and many aspects of the adaptation must be anticipated at or before runtime. DART is of interest to the Chisel project for its support for named object specifications, event driven dynamic adaptation of functional and non-functional behaviours, adaptation using behavioural reflection, and a configuration mechanism that can be adapted for individualised adaptation policies. But, its ability to implement a metatype adaptation model, and in particular dynamic metatype association is extremely limited due to its inability to support dynamic adaptation additions. A summary of the DART project is presented in table 2.1.4.2 below.

	DART
Anticipation of the adaptation contents	Compile time and before runtime
Anticipation of the adaptation location	Design time
Anticipation of the timing of the adaptation's application	Design time
Anticipation of the control logic for the adaptation's application	Design time
Adaptation binding time	Compile time and before runtime
Adaptation binding mode	Dynamic

Table 2.1.4.2 Summary of the adaptation characteristics of DART

2.1.5 Kava

Kava [159, 161], formerly called Dalang [158], is a reflective architecture for Java that statically binds meta object classes to base-level classes at load-time, where each instantiated object has an associated meta object. It operates by using a custom Java classloader, which inserts Java bytecode binding hooks at load-time, before the class is JIT compiled. There is no support to unbind or replace these meta objects at runtime. Kava also supports only one meta object per base object, but does state that more can be added by meta object chaining. The primary use of Kava in the literature is the implementation of security policies for Java programs [160], helped by the strong separation of the meta-level and base-level, and the inability of the base-level programs to bypass the operations of the metalevel. Although Kava only supports static adaptation, it does so in a manner that is unanticipated until the start of runtime since the meta object class binding is specified in a separate XML file, so no tangling code is required to support this adaptation.

Kava would be a candidate to support metatype association with base-level classes, but not objects. In addition, dynamic metatype association would not be possible.

	Kava
Anticipation of the adaptation contents	Start of runtime
Anticipation of the adaptation location	Start of runtime
Anticipation of the timing of the adaptation's application	Start of runtime
Anticipation of the control logic for the adaptation's application	Start of runtime
Adaptation binding time	Load time
Adaptation binding mode	Static

Table 2.1.5 Summary of the adaptation characteristics of Kava

2.1.6 Guaraná

Guaraná [109-111] is a reflective architecture for Java that supports the configuration and runtime association of meta objects with base classes and objects in a secure manner. For the purposes of this project, Guaraná is similar to Iguana/J and MetaXa (described in section 2.1.7) in many respects since these meta objects can be used to dynamically change the behaviour of base-level classes and objects in a manner similar to dynamic MOP selection

in Iguana/J. Similar to Iguana/J, Guaraná supports a strong separation of concerns between the base-level and the meta-level, and so can support the association of unanticipated adaptations.

Meta object association with base objects and classes can only be accomplished in a programmatic manner and so some tangling of adaptation code with application code must occur. Despite being able to perform unanticipated adaptation on pre-written application elements, again without access to source code, the adaptation must still be anticipated before the entire application is compiled. In this respect, it is similar to Iguana/J and MetaXa since even dynamic adaptation is anticipated in source code.

As part of later work on a development kit for Guaraná [141], a launcher for Guaraná was introduced that supports the association of meta objects with classes at the start of runtime, similar to the configuration file used to statically initiate metatype association in Iguana/J.

Guaraná is a prime candidate to support the metatype model, and dynamic metatype association. As discussed in [125], one of the main drawbacks of using Guaraná to implement the metatype model is the inability to automatically associate a metatype with a base-level class, and so change the behaviour of all instances of that class and its subclasses, since only the particular class is adapted, but not its current instances or subclasses.

	Guaraná
Anticipation of the adaptation contents	Compile time / start of runtime
Anticipation of the adaptation location	Compile time / start of runtime
Anticipation of the timing of the adaptation's application	Compile time / start of runtime
Anticipation of the control logic for the adaptation's application	Compile time / start of runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.1.6 Summary of the adaptation characteristics of Guaraná

2.1.7 MetaXa

MetaXa [56, 57] is another reflective architecture for Java, similar to Iguana/J and Guaraná, that supports the runtime association of meta objects with base-level classes and objects, and so can be used to dynamically change the behaviour of base-level classes and objects. In terms of this thesis MetaXa is very similar to Iguana/J and Guaraná and so merits little extra discussion, but one of the main features of the MetaXa system is its support for a very low-level and fine-grained meta-level programming model. This however means that MetaXa is more difficult to use than Iguana/J or Guaraná to implement higher-level dynamic adaptation.

Again, similar to the original Guaraná work, MetaXa only supports programmatic association of meta objects with base-level objects and classes. This again means that despite the ability to dynamically adapt previously written software components in an unanticipated manner, adaptation code is tangled into application code. This weak separation of concerns means that in its current form completely unanticipated dynamic adaptation is not possible with MetaXa since the adaptation must be anticipated before application compile time.

Similar to Guaraná, MetaXa could also be used to implement the metatype adaptation model. Again discussed in [125], MetaXa's support for unanticipated dynamic associations of metatypes is limited where more than one metatype is associated.

	MetaXa
Anticipation of the adaptation contents	Compile time / start of runtime
Anticipation of the adaptation location	Compile time / start of runtime
Anticipation of the timing of the adaptation's application	Compile time / start of runtime
Anticipation of the control logic for the adaptation's application	Compile time / start of runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.1.7 Summary of the adaptation characteristics of MetaXa

2.1.8 K-Components

K-Components [44, 45] uses asynchronous architectural reflection to build context-aware adaptive software. The adaptation logic that specifies the adaptive behaviour (adaptation policy) is written as adaptation contracts in a declarative programming language (ACDL). Adaptation occurs in response to adaptation events raised by either the application components or from the evaluation of adaptation rules themselves, e.g., anticipated failure to achieve a set goal. The meta-level configuration manager runs asynchronously and so periodically reflects on the need for adaptation, using polled adaptation events and the adaptation contracts, thereby greatly reducing reflective computation overhead. The reified software architecture is arranged as a typed directed component configuration graph, where changes to the configuration graph during dynamic adaptation are performed as transactional operations, so that the result is again a correct directed configuration graph. 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. A component’s external interface cannot be changed by architectural reconfiguration since only reconfiguration operations on the configuration graph is allowed. This maintains correctness of the component configuration graph but severely restricts how the system can adapt.

New components can be loaded at runtime from a DLL or as a remote CORBA component, but their interface must be previously specified since the configuration graph is a static representation of the architecture of the system, and cannot be extended to support new component types at runtime. The system also requires that the adaptation event types are known to the configuration manager at compile-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. Also, all components must be written to support the K-Components framework, whereas the application model for Chisel, as described in this thesis, supports adaptation of application objects that were written in a manner oblivious to any adaptations that might later be applied.

	K-Components
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	Design time
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.1.8 Summary of the adaptation characteristics of K-Components

2.2 Adaptation using AOP techniques

Aspect oriented programming (AOP) [30, 50, 53, 84] is a programming methodology that allows crosscutting concerns to be declared as "aspects". A cross-cutting concern is a property or function of a system that cannot be cleanly declared in terms of individual components, because the application of the crosscutting concern must be scattered or distributed across otherwise unrelated components. AOP supports the separate or "oblivious" [53] production of these aspects, which are later incorporated or "woven" into the application components at a specified or quantified set of "join points".

In [53], "obliviousness" is declared to be one of the key components of AOP. Obliviousness refers to the degree of separation between the aspects of the system and how they can be developed independently without preparation or cooperation. This preparation is related to anticipation, as described in Chapter 1, whereby the degree of unanticipation is equivalent to the degree of obliviousness supported. Many AOP systems support weaving before runtime, but newer dynamic AOP systems (e.g., AspectWerkz, JMangler, Wool, and PROSE) described in this section allow aspects to be woven at load-time or runtime, thereby allowing the incorporation of aspects into base programs to remain unanticipated until load-time or runtime. Aspect/J is discussed particularly for its contributions as one of the original

and most influential AOP systems, as an introduction to the general concepts of AOP, and in terms of how it used by other AOP tools. AspectWerkz, JMangler, Wool, and PROSE are discussed as tools that could form part of the Chisel framework, could implement the metatype model, or demonstrate some similar functionality to the Chisel framework. Trap/J is discussed as a prototype dynamic adaptation framework where many of the aspects of its supported adaptations can remain unanticipated until runtime, with its similarities to the Chisel framework and its limitations discussed. A number of other systems such as Hyper/J [114] or Binary Components Adaptation (BCA) [81] could also have been discussed here, however these systems do not support runtime adaptation and so do not additionally contribute to the discussion of research directly related to Chisel.

2.2.1 AspectJ

AspectJ is a very influential aspect-oriented extension to the Java programming language that supports aspect-oriented programming in a general-purpose manner [162]. In AspectJ, an aspect is a language construct, used to specify a modular unit to encapsulate a crosscutting concern. It is defined in terms of *pointcuts* (a collection of join points and contextual values at those join points), *advice* (code executed before, after, or around a join point when it is reached), and *introductions* (normal Java code to be introduced into base classes).

AspectJ also contains a comprehensive and expressive pointcut specification language that supports the specification of complex pointcuts. The join points supported by AspectJ include: method calls and execution, constructor calls and execution, read and write access to a field, exception handler execution, and object and class initialisation execution. Join points in AspectJ are dynamic in nature since they capture not just location, but also timing and context information. Pointcuts can be specified using boolean combinations of the following: matched or partially matched join points based on signature strings, type names, or the type of the join point caller, target, or arguments; other pointcuts; and join points that occur within the flow of control of other pointcuts. When using aspect weaving as a mechanism to support adaptation, this expressive pointcut specification language allows very fine-grained control of where and when adaptations are applied.

AspectJ supports aspect weaving at compile time, post-compile time, and load-time. Compile time weaving is used when the application code and the aspect code are specified together. Post-compile time weaving supports weaving compiled aspects into compiled java code using bytecode transformation [65]. Load time aspect weaving is accomplished using

the same bytecode transformations combined with a custom classloader specified at application startup. In this respect, it does not support unanticipated runtime adaptation as there currently does not appear to be an explicit mechanism to support dynamic re-weaving of aspects after they have been loaded. The dynamic join point locations at which the aspect advice and introductions are woven, and the control logic supporting the selection of these dynamic join points are specified in the pointcut specifications in the aspect definition code, before it is compiled, so adaptation location must be anticipated. A summary of AspectJ's support for unanticipated dynamic adaptation is given in table 2.2.1 below. Despite its lack of support for completely unanticipated dynamic adaptation, AspectJ is used by a number of other dynamic adaptation systems to insert hooks to be used at runtime to support dynamic adaptation, e.g., RAM (described in section 2.4.6) and TRAP/J (described in section 2.2.6).

	AspectJ
Anticipation of the adaptation contents	Before load time, specified in the adaptation code
Anticipation of the adaptation location	Before load time, specified in the adaptation code
Anticipation of the timing of the adaptation's application	Before load time, specified in the adaptation code
Anticipation of the control logic for the adaptation's application	Before load time, specified in the adaptation code
Adaptation binding time	Compile time, post compile time, and load time
Adaptation binding mode	Static

Table 2.2.1 Summary of the adaptation characteristics of AspectJ

2.2.2 JMangler

JMangler [85, 86] is a framework that supports bytecode level transformation of Java classes at load-time. These transformations can change almost any feature of the class being loaded, including its interface, members, and method code. In the latest version of JMangler, this is achieved using the Java HotSwap mechanism to replace the Java classloader that is used to load all application classes, but not the bootstrap classloader that loads system classes. JMangler essentially provides a framework to support the implementation of different aspect weavers. One weaver supplied with the framework, the BCEL weaver, uses the Byte Code Engineering Library (BCEL) [32] to perform bytecode manipulation. This weaver supports the weaving of *transformers* (aspects) into the code of the loaded class. Transformers are written in Java, and so can contain arbitrary conditions to control their own weaving characteristics, particularly their location (similar to pointcut specifications in AspectJ). Transformers can also make extensive use of the BCEL library, to perform both interface changes and code changes to classes as they are loaded. A list of transformers, along with information such as JMangler runtime options and known incompatibilities between transformers, can be specified in an XML file used at load time by the JMangler

framework. This maintains a strong separation of concerns between the adaptation mechanism and the application code. With this declarative specification method, none of the characteristics of the adaptations need be anticipated until loadtime of the target application class. However, JMangler cannot support dynamic weaving of transformers, since the adapted class cannot be reloaded to be adapted again, so dynamic adaptation is not possible. Since JMangler manipulates class bytecode, no access to the source code of the target application is required. JMangler is used, or planned for use, in a number of other adaptation frameworks, including Javassist and AspectWerkz (described in the next section).

	JMangler
Anticipation of the adaptation contents	At load time
Anticipation of the adaptation location	At load time
Anticipation of the timing of the adaptation's application	At load time
Anticipation of the control logic for the adaptation's application	At load time
Adaptation binding time	At load time
Adaptation binding mode	Static

Table 2.2.3 Summary of the adaptation characteristics of JMangler

2.2.3 AspectWerkz

AspectWerkz [3, 15, 155] is an AOP framework that supports both static and dynamic aspect weaving. In early versions it used the JMangler system [85, 86] as a backend to support runtime weaving, but a custom reimplementation of the Java bootstrap classloader using the BCEL library [32] has replaced this. AspectWerkz employs an aspect model similar to AspectJ by using join points, pointcuts, advice, and introductions.

In AspectWerkz, aspects can be defined either as source-code annotations, or in an XML format. Using source code annotations, aspects are defined as Java classes containing advice and introductions, and pointcuts. The kind of advice (e.g., around, before, after etc.), the location for introductions, and the matching of join points, are all defined in custom annotations in the aspect's source code or in the application source code. Using the XML aspect specification method, an aspect is defined by selecting a set of predefined named pointcuts, advice, and introductions. Named pointcuts are defined in XML format in terms of string-based join point lookups, with named advice and introductions, which are defined using Java implementation classes.

By using post-compilation or load-time static weaving, dynamic AOP can be enabled by use of hooks and aspect containers that can be used at runtime. Using this model, join points used in the pointcut must be specified at weave-time, but the advice and adaptation to be performed can remain unanticipated until runtime. These changes however can only be

performed via a programmatic interface, so recompilation is required for unanticipated adaptations.

Since join points are integrated into the code of the application at load time, it is very difficult to add support for unanticipated join points. However a complicated method of adding new join points is introduced in [155] using the Java HotSwap mechanism combined with load time preparation. In this case, all Java classes are slightly modified to support later (if necessary) hotswapping of Java classes that have the new join point enabled. Currently this method requires that the new join points only be accessed at runtime via a programmatic interface in code defining an aspect, and not from the XML deployment document. Therefore, similar to Iguana/J, no high-level or declarative method exists to exploit access to unanticipated adaptation locations without changing aspect source code and recompiling. This restriction is subject to change in the immediate future.

Currently AspectWerkz does not support specifying a join point in class constructors so it is not possible to adapt an object by intercepting its construction. Similar to PROSE (described in section 2.2.4) and Wool (described in section 2.2.5), AspectWerkz does not support the adaptation of individual objects, just classes. These factors would seriously affect the ability of AspectWerkz to support the metatype adaptation model.

	AspectWerkz
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	Start of runtime
Anticipation of the timing of the adaptation's application	Start of runtime
Anticipation of the control logic for the adaptation's application	Start of runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.2.4 Summary of the adaptation characteristics of AspectWerkz

2.2.4 PROSE

PROSE [117, 118] is a dynamic AOP framework for Java that supports runtime aspect weaving. PROSE was originally intended as a framework for debugging or rapid prototyping of AOP systems, which could later be completed using compile-time or load-time aspect weaving [118]. This was mainly due to its use of the Java Virtual Machine Debug Interface (JVMDI) [147], which resulted in a large performance penalty. A later version of PROSE [117] was implemented by modifying the open source IBM Jikes Research Java Virtual Machine [66], greatly improving its performance. In both versions, new aspects can be dynamically woven, with support for these aspects to define new join points, for which new interception hooks are created at weave time. Therefore, PROSE can

be used to support dynamic adaptation by weaving additional non-functional behaviours into the code at runtime. However, PROSE only supports weaving at a class level, therefore individual objects cannot be adapted individually, similar to AspectWerkz and Wool. A number of graphical user interfaces are included to manage the unanticipated weaving of new aspects at runtime. In this respect, PROSE supports completely unanticipated dynamic adaptation since the new adaptation (aspect), its join point, the timing of the aspect weaving, and the management of the weaving is all unanticipated until after the start of runtime. In the current implementation of PROSE (version 1.2.1) method return values cannot be changed, thereby severely restricting PROSE's ability to act as a dynamic adaptation framework, but this is due to be resolved in future releases.

PROSE is another possible candidate to support the metatype adaptation model, as discussed in [125], despite its focus on AOP rather than reflective adaptation. Again, as with AspectWerkz and Wool, individual objects cannot be adapted so dynamic metatype association with individual objects cannot be supported.

	PROSE
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	During runtime
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.2.5 Summary of the adaptation characteristics of PROSE

2.2.5 Wool

Wool [138] is a dynamic AOP framework that uses a hybrid aspect weaving approach by using both the Java Platform Debugger Architecture (JPDA), like PROSE, and the Java HotSwap mechanism, as seen in AspectWerkz. Since JPDA supports remote activation of breakpoints at runtime, join point hooks in the form of debugging breakpoints can be set from outside of the application. A pointcut may be made up of a number of these hooks. Each aspect specifies a pointcut, and a set of advices to be executed when one of the pointcut's join points (represented as breakpoints) is reached. New aspects can be serialised and sent to the target JVM for weaving at any pointcut. In one approach, when a join point is encountered, the inserted breakpoint redirects the operation to the wool runtime component in a manner similar to a debugger, where advices are then executed. An alternative approach allows the advice to be hotswapped into the application class thereby improving performance if the join point is encountered repeatedly. This is achieved by using Javassist to rewrite the class, without access to its source code, and have the adapted class

replace the original application class using the Java HotSwap mechanism. This also removes the breakpoint, so calls to the debugger are removed. However, this mechanism means that all objects of the woven class will have the adaptation incorporated, in a manner similar to AspectWerkz and PROSE.

Currently the aspect programmer must specify in the aspect's source code whether the advice should be hotswapped in or run by the debug interface, so in order to achieve good performance, the aspect writer should anticipate the access patterns of the aspect's pointcut. Wool does not support adding introductions as seen in AspectJ, but a proposed solution to use Javassist is provided.

Since aspects, their weave location, and the timing and control logic of their weaving are all specified separately from the application at runtime, this system is classified as supporting completely unanticipated dynamic adaptation. However, unlike the Chisel adaptation framework, no support is provided to allow users or operating context to influence how these adaptations are applied at runtime in a reactive manner.

Again, as with AspectWerkz and PROSE, Wool is a possible candidate to support the metatype adaptation model, but the inability to adapt individual objects means that Wool cannot be used as a mechanism to support dynamic metatype association with individual objects.

	Wool
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	During runtime
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.2.6 Summary of the adaptation characteristics of Wool

2.2.6 TRAP/J

TRAP/J [135, 136] is a prototype unanticipated dynamic adaptation framework for Java. It combines compile-time aspect weaving using AspectJ [162] and unanticipated dynamic adaptation with wrapper classes and delegate classes. At compile time the programmer selects a subset of application classes that will be adaptable. The TRAP/J system then automatically creates AspectJ code to replace all instantiations of the selected classes with wrapper class instantiations. Java code for each wrapper class and a meta object class for that wrapper class is also automatically created. At runtime, each instantiated wrapper object has an instance of the original wrapped object and a meta object bound to it. These wrapper

objects redirect all method calls to their meta object, which in turn act as placeholders for a set of delegate objects that may handle the invocation of the method, or adjust its parameters prior to execution by the original wrapped object. New, dynamically created delegates can be added or removed at runtime via an RMI interface using a management console. These delegates can be added on a per object basis since the meta objects can supply a name for each instance and register it in an RMI registry.

This framework was used to demonstrate the dynamic adaptation of a network-enabled application by replacing instances of the `java.net.MulticastSocket` class with instances of an adaptable socket class `MetaSocket` [80]. The TRAP/J framework however does not support completely unanticipated dynamic adaptation. The adaptation, its intelligent and controlled dynamic application, and the timing of its application all remain unanticipated until runtime, but the possible locations for the adaptations are specified in the application source code, since the version of AspectJ used requires access to the application source code. Despite improving the performance of the TRAP/J framework, this restriction greatly limits the nature of the unanticipated dynamic adaptations that can be applied. No information is provided about whether the generated meta object class code can be modified prior to compilation and weaving.

In addition, TRAP/J seems to delegate the invocation of the method to only one delegate; the first one it finds implementing the method, but this ordering of delegates can be configured. This means that only one adaptation can be applied at a time since adaptation behaviours are not automatically composed. Also TRAP /J does not seem to allow the user to apply an easily recognisable name to the base object being adapted, and so may make it difficult for the user to identify the object to which adaptations should be dynamically applied. From the documentation TRAP/J does not seem to support applying dynamic adaptations via new delegates on a structured class-wide or interface-wide basis since RMI registry look-ups are at a per meta object basis.

	Trap/J
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	Compile time
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.2.7 Summary of the adaptation characteristics of Trap/J

2.3 Adaptable middleware

This section discusses adaptable middleware with respect to how they are related to or have influenced the design of the Chisel dynamic adaptation framework. Although further discussed in Chapter 5, adaptable middleware is an important influence in the design of the Chisel adaptation framework. What constitutes middleware is hard to define and outside the scope of this research, but here middleware is considered as a set of enabling technologies that resides above a network-enabled operating system but support applications from below. Middleware systems should shelter the application developer from the intricacies of the underlying environment, communication subsystems and distribution mechanisms, thereby providing a single view of the underlying environment, to facilitate development, deployment, and management of applications.

A number of the other systems discussed in other sections, e.g. DART, K-Components, TRAP/J, CARISMA, RAM, and M3 could also be included in this section, however, their primary relevance to or influence on the Chisel framework was with respect to the topics covered in those sections. 2K, the OpenORB research, and ACT are adaptation frameworks which are specialised for performing dynamic adaptations on middleware. They are discussed in terms of how each one supports the application of adaptations where some or all properties of those adaptations can remain unanticipated until runtime. They are also discussed in terms of how their mechanisms could be extended beyond a middleware domain.

2.3.1 DynamicTAO / 2K

DynamicTAO [90, 91, 132] is an extension of the TAO ORB. The TAO ORB [140] allows different aspects of its operation to be selected at load time using the Strategy design pattern [55] as specified in a configuration file. DynamicTAO allows these strategies to be inspected and changed at runtime in a reflective manner, while still maintaining consistency. This can be achieved via the `DynamicConfigurator` component that supports dynamic querying of the current ORB strategies and supports the dynamic uploading of new strategies. However, only a fixed number of locations within the ORB can be adapted. Strategy implementations can also be uploaded to or downloaded from a remote location, then selected for use. A runtime graphical interface, *Doctor* (Dynamic ORB Configuration Tool), is presented to support this possibly remote runtime dynamic adaptation. A mobile

code model is also presented that allows adaptations to be copied and applied across a network of interconnected ORBs [92], again controlled by a graphical interface.

2K [90, 91, 93] is an adaptable operating system that is built on top of DynamicTAO. Component-based applications use 2K as an adaptable execution environment, where components can be downloaded at runtime and combined into the application. In the middleware, all entities in the system, including users, applications, components, and devices can be modelled as remotely accessible CORBA components and so can be queried. 2K also includes an automatic configuration system, whereby components provide a complete list of which resources they need and a list of other components they require, which allows these Quality of Service (QoS) requirements and inter component dependencies to be reified. This allows application programmers and users to view the system state and adapt the system as they wish while the 2K system automatically maintains the QoS requirements and component dependencies.

Reflection in 2K is based on architectural awareness, since the architecture of the system and dynamic runtime component interdependencies are reified in "component configurators", with one meta object for each component. Each component configurator can be extended to include application-specific adaptation code to adapt the component and its dependencies based on events fired by other component configurators. This use of these component configurators maintains a strong separation of concerns between the adaptation target and the adaptation mechanism.

In 2K, new components, and their configuration information, can be acquired from a remote component repository at runtime using a pull-based approach to updating the system. To improve performance, 2K also supports a push-based system using mobile agents, which include inspection code, configuration changes, and code for new components, as seen in DynamicTAO.

Mostly it is just the structure of the middleware that is adapted using architectural reflection techniques, but since DynamicTAO supports dynamic loading of components, application components can also be unloaded and reloaded to support behavioural change at the application level. This can be achieved in a completely unanticipated manner as the location of the adaptation, the control logic and timing of its application, and what the adaptation does, can all remain unanticipated until runtime. Although the component resource requirements can be specified in a declarative script format that is used when the component is first loaded, the adaptation logic must be specified in the component configurator source code.

The 2K framework is essentially an adaptive execution environment that supports dynamic loading and unloading of application components in a user-, application- and resource-aware manner, using both local and global context to support this adaptation. The main focus of the 2K project is on automatically maintaining both the local and network-wide consistency of a distributed adaptable execution environment, which adapts itself in a reactive but anticipated manner, rather than adapting applications that use the 2K framework. Recent work [142] further demonstrates how the 2K framework can be used to build dynamically adaptable applications, but the degree to which the adaptations in this implementation are unanticipated is unclear, since control logic is again coded directly into the component configurators.

Although a powerful dynamic adaptation mechanism, tentatively classified as supporting completely unanticipated dynamic adaptation, adaptations are only possible by the dynamic selection of predefined strategies in DynamicTAO, and the dynamic loading and unloading of components and component configurators in 2K. Chisel is a general-purpose dynamic adaptation framework, which can dynamically adapt any compiled Java application, in a completely unanticipated manner, without the need to statically compile adaptation logic into the adaptation itself.

	DynamicTAO	**2K**
Anticipation of the adaptation contents	During runtime	During runtime
Anticipation of the adaptation location	Design time	During runtime
Anticipation of the timing of the adaptation's application	During runtime	During runtime
Anticipation of the control logic for the adaptation's application	During runtime	During runtime
Adaptation binding time	During runtime	During runtime
Adaptation binding mode	Dynamic	Dynamic

Table 2.3.1 Summary of the adaptation characteristics of DynamicTAO and 2K

2.3.2 Next Generation Middleware at Lancaster

A large body of research has accompanied OpenORB [14], a component model and reflective middleware project at Lancaster University. At load time, appropriate middleware service components are selected and composed as a middleware instance. At runtime, components can be dynamically replaced with a new component exporting the same interfaces. Every object or component is associated with a “meta-space”. In order to help separating the concerns at the meta-level, the meta-space is broken into a number of meta-models, each of which can be accessed using a different MOP. Using each meta-model, different aspects of the middleware instance can be inspected and adapted.

The OpenORB model supports structural reflection using the encapsulation and compositional meta-models. The encapsulation meta-model (also called the interface meta-model) represents a component interface in terms of its methods, its principal attributes and other characteristics like inheritance relationships. Each component's interface is defined in terms of required support and provided support for standard method invocations, streaming data interfaces, and an events interface. Interfaces of different components are connected over bindings, which may be local or distributed, and represent a link between a provided interface and a required interface. These bindings are reified as usable components, which may themselves contain several other components bound together. The composition meta-model (also called the architecture meta-model) reifies the implementation and composition of the component, by providing a causally-connected component graph, and is used to examine, validate, and adapt the make-up of the component and its bindings to other components. This meta-model is used to add, remove, and replace components, and change the architectural constraints used to validate component assemblies, and control and constrain adaptation.

Behavioural reflection is supported with a number of meta-models, depending on the OpenORB version. If present, the environmental meta-model, for each interface, represents the execution environment and reifies the management of messages, processes etc, which are also reified as graphs. The interception meta-model, when present, allows the association of "interceptors" with interface bindings to provide pre- and post-processing behaviours, parameter inspection, and insertion of methods.

The resources meta-model, introduced in [9, 11], is based on the abstraction and reification of resources and tasks. Low-level resources are managed by resource managers that can also act as resource factories to create higher-level or more complex resources. For each address space, a single resources meta-model can view and adapt the components that represents the resources of the system. Dynamic adaptations for QoS are usually accomplished using this meta-model, when present, or by the environmental meta-model.

Management components can also be added to OpenORB architectures. These components can be used to monitor or adapt the system at runtime, either by implementing new component strategies or selecting new strategies to be incorporated into the component graph. These management components can themselves be dynamically managed to support dynamic changes to adaptation policies. The dynamic specification of adaptation policies using scripting languages is also introduced in one of the older OpenORB prototypes [9, 12]. This concept of managed and constrained adaptation is further discussed in [10], where

it was planned that the impact of alternative middleware component configurations would be analysed before they were applied, thereby allowing their application to be managed by adaptation managers, again possibly using adaptation policy scripts to control how the reconfiguration is accomplished. These adaptations could be triggered at runtime by user interaction, or by context changes. However, support for and use of these policy-based management techniques does not seem to have been included in later versions of the OpenORB prototypes.

The dynamic adaptation models described here as part of the next generation middleware research currently ongoing at Lancaster University present an array of methodologies and tactics for adapting systems. Many prototypes exist using combinations of some of these methodologies. An early version of OpenORB was prototyped using the Python programming language [2] with a resource management framework added in [48]. The OpenCOM component framework [115], again based on the OpenORB model, is built on top of a subset of Microsoft's COM [100]. OpenCOM provides low-level support for meta-models, using a series of COM type interfaces to objects that are encapsulated within the service component being developed. OpenORBv2 [13], is implemented in C++ as a CORBA compliant ORB using the OpenCOM component framework, as described in [28]. A reflective middleware framework for mobile computing, called ReMMoC [60], has also been implemented on top of OpenCOM. ReMMoC focuses on dynamic discovery and use of distributed services, for use in a heterogeneous mobile computing environment.

As a result of the several designs, and several prototype implementations of the OpenORB models, it is difficult to classify this research by how unanticipated dynamic adaptation is supported, and what can be supported for each programming language used. The use of dynamically changeable management components that use interpreted scripting languages to support unanticipated dynamic adaptation and its management would seem to be of most interest to this thesis. The ability to dynamically add and remove components, and as a result management components, means that adaptations, the adaptation locations, and when the adaptations are applied can remain unanticipated until during runtime. The use of metadata and metalevel architectural information to constrain and manage dynamic adaptation is also of importance with respect to dynamic and unanticipated use of contextual information. When combined with user interfaces to support unanticipated dynamic adaptation and requirement specification, adaptation management logic can also remain unanticipated. Overall, this thesis tentatively classifies the OpenORB research as supporting

completely unanticipated dynamic adaptation, but whether these techniques are all currently implemented in entirety in one or more prototypes is uncertain.

The degree to which the metatype adaptation model can be incorporated into the OpenORB model is also difficult to define. In general, the OpenORB model of adaptation is to replace components or component implementation strategies, rather than associate adaptations with existing components. A slight exception would be the use of interceptors at interface bindings, where wrapping behaviours can be added, in a manner similar to associating a new behaviour to an interface. Although presented as an adaptable middleware framework, the research associated with OpenORB and its associated projects can largely be extended to a generalised component-based model for applications other than middleware. However, unlike Chisel which is a general-purpose adaptation framework, all components to be adapted must be written according to the OpenORB programming model.

	OpenORB
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	During runtime
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.3.2 Summary of the adaptation characteristics of OpenORB

2.3.3 ACT

ACT [133, 134] is a generic adaptation framework for CORBA compliant ORBs that supports completely unanticipated dynamic adaptation. When the ORB is started ACT is enabled by registering a specific portable request interceptor [106], intercepting every remote invocation request and handing them to a set of dynamically registered interceptors. These dynamically registered interceptors can be added in an unanticipated manner. Rule-based dynamic interceptors allow the request to be redirected to a different source or handed to either a number of local proxy components exporting the same interface as that of the destination server component [133] or to a generic local proxy component [134]. This generic proxy component can also be dynamically created in an unanticipated manner. This proxy in turn can request a rule-based decision maker component, which can incorporate an event service, to either perform the invocation, change parameters and forward the request to its original destination or a different destination.

A prototype implemented in Java is described whereby the Quality Objects (QuO) framework [4], an aspect-oriented QoS adaptation framework for CORBA ORBs that uses

compile-time weaving, was used with Orbacus [69], a CORBA-compliant ORB, to support completely unanticipated runtime aspect weaving in the ORB. A number of management interfaces were also provided to manage the runtime registration of new rule-based dynamic interceptors, and the addition of new rules to these interceptors.

The ACT framework is classified in this thesis as supporting completely unanticipated dynamic adaptation since any adaptation, its intended target, when the adaptation is applied, and the control logic governing its application can all remain unanticipated until after the start of runtime. However, this framework differs from the Chisel framework presented in this thesis since ACT can only be used to adapt CORBA compliant middleware systems. Chisel is a general-purpose dynamic adaptation framework since it can be used to dynamically adapt any application at runtime, including middleware frameworks, as described in Chapter 5.

	ACT
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	During runtime
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.3.3 Summary of the adaptation characteristics of ACT

2.4 Policy or interpreted script driven adaptation

As discussed in Chapter 1, a policy-based adaptation management model is an ideal mechanism to drive a general-purpose dynamic adaptation framework, since the adaptation mechanism can be completely decoupled from the adaptation management. Adaptation frameworks that use a policy-based management model supporting dynamically updateable policy specifications allow the timing requirements and control logic for the application of adaptations to be dynamically specified, in an unanticipated manner.

This section discusses policy-based management systems, and adaptation frameworks that support policy-based management of adaptations. Similarities in design, design influences, and relevance to the Chisel framework are also discussed. Ponder, GEM, and REI are discussed particularly in terms of their contribution to the area of policy-based management, and in terms of how they influenced the design of Chisel's support for policy-based adaptation management, rather than as mechanisms that could be directly used by Chisel. Conversely, Correlate, CARISMA, RAM, and M3 are discussed in terms of how they

support policy-based management of adaptation, and in terms of how they compare to the Chisel framework. A number of other systems such as the work in Lancaster University [49] could have been discussed here but again they are not directly related to the Chisel framework.

2.4.1 Ponder

Developed at Imperial College London, the Ponder policy language [33-35, 47] is a declarative, object-oriented language for specifying security and management policies for distributed object systems. This project is built upon the experience in policy-based management in Imperial College over the last several years, e.g., [98, 104, 143]. In Ponder there are four basic types of rule. Positive or negative authorisation rules provide or restrict permission to perform an action. Obligation rules specify that an action must be performed in response to an event. Refrain rules specify that an action cannot be performed. Delegation rules allow a subject to delegate some of their authorisations to another subject. The policy subjects, to whom the rule applies, are a set of managers that manage a set of managed objects (target objects). Actions are specified as operations with parameters on a target object, while goals are a higher-level view of an objective that can be decomposed into a series of actions. Each rule is also governed by a series of constraints that must be satisfied before the goals or actions can proceed.

Ponder also supports several types of composite policies, i.e., groups of policies that model enterprise specifications on organisational structures. “Roles” are used to describe a set of policies that all contain the same subject to show a high-level view of the position or enterprise role of a subject. Ponder also supports hierarchies of Roles. “Relationships” are used to model the interactions between roles. Related policies can also be joined together to form “groups” that are used as units for organizational and reuse purposes. “Management Structures” are used to model organizational units by containing a predefined set of roles and relationships. Meta-policies can also be used to provide a high level view of groups of policies as a policy itself and is mainly used to describe constraints between groups of policies.

The policy language described in this thesis is loosely based on Ponder obligation policies. In the Chisel system, as events occur due to changes in context, the adaptation manager is obliged to adapt the behaviour of underlying system services if constraints allow. A fully functional policy language to specify security constraints and enterprise level policies are however outside the goals of this research.

2.4.2 GEM

Also created in Imperial College London, GEM [97] 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 provides for the dynamic definition of simple events, composite events, and rules to describe how monitors should respond to these events. Again, this language is interpreted so the script can be dynamically updated to allow the system to adapt in a policy-controlled manner.

The language used in the Chisel framework described in this document closely resembles the language used in GEM to define and respond to events. Also included in GEM is support for systems where the values of status variables can be thresholded so that jitter does not occur if the status variable changes rapidly to values that are close to the threshold value.

2.4.3 REI

The REI policy framework [78, 79] is an application-independent policy-based management system. It has four basic policy types: *rights, prohibitions, obligations and dispensations*, which correspond to positive and negative authorisations, obligations, and refrains in Ponder. The REI framework is mostly focused towards security policies. A set of rules, written in a Prolog like language, can be associated with a managed domain entity. Any time an action is to be performed on this entity, the REI policy system is requested to verify that this action can be performed. REI also provides a mechanism to define *actions* that can be used in obligation rules. A general action is described by a unique identifier, the target objects on which the action can be performed, a set of preconditions that must be satisfied before the action can be executed, and lists the effects of the action. REI provides the ability to reason about rules and respond to queries but does not provide a mechanism to enforce policy rules or perform actions. Therefore, in order to support unanticipated dynamic adaptation, REI must be used in conjunction with a separate adaptation mechanism.

2.4.4 Correlate

Presented by the DistriNet research group in Katholieke Universiteit Leuven, Correlate [73, 74, 128, 129, 154] is a concurrent object-oriented language based on C++ (and later Java) to support mobile agents. It has a flexible runtime 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 the metatype adaptation model. The non-functional behaviours are designed to be largely application independent. However, independent policy objects can be defined to contain application-specific information to assist in the operation of these meta-level non-functional behaviours. The meta-level system was initially used to implement non-functional concerns such as real-time operation [5], load-balancing and security [131], and fault tolerance [128]. Later this system was used to customise ORBs, using application-specific requirements, as an adaptable graph of meta-level components that could be extended or adapted at runtime [153].

The application-independent non-functional behaviours are implemented as meta object classes, which can interact with the base program to adapt its operation using a message-based MOP [130]. These meta object classes define a set of possible property values in a policy template. Each application class has an associated singleton policy class, which is an instantiation of one of these templates, containing application-specific information. These singleton policy class objects are consulted by the meta-level before performing the non-functional behaviours of the application [131], allowing the operation to be customised in an application-specific manner. However, this policy system is limited since policy templates are imposed at the time the meta program is written. These templates, written in a declarative language, must fully define what possible customisations an application may require. The policies, also written in the same declarative manner, select values for template properties according to the application class they are associated with.

These templates cannot be changed, so adaptation in response to unanticipated requirements cannot be fully handled. Policies are written before runtime by a system integrator, and these policies are then integrated with the application. Policies cannot be changed at runtime, since policies are translated to code that is compiled. Unanticipated forms of dynamic adaptation cannot be achieved in this architecture as the meta-level programmer and template designer needs complete a-priori knowledge of the possible changes in context values that may occur, and also the set of customisations from which the meta-level can choose is fixed at compile time. Again, all adaptable systems using Correlate must be designed and written according to the Correlate programming model.

	Correlate
Anticipation of the adaptation contents	At compile time and before runtime
Anticipation of the adaptation location	At compile time and before runtime
Anticipation of the timing of the adaptation's application	At compile time and before runtime
Anticipation of the control logic for the adaptation's application	At compile time and before runtime
Adaptation binding time	At compile time and during runtime
Adaptation binding mode	Dynamic

Table 2.4.4 Summary of the adaptation characteristics of Correlate

2.4.5 CARISMA

Research carried out at University College London on the CARISMA project [20-23] presents a design for peer-to-peer middleware based on service provision. Each node can export services and possible different behaviours or implementations for those services. Services can be selected according to user and application context information, as specified in an "Application Profile", an XML document. Embedded in this application profile is the application-specific information that the middleware uses when binding to these services, e.g., which service behaviour to use in response to changes in the execution context. The middleware is responsible for maintaining a view of the system environment by directly querying the underlying network-enabled operating system. Applications may request to view and change their profiles at runtime, thereby adapting the middleware as application-specific and user-specific requirements change dynamically.

This system also provides the ability for the application to be informed by the middleware of changes in specific execution conditions, supporting the development of resource-aware applications. This system is based on the provision of multiple implementations of the same service with different behaviours, in a manner similar to the strategy pattern [55], unlike the Chisel framework, which adapts the service itself.

However, the primary contribution of this work focuses on the identification and resolution of profile conflicts [22], and not on the actual provision of an adaptable middleware implementation. No information is provided about how the services are implemented, if they can be dynamically loaded, how they implement their different strategies, or if these strategies can be expanded at runtime. It is suggested that the ReMMoC middleware framework [60] could be used here, but no implementation information is available. This means that the degree to which this system supports unanticipated adaptation cannot be evaluated. However, it should be noted that the application profile that controls how the system adapts, and the mechanism for profile conflicts, can both be adapted at runtime in an unanticipated manner.

2.4.6 RAM

RAM (Reflection for Adaptable Mobility) [17, 36-38, 71] from École des Mines de Nantes, takes the approach of completely separating functional and non-functional aspects of an application in a manner related to aspect oriented programming (AOP). 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. *Container* meta objects wrap each application, and support the composition of other meta object which implement these non-functional concerns. The wrapping of application objects with *Container*s occurs at either load time using Javassist in [17] or compile-time using AspectJ in [36-38, 40]. These meta objects provide the middleware services by selecting appropriate *RoleProvider* objects for each service, i.e., the meta objects that provide the actual implementations of the services. Adaptation can occur by adding, removing, or reordering these *RoleProviders*.

RAM also provides a resource manager, whereby the system maintains a tree of *MonitoredResource* objects, which describe a contextual resource or group of resources. These *MonitoredResource* objects are updated by *probe* objects that actively monitor the environment. *MonitoredResource* objects can be queried explicitly or change notifications can be requested. The *Container* meta objects, that wrap each application component, can also expose the *MonitoredResource* interface, supporting queries of application context as resources, thereby exploiting application-specific knowledge in the adaptation process, as also seen in Chisel and K-Components systems. The resource manager can also signal the adaptation engine when an interesting resource change occurs in a manner similar to the context manager in the Chisel framework.

The set of meta objects (aspects) to use in each *Container* is adapted at runtime by means of an adaptation engine that uses a declarative application policy and a system policy, written in a Scheme-like language, and which are both passed to the adaptation engine when the application is started. Application policies define pointcuts (a dynamic set of join points, i.e., *Container* objects) in the application, and the named non-functional aspects to be used at these pointcuts, in an application-aware but resource-independent manner. The set of rules that determine which join points make up a pointcut is specified in the application policy, but these rules are dynamically evaluated, so this set of join points can change dynamically. The non-functional aspects woven at these pointcuts are defined in the system policy in an adaptive Condition-Action model, where sets of application-independent but

resource-aware conditions are dynamically evaluated to decide which meta objects will implement the non-functional aspect. When the conditions are dynamically evaluated, the bindings of meta objects can be changed, in a manner similar to dynamic aspect weaving. Therefore, the set of join points that make up a pointcut, and the set of meta objects that implement an aspect are dynamic in nature according to the rules in the policies. The current system does not support dynamic changes to the policies, and so cannot support unanticipated adaptation logic, however this is planned for future versions. In most cases where AspectJ is used, access to the source code of the application is also required.

A version of RAM suggests using a configuration file to specify the set of join points that can be used, and use AspectJ to create these join points at compile time rather than have `Container`s wrap every application object [40]. This means however that all possible locations for adaptation must be anticipated in the source code of the application.

Preliminary designs for an adaptation framework extending RAM, which would possibly support completely unanticipated adaptation by allowing dynamic specification of policies and dynamic selection of adaptation locations, is presented in [39], but this system has yet to be implemented.

	RAM
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	Compile time and load time
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	Start of runtime
Adaptation binding time	During runtime
Adaptation binding mode	Dynamic

Table 2.4.6 Summary of the adaptation characteristics of RAM

2.4.7 M3

The M3 [67, 68, 120] architecture is 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 reconfigure how application and system components interact with each other within the runtime environment (M3-RTE). In this system, all components interact and coordinate with each other using only events. As these events occur, they can be monitored and used to trigger adaptation of the architecture and the underlying collection of distributed services and network protocols. The M3 runtime environment also maintains context variables that can be used to perform these adaptations in a context-aware manner. An entire application architecture can be modelled as a series of enterprise roles with duties whose scope are defined in *obligation*,

prohibition, and *permission* policies similar to those in the Ponder system. The system then adapts itself as roles move in/out of contexts, or as the roles' policies are changed. The current prototype parses the MEADL script, translates it to XML, and then translates these rules into a series of event notification subscriptions using the Python programming language. Prototype adaptable tickertape and email client applications have been developed to run on the M3 runtime environment.

While the Chisel framework 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 general-purpose reflective mechanism like dynamic metatype association, as used in the Chisel framework. Morphable objects, i.e., objects that can change their type at runtime, are mentioned in [67] but no more information is available about these reflective techniques. It is suggested that the MEADL script can be dynamically updated and reparsed in an unanticipated manner, but no confirmation of this is available in the documentation. However, a programmatic interface also exists to create and remove rules at runtime, which can also be used from within other rules.

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 cannot often be measured as a boolean state, a ranged metric value would provide more expressiveness and accuracy.

	M3
Anticipation of the adaptation contents	During runtime
Anticipation of the adaptation location	During runtime
Anticipation of the timing of the adaptation's application	During runtime
Anticipation of the control logic for the adaptation's application	During runtime
Adaptation binding time	No adaptations are bound, only component interactions are changed dynamically
Adaptation binding mode	No adaptations are bound, only component interactions are changed dynamically

Table 2.4.7 Summary of the adaptation characteristics of M3

2.5 Overview

The previous sections of this chapter provide an in-depth discussion of software adaptation with respect to support for completely unanticipated dynamic adaptation, support for general-purpose adaptation of compiled software in a user-aware and context-aware manner, and the applicability of the metatype adaptation model in a subset of these systems.

As stated, the ability to fulfil these requirements is the prime objective of the Chisel dynamic adaptation framework. As can be seen from the review there remains a lack of usable mechanisms to support this combination of requirements.

Using the degree of support for completely unanticipated dynamic adaptation as the primary requirement for adaptation mechanisms provides only a small number of frameworks that fulfil this requirement. While 2K, the OpenORB model, ACT, and the M3 architecture all support completely unanticipated dynamic adaptation, they are not suitable for general-purpose adaptation of third party software, since the target software for these systems must be designed specifically for these systems. The Java HotSwap mechanism is tentatively classified as supporting completely unanticipated dynamic adaptation, but this mechanism provides no high-level support for managed dynamic adaptation. PROSE and Wool also both support completely unanticipated dynamic adaptation, however these mechanisms again provide no high-level support for user or context awareness.

Guaraná, MetaXa, and Iguana/J can also be extended to support completely unanticipated dynamic adaptation with the provision of a mechanism to support the runtime specification of unanticipated adaptation directives and support for those adaptations.

This detailed classification of adaptable systems based on varying support for completely unanticipated dynamic adaptation is further illustrated in table 2.5.1 at the end of this chapter.

Table 2.5.2, also at the end of this chapter, further classifies the adaptable systems discussed in the previous sections according to the degree of support for dynamic adaptation found in each system. Systems are classified according to their adaptation binding times, and their adaptation binding modes as described in [31]. Here "dynamic" binding refers to the ability to bind and then unbind an adaptation at runtime, whereas in [31] this is referred to as "changeable" binding. Binding time refers to the specific time in a software product's lifetime that a feature is bound or woven into that product. Examples of binding times include design time, compile time, before runtime, at runtime, load time, during runtime etc.

This classification is in addition to the primary classification of systems in terms of their support for completely unanticipated dynamic adaptation, and could have extended to classify any of a large number of adaptation classifications, e.g., composability of adaptations, granularity of adaptation, application or problem domain addressed by the system, programming languages supported, etc. However, as stated, this thesis is particularly focused towards unanticipated dynamic adaptation so the classifications used are limited to characterising support for unanticipated adaptation and support for dynamic adaptation. A more generalised taxonomic classification of adaptation mechanisms and models is given in [18].

2.6 Conclusions

This chapter described a number of systems and research that influenced and are similar to the contributions of this thesis.

The objectives of this research are: to support adaptation where no aspect of the adaptation is anticipated until the adaptation is needed and if possible to perform this adaptation without access to the source code of the application, and to investigate the usefulness of the metatype adaptation model to dynamically adapt software.

In order to reach these objectives, a number of requirements must be met. An adaptation framework that supports completely unanticipated dynamic adaptation is needed. This adaptation must be able to perform adaptations on third party software modules. To test the metatype model, a framework is needed that supports the dynamic definition, loading, and association of metatypes at runtime.

From the research described, it can be seen that there currently exists no mechanism that completely fulfils the objectives and requirements of this research. This thesis continues with a detailed design of a dynamic adaptation framework, Chisel, which does meet these requirements.

Project name	Anticipation of the contents of the adaptation	Anticipation of the adaptation location	Anticipation of the timing of the adaptation's application	Anticipation of the control logic for the adaptation's application
Iguana/C++	Compile time	Compile time	Design time and Compile time	Design time and Compile time
Iguana/J	During runtime	During runtime	Start of runtime / compile time [1]	Start of runtime / compile time [1]
Java HotSwap	During runtime	During runtime	During runtime	During runtime
Javassist	Compile time [8]	Compile time [8]	Compile time [8]	Compile time [8]
DART	Compile time and before runtime	Design time	Design time	Design time
Kava	Start of runtime	Start of runtime	Start of runtime	Start of runtime
Guaraná	Compile time / start of runtime [2,7]	Compile time / start of runtime [2,7]	Compile time / start of runtime [2,7]	Compile time / start of runtime [2,7]
MetaXa	Compile time / before runtime [7]	Compile time / before runtime [7]	Compile time / before runtime [7]	Compile time / before runtime [7]
K-Components	During runtime	Design time	During runtime	During runtime
AspectJ	Before load time [8]	Before load time [8]	Before load time [8]	Before load time [8]
JMangler	Load time	Load time	Load time	Load time
AspectWerkz	During runtime [7]	Start of runtime	Start of runtime	Start of runtime
PROSE	During runtime	During runtime	During runtime	During runtime
Wool	During runtime	During runtime	During runtime	During runtime
TRAP/J	During runtime	Compile time	During runtime	During runtime
DynamicTAO	During runtime	Design time	During runtime	During runtime
2K	During runtime	During runtime	During runtime	During runtime
Next generation middleware at Lancaster	During runtime	During runtime	During runtime	During runtime
ACT	During runtime	During runtime	During runtime	During runtime
Correlate	Compile time and before runtime [4]	Compile time and before runtime [4]	Compile time and before runtime [4]	Compile time and before runtime [4]
CARISMA	Unknown [3]	Unknown [3]	During runtime	During runtime
RAM	During runtime	Compile time / load time [5]	During runtime	Start of runtime
M3	During runtime	During runtime	During runtime	During runtime [6a]

Table 2.5.1 Overview of adaptation anticipation in reviewed dynamic adaptation systems

Notes for Table 2.5.1:
Note: The policy frameworks, Ponder, GEM and REI are not included in these tables as they are management frameworks, not adaptation frameworks.
1: In Iguana/J adaptation application can be applied at start of runtime or in the source code. Design for an administrative console for adaptation during runtime is also presented.
2: In Guaraná, support for adaptation unanticipated until the start of execution was added later as a launcher with the Guaraná Development Kit (GDK), otherwise dynamic adaptation code must be embedded in the application code.
3: In CARISMA, no information about how services or their operation strategies (policies) are implemented are provided in the documentation.
4: In Correlate, adaptation policies can be defined prior to runtime, but only according to templates and parameters set at meta program compile time, so Correlate adapts at runtime but only according to policies and strategies defined prior to runtime.
5: In RAM, different versions exist, each with different anticipation of the location of the adaptation.
6a: In M3, it is suggested that the MEADL configuration script can be dynamically updated in an unanticipated manner at runtime, but no confirmation is available in the documentation.
7: Supports unanticipated dynamic adaptation of compiled code, but only has a programmatic interface; so all adaptation must be specified and compiled before runtime.
8: All adaptation characteristics are specified in the adaptation code before the adaptation is compiled.

Project name	Adaptation binding
Iguana/C++	Dynamic, at compile time and during runtime
Iguana/J	Dynamic, load time or during runtime
AspectJ	Static, at compile time, post-compile time, or load time
DART	Dynamic, at compile time and before runtime
K-Components	Dynamic, during runtime
DynamicTAO	Dynamic, during runtime
2K	Dynamic, during runtime
JMangler	Static, at load time
AspectWerkz	Dynamic, during runtime
Kava	Static, at load time
Java HotSwap	Dynamic, during runtime
ACT	Dynamic, during runtime
TRAP/J	Dynamic, during runtime
PROSE	Dynamic, during runtime
Guaraná	Dynamic, during runtime
MetaXa	Dynamic, during runtime
CARISMA	Unknown [3]
Correlate	Dynamic, at compile time and runtime[4]
M3	Adaptations are not bound [6b]
RAM	Dynamic, during runtime
Javassist	Static, at loadtime
Wool	Dynamic, during runtime
Next generation middleware at Lancaster	Dynamic, during runtime

Table 2.5.2 Overview of the adaptation binding categories used in reviewed adaptation systems

Notes for Table 2.5.2:
Note: The policy frameworks, Ponder, GEM and REI are not included in these tables as they are management frameworks, not adaptation frameworks.
3: In CARISMA, no information about how services or their operation strategies (policies) are implemented are provided in the documentation.
4: In Correlate, adaptation policies can be defined prior to runtime, but only according to templates and parameters set at meta program compile time. Correlate adapts at runtime but only according to policies and strategies defined prior to runtime.
6b: In M3, how the components of the system coordinate and communicate with each other is changed dynamically (policy controlled), but no adaptations are bound to the components.

Chapter 3
THE CHISEL FRAMEWORK, CONCEPT AND DESIGN

This chapter describes the design and operation of the Chisel dynamic adaptation framework.

This chapter begins by discussing the aims and objectives of the Chisel project. From this discussion a series of requirements for the Chisel dynamic adaptation framework is established. This chapter then continues with an in depth discussion of the metatype model, explaining the metatype concept, discussing both the possibilities and limitations of the model, and presents the use of the dynamic association of metatypes as a dynamic software inspection and adaptation technique.

The use of events in the Chisel adaptation framework is then discussed, and the Chisel event model is presented. This is followed with a discussion of the Chisel context model, and how the Chisel framework supports context-awareness. The use of this event model and context model as parts of the policy-based management model used in the Chisel framework is described, accompanied by an overview of how the Chisel adaptation framework operates. The Chisel policy language is then presented and discussed.

The Chapter ends with a discussion of how the Chisel dynamic adaptation framework design presented in this chapter fulfils all the objectives and requirements in order to support completely unanticipated dynamic adaptation of general-purpose software, in a context-aware manner.

3.1 Objectives and requirements

As described in Chapter 1, the objectives of this thesis are twofold: having researched the field of unanticipated dynamic adaptation, to design and build a prototype dynamic adaptation framework that supports completely unanticipated dynamic adaptation and to demonstrate the feasibility of using runtime behavioural reflection and dynamic use of metatypes to support dynamic adaptation of software in a general-purpose manner As these objectives are discussed, a number of requirements that must be satisfied by the Chisel framework will become apparent.

3.1.1 Requirements for completely unanticipated dynamic adaptation

Completely unanticipated dynamic adaptation is introduced in Chapter 1. A particular adaptation is completely unanticipated only if, the nature of the adaptation (*what*), the location of the adaptation (*where*), when the adaptation is applied (*when*), and what control logic drives or triggers the application of the adaptation (*how*), can all remain unanticipated and unprepared until after the application being adapted has started executing.

Location of an adaptation unanticipated until runtime

If the location at which an adaptation is to be applied is to remain unanticipated until runtime, there must exist a mechanism for the insertion of a hook to support the unanticipated placement of adaptive code at the required location at runtime. An alternative is to have a very extensive set of prepared locations (hooks) for adaptations already woven into the target application. In this alternative situation the location of particular adaptations remains unanticipated, only the possible need for arbitrary future adaptations has been anticipated. This does not break the requirement at which the location that a particular adaptation is applied remains unanticipated

If the system is capable of dynamic adaptation, unanticipated adaptations can be applied at an unprepared location, or one of a large set of prepared locations. In order for this requirement to be fulfilled, there must also exist some mechanism to name, describe, and/or specify the adaptation location at runtime, otherwise the locations cannot be found and used. This dictates that some form of runtime support is available to support the dynamic identification and specification of adaptation locations. In an object oriented system the

granularity of these locations will be have the granularity of an object, an interface, or a class.

This requires support for inspection of the software in order to locate where the adaptation is required, and in addition, support to apply a unique identifier to that location for use in the adaptation control logic. This requirement is discussed in more detail in section 3.1.2.

Management and control of an adaptation unanticipated until runtime

The control logic that is used to manage the application of an adaptation can also remain unanticipated at runtime if there exists some method for the control logic to be specified and interpreted at runtime. This control logic must be capable of specifying what adaptation should be applied, where and when it should be applied, and conditions to restrict the application of the adaptation if necessary. This adaptation logic may be of a reactive nature, whereby the adaptation manager should wait for something to happen and then react to that event in the manner specified, or, the adaptation logic may be proactive in nature, whereby the adaptation should be immediately applied.

Since many dynamic adaptations are necessarily required because some state, resource, or requirement has changed for the user, application, or execution environment, this dynamically specified control logic must also support the querying of this runtime context. If this context information remained constant, many adaptations would not be required, or could be anticipated before runtime. However, what context information is of importance may not be known before the need for an adaptation has arisen. Therefore it must be possible to have access to arbitrary context information, but allow the unanticipated specification of what context information should be managed,

Timing of the application an adaptation unanticipated until runtime

When an adaptation is applied will be defined by the control logic used by the adaptation manager to manage its application. As stated above, this control logic may be either proactive or reactive in nature. Proactive adaptation control logic means that the application of the adaptation is triggered as soon as possible after the adaptation manager receives and interprets the adaptation control logic, which can be specified at an unanticipated time. For reactive control logic, the application of the adaptation is triggered after the occurrence of some arbitrary event or the evaluation of some arbitrary condition. In this case it should be possible to apply the adaptation after the possibly unforeseeable triggering of an adaptation operation. Therefore, the adaptation framework must support the dynamic binding of adaptations at unanticipated times. It should also be possible to unbind that adaptation in the

case that an adaptation it is no longer required, or would conflict with another previously unanticipated adaptation.

Contents of an adaptation unanticipated until runtime

In order to support completely unanticipated adaptation, it must also be possible to support the dynamic and unanticipated specification of the particular adaptations themselves. This is only possible if the execution environment of the target application supports the dynamic loading the new executable code that defines the adaptation. In addition, it must be possible to refer to the possibly newly-loaded adaptation so that it can be used in a controlled manner at runtime, since some form of reference to the adaptation must be incorporated into the adaptation logic used by the adaptation manager. In order to support dynamic adaptation it is also a requirement that the adaptation mechanism support the dynamic binding and unbinding of the adaptation with its target once it is loaded, otherwise it cannot be applied.

Summary of requirements for completely unanticipated dynamic adaptations

The objective to design and build a general-purpose adaptation framework that supports completely unanticipated dynamic adaptation where the nature of the adaptation (*what*), the location of the adaptation (*where*), when the adaptation is applied (*when*), and what control logic drives or triggers the application of the adaptation (*how*), can all remain unanticipated and unprepared until after the application being adapted has started executing, necessitates that each of the requirements specified in this section are addressed. These requirements form the primary design requirements of the Chisel dynamic adaptation framework.

Unanticipated location (*where*)	• Ability to dynamically create adaptation hook locations at runtime *or*, Have already inserted a large set of hook locations, suitable for any adaptation • Ability to identify and refer to adaptation locations at runtime
Unanticipated control (*how*)	• Ability to dynamically specify adaptation control logic, and have this control logic interpreted at runtime • Ability to include arbitrary context monitoring in this control logic, and support the dynamic specification of what should be monitored.
Unanticipated timing (*when*)	• Ability to bind and unbind adaptations at specified points at any time during execution
Unanticipated adaptation contents (*what*)	• Ability to dynamically create arbitrary executable code at runtime • Ability to load this newly created code refer to this executable code after it is loaded

Table 3.1.1 Summary of requirements to support completely unanticipated dynamic adaptation

3.1.2 The ability to inspect and identify internal parts of the software

In any closed or compiled system where unanticipated dynamic adaptation is required, it is first required to locate and identify the part of the software to be adapted. For software written using an object-oriented methodology the adaptation is usually required at the granularity of a class, an object, or a group of objects or classes. This is made particularly difficult when the software to be adapted is written according to a "black-box" model, whereby the implementation details of how a software module operates are hidden [82]. The black-box model has a number of advantages. It hides unnecessary implementation detail and clutter from users of the module. It protects, by obscuring, the internal operation of the module, thereby securing the value of the software module. It protects the module's internal working data and code from being used in unsupported and possibly dangerous manners. It also provides an easy to use modular building block for building more complex software whereby the boundaries and interface to a module are clearly defined. But this black-box model for software has some very serious disadvantages, some of which even severely restrict the reusability of the software, one of the core aims of the model in the first place. It reduces the set of use-cases of the module to a set that is restricted to exactly those supported by its interface, even if the module would be capable of wider use with possibly very minor changes. There should be no need to reinvent a module because it cannot fulfil an unanticipated requirement, especially if the original module could fulfil that requirement with only slight adaptation. Yet another drawback of this black-box approach is that the user of a software module has no way to discern how the module is operating, so that the user might improve upon it, debug it, tailor their own code to use this module in a better manner, or learn from the lessons and contributions of the module's implementation. A closed black-box module is difficult to adapt because its operation is obscured, making the inspection and identification of the operation of compiled software modules a difficult task. If a module's internal operation can be probed and observed, then it may be possible to adapt that module to extend its capabilities.

To challenge this model, the Chisel framework should support the probing and inspection of a compiled class or object. It should be possible to inspect and view all method and state accesses, both into and out of the object or class, as well as the types and values of all method parameters and return values, and the types of all classes and objects used by that class. Along with these requirements, this must be achieved without access to the source code of the module.

Once a complete picture of all of the fields and methods of a particular object or class, with a set of data values as method parameters and their corresponding return values, and the same for all classes and objects accessed by that particular class or object, the observer has a good deal more information to help deduce and reverse engineer the internal operations and behaviours of the class or object, and so break down the black-box model to a certain extent. Once the user has determined how the object or class operates, the user is then in a position to adapt that object or class in an intelligent and informed manner. This ability to adapt and extend third-party software in a general-purpose manner, without requiring access to the source code of that software, is a core objective of this research. As stated, the ability to inspect, probe, and profile the operation of arbitrary software is therefore a key requirement of the design of the Chisel framework.

While it is often not necessary that these inspection and probing features are available for use in an unanticipated manner, the Chisel framework still aims to provide a mechanism to support the unanticipated runtime probing, debugging, and evolution of software modules.

Introspection	• Ability to dynamically inspect, probe, and profile the operation of arbitrary compiled software at runtime • Ability to perform this introspection without access to the source code of the software • Ability to perform this introspection in an unanticipated manner

Table 3.1.2 Summary the requirements to enable introspection of arbitrary software

3.1.3 Demonstrating metatypes

Metatypes [125, 127, 139], as introduced in Chapters 1 and 2 are type-independent changes in behaviour, i.e., "snap-on" behaviours, that can be associated with classes and objects, which may change the current functional behaviour of the target, or may implement new non-functional behaviours with that class or object. The metatype model is relatively new and untested, and has not been demonstrated to any large extent. This thesis aims to prove the capabilities, explore the limitations, and demonstrate the operation of metatypes for dynamic adaptation of running software. It is therefore a requirement that the Chisel adaptation framework makes use of the metatypes to perform dynamic adaptations. A detailed discussion of metatypes, i.e. what they are, what they do, what they cannot do, and how they do it, follows in the section 3.2 of this chapter. Further discussion about the metatype model will follow throughout this thesis.

Metatypes	• Make use of metatypes as a dynamic adaptation mechanism, to demonstrate their abilities and usefulness

Table 3.1.3 Summary the requirement to demonstrate the use of metatypes

3.2 The Chisel adaptation mechanism: dynamic metatype association

This section describes in detail what exactly metatypes are, and how they are used. This section also describes how dynamic metatype association can be used as a mechanism to achieve dynamic software adaptation. How this mechanism for dynamic adaptation is harnessed by the Chisel dynamic adaptation framework is also presented.

3.2.1 What are metatypes

Although metatypes have been introduced and discussed in Chapters 1 and 2, this section will define what is meant by a "metatype" with respect to this research.

In an object-based computational system, each object models some distinct entity in an application domain. An object's "object model" refers to the set of concepts used to construct that object so that it represents that part of the modelled domain, i.e., the way the object is implemented and how it makes use of facilities provided by both the programming language and the runtime environment during its execution. An object's object model is different from its type or class, since many objects of different types can make use of the same language and implementation support, and be defined in terms of the same underlying concepts and constructs (i.e., have or use a similar object model).

Schäfer first introduced the concept of a metatype by stating that an object's metatype is a characterisation of an object's object model [139], and every object has an associated metatype. This metatype may be the default metatype defined by the programming language used to implement the object, and provided by the object's execution environment at runtime. This metatype can be changed, either statically or dynamically. The new metatype extends the default metatype by introducing new concepts and behaviours that will be automatically used by the object at runtime.

Redmond [125] refines the concept of what a metatype is by stating that a non-default metatype represents a behaviour change to the object from the behaviour specified in the object's source code. This means that behavioural change can be accomplished by changing the metatype of an object. This is achieved by either changing the way the object is implemented, or wrapping the behaviour of the object with code implementing a new behaviour, either statically or dynamically.

However, directly changing or replacing the implementation of an object is a difficult process, especially if the part to be replaced or changed is currently active, so this replacement model is usually best suited to evolving a system in a debug environment [42, 43]. Behavioural change can also be accomplished by "associating" these changes with locations in the target object, or locations within the execution environment, where instead of inserting the new behaviour's code into the application source code at these positions, the execution of the operation is delegated to an object responsible for carrying out that operation, while adapting or redirecting its operation if necessary [125]. The execution of a new behaviour can occur alongside or around the original behaviour of the target object, by wrapping the behaviour of the target object and adapting or tailoring the intercepted operation, or by introducing the new behaviour before, after, or instead of the intercepted operation [125]. This mechanism has the added advantage of maintaining a clean separation between the new behaviour's code and the target object's code.

Metatypes can also be associated with classes. In this case the behaviours that are changed may be both the static[1] behaviours of the class, the behaviours of each current and future instance of the class, and the behaviour of all subclasses and their current and future instances [125]. This is a useful method where the behaviours of many objects are to be changed, instead of just one.

Dynamic metatype association refers to associating a new metatype with an object or class at runtime and so changing its behaviour on the fly. Dynamic metatype association is not a specialisation of metatype association, but instead a core constituent of the metatype model. Each time metatype association is mentioned in this section, dynamic metatype association can also apply.

3.2.2 The use of metatypes for behavioural change

Since metatypes contain behavioural changes, metatype association implies behavioural change. This section looks specifically at adapting or evolving an object or class, possibly changing the functional operation of the object or class. One particular focus of the Chisel project is the investigation of the usefulness of dynamic metatype association as a mechanism to perform dynamic introspection and adaptation of behaviour.

---

[1] Here static refers to the behaviour and data embedded in a class, instead of in each of its instances. For example, static methods, static data fields, and class initialisation procedures, implemented using the **static** keyword in Java and C++

A short discussion on software adaptation and evolution is presented in Chapter 1. This section will deal with the types of software adaptation that can be accomplished with metatype association, particularly dynamic metatype association. This section also focuses primarily on behaviour change rather than structural or architectural change.

Metatype association is an adaptation mechanism whereby the operations and accesses to an object are intercepted and wrapped by adaptation code providing new behaviours. This allows the insertion of adaptation code instead of, before, after, or around the operation of the intercepted operation, or redirection of the operation to a different operation or perhaps a different object, all without changing, replacing, or damaging any part of the object's source code. The metatype association model does not support the insertion, or replacement of code in the adapted object. This means that the blocks of code in an adapted object can be treated as "black boxes", that may be used, possibly combined with extra processing of their inputs and outputs, but may not be changed themselves. If the code is fully compiled and the source code is not available, this is especially true.

An object's functional behaviour is described as "how an object acts and reacts, in terms of its state changes and message passing", where a message is analogous to an operation or method-call, so "message passing" refers to method calls performed on and by the object [16]. If accesses to the object state, method calls into and out of the object, and creation and deletion of that object, can all be intercepted and possibly changed, then the behaviour of an object can be completely wrapped or changed. The metatype model supports these interceptions. So all behaviours, of all objects, can be changed in any manner. In practice however, this is rarely the case. The degree of behavioural change supported is restricted to the amount of interception support provided by the particular implementation of the metatype model used.

An object's functional behaviours dictate what the object does. An object's non-functional behaviours describe how it performs its functional behaviours. Non-functional behaviours are often not specified directly in the object source code since default versions of these non-functional behaviours are often provided at runtime as part of the execution model of the runtime environment. Example non-functional behaviours include characteristics of performance, security, remote accessibility, persistence, concurrency control, etc. These types of non-functional behaviours can also be provided or adapted by metatype association.

3.2.3 Adaptations using metatypes implemented using Iguana

As described in Chapter 2, Schäfer and Redmond both used the Iguana reflective model to implement metatype associations. Schäfer presented, and then used Iguana/C++ in [139], while Redmond presented, and then used Iguana/J in [125]. Firstly, computational reflection was used to selectively reify parts of the underlying object model of objects and classes. Metatype association was then accomplished by adapting the reified representation of the object model, thereby associating the new behavioural change with the target object by intercepting and wrapping its reified object model.

Both mechanisms tightly bind metatype implementations with runtime behavioural reflection. Metatype behaviours are embedded in the meta objects that are instantiated, ordered, and controlled by a runtime meta-level management system. Behavioural interception is accomplished by providing default meta object implementations for the reified parts of the object model, and then allowing these meta objects to be extended.

A MOP implementation is a collection of meta objects that act together to provide a new behaviour, i.e., a metatype implementation. An object or a class can only select one MOP, but this MOP may be composed of other MOPs. MOP composition follows the rules of metatype composition described above. In Iguana/C++ the concept of a metatype is tightly bound to that of a MOP (hence, only one metatype per object). In Iguana/J metatypes are also implemented by MOPs, but each MOP is considered to be a different metatype, each providing a change in behaviour.

Different implementation of the Iguana reflective programming model support different types of interceptions by reifying different parts of the underlying object model. (See reification categories in [19, 59, 64, 125, 126, 139]). Since Iguana/J is used by the Chisel framework, the reification categories provided by Iguana/J are of primary importance to this design. The operations that can be intercepted, called reification categories, include: object creation, object deletion, method invocations inwards, method invocation to other objects, and state read and write accesses. Once an operation is intercepted, the performance of that operation is delegated to a meta object to perform or adapt the performance of that operation by wrapping its execution. As described in Chapter 2, a metatype is implemented as a combination of meta objects each of which provides an implementation of a reified reification category. Figures 3.2.1 - 3.2.5 below demonstrate, using segments of code from an Iguana/J meta object class, `ExampleExecuteIntercepted`, how method invocations are reified, and how these reified invocations can be adapted.

```
class ExampleExecuteIntercepted extends ie.tcd.iguana.MExecute {
  public Object execute(Object obj, Object[] args, Method meth ) …  {
      …
      Object result = proceed(obj, args, meth);                    /* execute the method */
      …
      return result;
  }
};
```

Figure 3.2.1 Default operation of an intercepted method invocation

Once an operation is intercepted additional adaptation operations can be inserted before or after the original operation, as seen in figure 3.2.2, where synchronisation operations can be profiled, for example to allow the user to detect synchronisation errors.

```
class ExampleExecuteIntercepted extends ie.tcd.iguana.MExecute {
  public Object execute(Object obj, Object[] args, Method meth ) …  {
      …
      // Perform some before processing e.g.
      if ( (meth.getName().compareTo("wait")==0)
          System.out.println("Iguana: Sync: wait starting for object: "+obj.toString());
      …
      Object result = proceed(obj, args, meth);                    /* execute the method */
      …
      // Perform some after processing e.g.
      if ( (meth.getName().compareTo("wait")==0)
          System.out.println("Iguana: Sync: wait is over for "+obj.toString());
      …
      return result;
  }
};
```

Figure 3.2.2 Before and after behaviours for an intercepted method invocation

```
import java.lang.reflect.Method;
class ExampleExecuteIntercepted extends ie.tcd.iguana.MExecute {
  public Object execute(Object obj, Object[] args, Method meth ) …  {
      …
      // redirect the invocation to the static method "mywait" of class "MySemaphore"
      if ( (meth.getName().compareTo("wait")==0){
          obj = null;              //call to a class static method, so invocation target is set to null
          meth = MySemaphore.class.getDeclaredMethod("mywait" … );
          …
          args[0] = 1000;          //change a parameter, in this case the wait timeout in millisec
      }
      …
      Object result = proceed(obj, args, meth);                  // execute the new method
      …
      return result;
  }
};
```

Figure 3.2.3 Redirecting and adapting an intercepted method invocation

Once an operation is intercepted the operation itself can be adapted by changing the parameters, changing the return value, redirecting the operation to a different operation, or redirecting the operation to a different class or object, and then performing the adapted

operation. This is demonstrated in figure 3.2.3 above, where all synchronisation accesses for the target object are redirected to a static method call in a different class.

These meta object classes can then be combined to form a metatype and dynamically associated with an arbitrary application object or class, as seen in figure 3.2.4 and figure 3.2.5, and as seen in section 2.1.1 of Chapter 2.

```
protocol ExampleMetatype {
  reify Creation: ExampleCreationIntercepted();
  reify Execution: ExampleExecuteIntercepted();
  …
}
```

Figure 3.2.4 Iguana/J: metatype declaration

```
import ie.tcd.iguana.Meta;
java.net.Socket  mySocketObject = new java.net.Socket ();
Meta.associate( mySocketObject, "ExampleMetatype", …);
Meta.associate( java.awt.Button.class, "ProtocolVerbose", …);
```

Figure 3.2.5 Iguana/J: Dynamic metatype association

The examples above declare a new metatype called *`ExampleMetatype`*, which is then associated with the object *`mySocketObject`*, which is of type *`java.net.Socket`* and also with the class *`java.awt.Button`*. Further examples of the use of Iguana/J are included throughout this thesis.

These mechanisms of intercepting and adapting the behaviour of an application object or class can be used to adapt either the functional or non-functional behaviours of the object or class. Depending on the behavioural adaptation that is required, different parts of the object's object model may need to be reified, either for introspection or for adaptation. Different versions of the Iguana reflective frameworks, which implement the metatype model, support the selective reification of different parts of the object model, e.g., reification of method invocations on the object, object creation, object state access, the object's class, inheritance information, etc. Examples of possible behavioural changes may include introspection and error checking [139], persistence and remote operation [64], software evolution [125], selection of alternative method implementations [46]. All were achieved using metatype association. All of these behaviour changes can be applied to objects of different types, since the metatype of an object is orthogonal to its type. For example, adding persistence behaviour to an object by wrapping its old behaviour and redirecting object creation and deletion to a persistent object store, all without changing the object's data, interface or internal behaviour, does not change its type, but rather its metatype.

As an example, the following section looks specifically at the dynamic addition of non-functional behaviours for introspecting, probing, and profiling of an object or class.

3.2.4 Introspection, probing, and profiling using metatypes

It is often difficult to know what is happening as an application executes, even for the designer and developer who actually specified what should happen in the design and source code of the application. Sometimes it is necessary to look inside an object as its data is accessed and updated, and its code executed. The degrees of introspection required may include the ability to indicate what the access patterns are, to identify what is being accessed or changed, to see the value that something has been changed to, to identify the results of an operation, etc. [41]

Rather than rewriting the source code of a class or object to include the behaviour that supports introspection, probing, and profiling, an ideal mechanism to enable this non-functional behaviour is the association of a metatype with that class or object, which implements this introspective behaviour. This can be accomplished by intercepting all accesses to the data and operation of the object or class. The observer may also want to know the result of the original intercepted operation, how long the operation took, what information was on the call stack or in memory when that operation was called, etc. Once these access operations have been intercepted, information contained in the operation request can be made available to the observer or simply logged for later inspection or interpretation. All of these behaviours can be embedded in metatypes. A simple example of how object creation and method invocation can be profiled is shown in figure 3.2.6 below.

Combined with dynamic metatype association, this means that whenever the observer needs more insight into the operation of an object or class, the metatype can be associated with the class or object at runtime. When the observer has enough information, the metatype can be disassociated, and the object or class continues its operation as before. This is especially useful for runtime debugging, error detection and prevention, identification of bottlenecks and hotspots, etc. Of added interest to the Chisel project, this mechanism can be used to probe an object or class and can provide the observer with valuable information about how a class or object works, and how it might be adapted, particularly if the observer did not design or develop the code for that object or class, or does not have access to sufficient documentation about the operation of the class. The observer may have no other way except profiling, introspection, and probing to discover how an object or class works or what it is doing.

```
class ProfileExecuteIntercepted extends ie.tcd.iguana.MExecute {
  public Object execute(Object obj, Object[] args, Method meth ) …  {
      …
      Profiler.log("Starting method "+meth.toString()+", method parameters: ");
      for(int cnt = 0; cnt < args.length; cnt++) {
          Profiler.log ("<"+args[cnt].getClass().getName()+">:"+args[cnt].toString()) ;
      long strtme = Timer.getTime();
      Object result = proceed(obj, args, meth);                    /* execute the method */
      long endme = Timer.getTime();
      Profiler.log("End of method "+meth.toString()+": Operation timing: "+(endme-strtme));
      Profiler.log("Result was <"+result.getClass.getName()+">:"+result.toString());
      …
      return result;
  }
};
class ProfileCreateIntercepted extends ie.tcd.iguana.MCreate {
  public Object create(Constructor cons, Object[] args)  …  {
      …
      Profiler.log("Starting creation of "+cons.getName()+", constructor parameters: ");
      for(int cnt = 0; cnt < args.length; cnt++) {
          Profiler.log ("<"+args[cnt].getClass().getName()+">:"+args[cnt].toString()) ;
      long strtme = Timer.getTime();
      Object result = proceed(cons, args);                         /* create the object */
      long endme = Timer.getTime();
      Profiler.log("End of object creation "+ cons.getName()+": Creation timing: "+(endme-strtme));
      Profiler.log("Result was <"+result.getClass.getName()+">:"+result.toString());
      …
      return result;
  }
};
```

Figure 3.2.6 A example of profiling intercepted object creation and method invocations

There remains an open issue with respect to how the user makes the semantic leap to understand how the application operates, just from the profiling and probing information available about an application and its constituents. It is necessary for the user to interpret the results of probing and profiling information, to establish the roles, behaviours, and modes of operation of the different elements within the system. Without this user's intelligence to understand how to interpret and exploit the probing and profiling data collected, this and other methods of introspection are of limited use. However, depending on the complexity of the system being inspected, this level of understanding should be attainable in many cases. For example, if the user initially knows enough about a system to require it to be probed, profiled, or adapted, that user must already have some understanding of why these requirements have arisen, and so will already possess some degree of semantic knowledge about the system to assist in interpreting profiling data. The Chisel framework provides no high-level support to help the user understand how a system operates or how it can be adapted, but rather provides the mechanisms to allow this probing, profiling, and adaptation to be achieved in a structured manner..

These methods presented to probe and examine third party software may be considered to be only somewhat promising and of an ad-hoc nature, since the use of probing techniques to

reverse engineer [27] and re-engineer black box software is an old and active research field [116], as seen with the integration of commercial-off-the-shelf software modules when developing software [105], the use of legacy systems [8] etc. However, the use of the metatype model is a valid, useful, and powerful technique to support the profiling techniques described or more sophisticated techniques to reverse engineer, refactor, and expose the operation of arbitrary compiled software modules, especially where the source code for that module is unavailable. Once exposed, the internal operations of these modules can be adapted, or alternatively, the interface of these modules can be wrapped in an adaptive manner. In the event that a naïve user wishes to adapt a piece of software, but is unable to locate where the adaptation should be applied, then the Chisel adaptation framework would not be useful for that user

3.2.5 Metatype composition and metatype inheritance

When Schäfer [139] and Redmond [125] introduced metatypes, they also introduced a set of rules for the use and combination of metatypes.

Metatype composition refers to the combination of two metatypes, each possibly providing very different behaviours, so that together they can be associated with objects and classes, thereby associating both of the behavioural changes with that class or object.

A metatype can be inherited from other metatypes. Metatype composition is accomplished using this method. All behaviour changes from both the parent metatype and the new metatype are composed. Metatype behaviours are not specialised by inheritance, they are combined. If it necessary to compose a number of available metatypes, a new metatype is derived from them, thereby composing them. Schäfer [139] asserts that an object can only have one metatype, so these combined behaviours are considered one metatype. Redmond [125] declares that they are considered distinct metatypes. The default mechanism for composing metatypes is that the behaviours are composed sequentially. The order in which metatypes are composed may be an issue with respect to interference and conflict between metatypes. For example, when a metatype providing persistence behaviour is combined with a metatype providing remote access behaviour, the two behaviours may conflict if they are not ordered correctly. Metatype model implementations may support the specialisation of ordering mechanisms, as seen in Iguana/C++.

• The set of metatypes of a class must include the set of metatypes of its superclasses • The set of metatypes of an object, must include the set of metatypes of its class • The metatypes of a class are inherited by subclasses • The metatype of a class is propagated to all its current and future instances

Table 3.2.7 The four rules of metatype use

Table 3.2.7 describes the four rules of metatype use, metatype inheritance relationships, and automatic metatype propagation. These rules also hold for dynamic metatype association with classes and objects. These rules result in a number of effects that may not be obvious from casually reading the rules. A new metatype can be dynamically associated with an object, so long as that metatype is somehow inherited from the metatype(s) associated with the object's class. That metatype can then be disassociated from the object by associating a metatype further down the inheritance chain. However, a metatype cannot be dynamically disassociated from an object, if the object's class or superclasses still has that metatype associated with it. In addition, a metatype cannot be dynamically disassociated from a class, if one of that class's superclasses still has the metatype associated with it.

Since the metatype model assumes the multiple metatypes can be associated with an object or class, with composition performed automatically in Iguana implementations, there is no way to automatically detect if the multiple metatypes are interfering with each other. Research described in [64] discusses the effects of composing metatypes and presents a model to prevent and resolve such conflicts, but such a mechanism does not exist for Iguana/J.

3.2.6 Alternatives to Iguana and reflection for metatypes

Although only implemented in Iguana, the metatype model can be implemented by any system that supports reification of the object model or interception of object behaviours, and implements the four rules of metatype use specified above. A key requirement is support for dynamic metatype association.

A number of the adaptable systems analysed in Chapter 2 provide some support for metatypes, e.g. Kava, Guaraná, MetaXa, AspectWerkz, PROSE, Wool, etc., but none of these frameworks fully support the metatype model at present. However, almost any adaptation framework could be used when implementing a new metatype implementation framework.

A new metatype implementation framework must provide a mechanism for behaviour interception of individual objects. This intercepted behavioural operation must be redirected to a runtime manager, which in turn supports the insertion and ordering of metatype behavioural changes, while maintaining the metatype association rules. This interception can be achieved using source code reprocessing (as seen in Iguana/C++ and the original Aspect/J versions), compiled executable or bytecode reprocessing (as seen in Javassist), load-time reprocessing of code (as seen in Iguana/J, Kava, JMangler), or runtime interception (Java HotSwap, Wool, PROSE, Guaraná, MetaXa, Java Platform Debugger Architecture).

As described above, the Iguana implementations that support metatypes [125, 127, 139] use computational reflection to implement metatypes. Any computational system that maintains a causally connected representation of its own behaviour and supports computation about its own behaviour is, by definition, a behaviourally reflective system. Therefore, any system that implements the metatype model is a reflective system.

However, it is not necessary that reflective programming techniques be used in a system implementing metatypes. As seen from the list of systems above that could implement a metatype system, a number of these systems are not designed to support behavioural reflection, e.g., dynamic AOP systems, source code and executable code re-processors, debug architectures, ad hoc solutions, etc.

Although the Iguana models presented in [58, 59, 125, 126, 139] are language independent, each version of these versions of the Iguana model is implemented and restricted to individual languages. Recent research in language-independent AOP techniques (e.g.,[94]) has prompted ongoing research to reimplement the Iguana model to support a language independent metatype model.

3.2.7 Why use metatypes in the Chisel framework

Dynamic metatype association was chosen as the adaptation mechanism for the Chisel framework because from the very start of this research, one of the objectives of the Chisel research project is to evaluate the usefulness of metatype association as an adaptation technique.

The primary objective of the Chisel project is to design and implement a dynamic adaptation framework that supports completely unanticipated dynamic adaptation. This objective is partially fulfilled by the use of dynamic metatype association, since dynamic metatype association is a mechanism that supports the dynamic adaptation of objects and classes,

irrespective of the type of the target object or class, at any time during the execution of application. This means that behavioural adaptations can be applied to unanticipated locations at unanticipated times.

3.2.8 Consequences of the use of metatypes in the Chisel framework

The primary consequence of the use of metatypes in the Chisel dynamic adaptation framework stems from the limited availability of mechanisms that implement the metatype model. There are currently only two implementations based on prototype versions of the Iguana reflective programming model, i.e., Iguana/C++ and Iguana/J discussed above, neither of which have been widely adopted, used, or tested. The alternative of re-implementing the metatype model, possibly using or adapting a more stable adaptation mechanism, was however outside the scope of this research project. Therefore, Iguana/J was chosen as the adaptation mechanism to be used. Iguana/J was chosen over Iguana/C++ because to its better support for metatypes, its support for the dynamic association of unanticipated metatypes, and because the Java programming language (compared to C++) supports more high-level runtime manipulation of structural and type information, especially in dynamically loaded code.

A consequence of the use of the metatype model, and as an extension the Iguana/J platform, which focuses on adaptation by behavioural reflection, means that there is limited support for structural or architectural adaptation, except through the use of behavioural reflection. Many adaptations, such as the dynamic direct manipulation or inspection of operation code, are not possible in the Chisel framework.

Secondly, the metatype model is tightly bound with the object-oriented (OO) model of software engineering. There is no support to handle non-OO constructs such as components, non-method functions/procedures, aspects, etc. In addition, the metatype model has no automatic support for dealing with groups of objects as a unit, or anonymous classes.

Since metatypes can be disassociated as well as associated, at disassociation time the metatype and its constituent meta objects will be discarded, with all state lost. If it is necessary to maintain state across metatype changes, it is necessary that a separate meta-level data store mechanism be used. In addition, the choice of Iguana/J results in a lack of a structured mechanism to detect and resolve conflict between multiple metatypes associated with a single class or object. Examining such a mechanism would prove a complex and rewarding research topic; however it is outside the scope of this research.

Each of these consequences erodes the generality of Chisel dynamic adaptation framework, since only adaptations supported by Iguana/J can be specified for use by the Chisel framework. However, as can be seen from the literature concerning Iguana and metatypes [19, 59, 64, 124-127, 139] and in particular the Iguana/J literature [125-127] the adaptation domains to which the Chisel framework would be restricted is still broad enough to consider the Chisel framework to be a general-purpose adaptation framework.

Consequences of the use of the Java programming language

In addition, this choice of Iguana/J has also restricted the Chisel framework to use the Java programming language. Ideally, a language independent implementation of the Chisel framework would be preferable. However, this restriction to use Java is in some ways fortuitous since Java is a suitable language with which to implement the Chisel framework. Although Chisel is not specifically designed for implementation in any particular language the Java language provides a good deal of support for runtime manipulation and inspection of both the adaptation software and the target software, and their metadata. The manipulation and inspection of the software and its metadata is particularly difficult in fully compiled languages (for example, C++) since nearly all of the metadata concerning the software is compiled and optimised away. Java provides extensive support for such runtime inspection and manipulation with its extensive reflective API and by its runtime interpretation of the software rather than direct execution. This runtime inspection and manipulation of software and its metadata could have been made even easier by selecting a fully interpreted language such as LISP, where application and adaptation code can be directly treated as data. However, in an effort to be usable in a more widespread manner, and to demonstrate the ability to dynamically adapt compiled code, it was decided that the Chisel framework should not limit itself in such a manner, but rather to use a programming language in widespread commercial use to preserve the generality of the Chisel framework. In this respect, due to the difficulties presented by implementation using C or C++, the Chisel framework implementation would have been restricted to use either Java, or one of the languages provided by the Microsoft® .NET platform [101].

3.2.9 Summary of the metatype model for dynamic adaptation

This section has described in detail the metatype model for dynamic adaptation. This draws mainly from the work on the Iguana project [19, 59, 64, 125, 126, 139] and is based on the metatype models introduced by Schäfer in Iguana/C++ [139] and extended by Redmond in Iguana/J in [125]. The following sections of this chapter and the following chapters continue

the discussion of how the metatype model is incorporated into the Chisel project, with a number of examples being presented to demonstrate how dynamic metatype association can be used as a mechanism to support unanticipated dynamic adaptation of arbitrary compiled applications in a generalised manner.

3.3 The design of the Chisel dynamic adaptation framework

This section provides an overview of the Chisel dynamic adaptation framework. The design of the framework is introduced, along with the major design decisions made during the design. This section also introduces the main constituent parts of the framework and how each of these parts work together during the adaptation process to fulfil the objectives of the Chisel project.

3.3.1 The Chisel dynamic adaptation manager

The Chisel adaptation manager (see later in figure 3.3.1) is the primary constituent of the Chisel dynamic adaptation framework. Since the Chisel project aims to adapt arbitrary applications in a general-purpose manner and apply those adaptations in a manner that may be unanticipated, it is necessary to have some form of runtime controller to initiate and manage these dynamic adaptations. As described in the previous section, this adaptation framework makes use of dynamic metatype association as its adaptation mechanism. The metatype model also requires runtime support to manage how metatypes are associated with base-level objects and classes, in this case, the Iguana/J runtime component is used.

As discussed at the start of this chapter, the Chisel dynamic adaptation framework must meet a set of requirements, and so the Chisel runtime component, the Chisel adaptation manager, must satisfy the runtime aspects of these requirements. There must be support to dynamically identify and specify unanticipated arbitrary adaptation locations at runtime. There must be support for the dynamic specification of adaptation control directives that specify when, where, and how individual unanticipated adaptations will be applied. There must be support for the dynamic identification and loading of adaptation code, in this case metatypes, and support the manipulation of these metatypes.

The adaptation manager cannot contain any a-priori knowledge of any adaptation that may be applied if completely unanticipated specification of adaptations and their management

controls are to be supported. The adaptation manager must allow adaptation directives to be specified at runtime, so support for the unanticipated loading and interpretation of these directives is included. This is handled by the Chisel policy manager. The policy manager is responsible for accepting these specifications and translating them into data that can be used to control the adaptation process. The Chisel rule manager is responsible for taking this control logic data and acting as the runtime adaptation controller, managing the adaptation process as the supplied adaptation logic changes dynamically.

New behaviour can be dynamically specified within a metatype, which may then be compiled, located, and loaded in an unanticipated manner. Once loaded, the metatypes must be applied in a controlled manner to unanticipated target objects or classes within the managed application. This dynamic manipulation and association of metatypes is achieved using the Chisel behaviour manager, and in this respect, it is the Chisel behaviour manager that actually performs the dynamic adaptation by performing dynamic metatype associations.

These adaptations should be performed in a manner where user, application, and environmental context can be used to drive adaptations. Dynamic adaptations are mostly required due to changes in the state, resources, or requirements of the application, user, or operating environment, all of which are classified as context changes in the Chisel framework. This concept of context-awareness is achieved by the Chisel context manager and the Chisel service manager.

The adaptation manager must also support the dynamic identification and use of arbitrary target objects and classes for these adaptations, since any object or class may be adapted. A mechanism is required to identify and find these objects and classes at runtime for adaptation. This is handled by the Chisel service manager, in conjunction with the Chisel named object store. Together the service manager and the named object store must provide access to any application object or class, and allow the state and methods of these objects and classes to be accessed as context lookups.

If adaptation is to be driven by changes in context as described above, it is necessary that these changes are signalled to the adaptation manager to allow the adaptation manager to initiate these adaptations. This is achieved using an event-based model, which is supplied by the Chisel event manager. Since this event-based model is used to signal changes to unanticipated context information in a responsive manner, this event model must support the dynamic specification of events and the logic that controls how they are fired in order to handle this unanticipated context monitoring and signalling.

This modularised design allows the Chisel adaptation manager to be built in a flexible and extensible manner, with each sub-manager assigned a number of responsibilities. An overview of the design of the Chisel adaptation manager is presented in figure 3.3.1 below. The design and operation of each of these sub-managers are described in further detail in the remainder of this chapter.

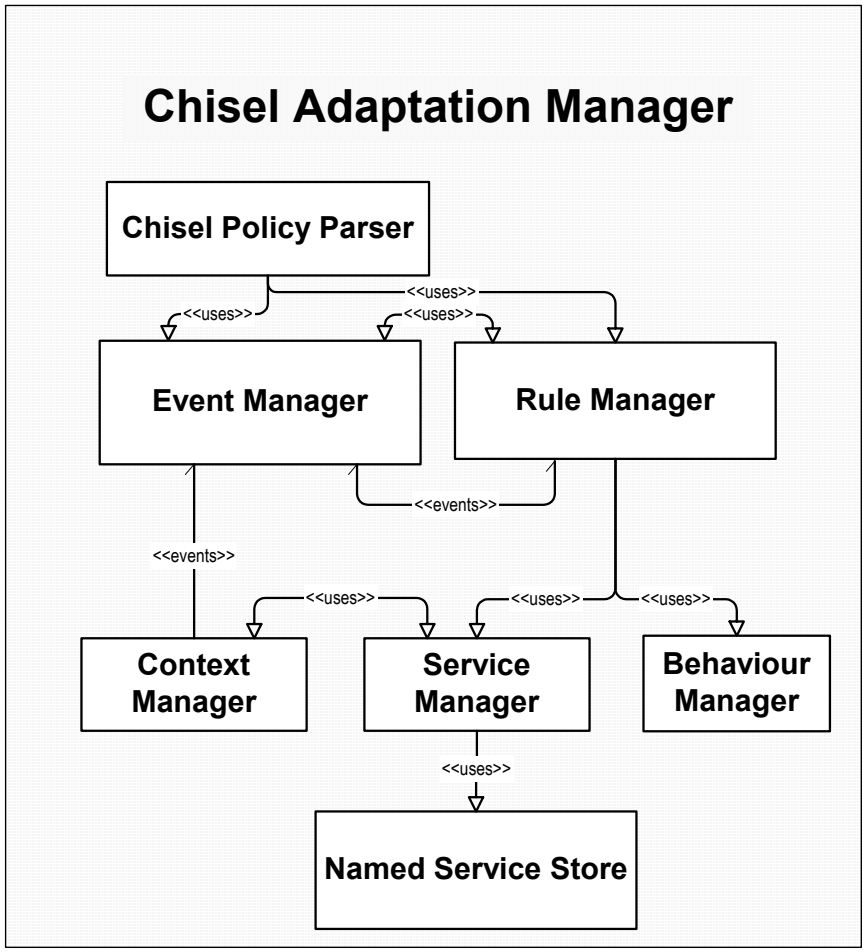

Figure 3.3.1 Overview of the Chisel Adaptation Manager

There are a number of possible mechanisms to implement the linking of the adaptation manager to the target application. One option is a launcher program to initialise the adaptation manager and then start the target application. Another alternative is to link the adaptation manager as a meta object with the main class of the application object so the adaptation manager is initialised as the target application class is loaded. Yet another possible mechanism that can be used, if the application source code is available, is the addition of initialisation code for the adaptation manager in the application source code, and the recompilation of the application code. None of these mechanisms break the requirement to support completely unanticipated dynamic adaptations since only the possible future need

for adaptation support is anticipated, and not any particular adaptation that may be needed later. However, the Chisel framework specifically aims to support the unanticipated dynamic inspection and adaptation of arbitrary compiled applications, so it is preferable to make use of the mechanisms that do not require access to the application source code.

3.3.2 Why event-based adaptation management?

A requirement of the Chisel framework is the support of unanticipated adaptation, in response to unanticipated context change. Here context refers to the state, resources, and requirements of the operating environment, the application, and the user. In order to adapt in response to context change, it is necessary that the context be monitored to track such context changes.

The two main mechanisms used to monitor a dynamic system are event-driven monitoring and time-driven monitoring [97]. Time-driven monitoring loosely relates to periodic probing (polling) of the system being monitored to provide a view of the system's status. Event driven monitoring refers to waiting for a change in the status of the system, then being notified of the change. For the Chisel framework, or any system that requires waiting for contextual values to change, an event-based system is the obvious choice. Polling a value to monitor change requires a trade-off between time spent polling for values that may have not changed, and the sampling rate. Trying to monitor change by polling also presents the opportunity to miss the context change of interest, if the sample rate is too low. Event-driven monitoring leads to "liveness" without the need for constant polling.

The state or status of an object is a representation of the cumulative results of its behaviour, and is represented by the values of all of its properties [16]. An event is defined as an atomic entity that reflects a change in the status of an object [97]. Since this status usually changes continuously, the behaviour of an object is normally observed as a subset of these events that are of significance to the management system and so are only generated when a set of defined conditions are satisfied [97]. This idea of events is therefore central to the design of the Chisel framework since the firing of an event in the Chisel framework can easily be used to signal an interesting change in context.

At design time or when developing any software system, there is no generalised way to anticipate what context change may be of interest. The specification of what constitutes an interesting change must be specified dynamically. A particular requirement of the Chisel framework is the ability to monitor any object, of any type, since neither the designer nor developer of an application can be expected to anticipate at design time what information

may be of interest to the user when controlling unanticipated adaptations at runtime. This is achieved in conjunction with both the Chisel service manager and the Chisel context manager.

The Chisel framework must also be capable of the dynamic definition of events, and the dynamic specification of a set of conditions to specify when this event should be fired. With this support, not only can particular adaptations and their reactive application management be specified in a possibly unanticipated manner, but the triggers used to initiate the application of these adaptations can also be dynamically defined in an unanticipated manner. It should also be possible to manipulate events in an unanticipated manner to allow the combination and filtering of event signals. This is achieved by the use of event declaration policy rules, in conjunction with the Chisel policy manager.

All aspects of event manipulation in the Chisel framework are supported by the Chisel event manager. The event manager is responsible for the dynamic definition and registration of new events, and the specification of when these events are triggered. The event manager also provides the mechanism to allow these events to be manipulated and fired at runtime. In all, the event manager is responsible for supporting the entire operation of the Chisel event model. More information on the operation of the Chisel event manager, what happens when an event fires, and the manipulation of events is the Chisel adaptation framework is presented in the following sections of this chapter, and in the following chapter.

3.3.3 Why have a policy based management approach?

An adaptable system that has its adaptation logic encoded directly into the application cannot operate in a general-purpose manner or adapt in response to unanticipated changes. Chapters 1 and 2 have introduced the use of policy-based adaptation management to decouple the adaptation control support and the adaptation mechanism in adaptable systems.

This need for general-purpose completely unanticipated dynamic adaptation requires some mechanism where adaptation logic can be dynamically specified by the user, and then dynamically interpreted by an adaptation manager at runtime to determine how the system should be adapted. If the system is to support adaptations that remain unanticipated until during execution, there must exist some mechanism to support the dynamic specification and adaptation directives, and support dynamic change of these directives. A prime candidate for this type of dynamic adaptation directive would be the use of a configuration file that contains adaptation directives specified by the user, which the adaptation manager then parses and interprets.

Any adaptation directive must contain a number of necessary elements, including, what to adapt, which adaptation to apply, how or when should it be applied, what constraints may limit the application of the adaptation, etc. The use of adaptation directives, or adaptation rules allow a declarative specification of how the system should be adapted, without the necessity to specify how this adaptation should be accomplished, thereby allowing complete decoupling of the adaptation logic and the adaptation mechanism. When combined with the use of events in the Chisel framework to signal a change in context, and thereby a possible need to adapt the behaviour of the application, these types of adaptation policy rules can be cleanly specified in an Event-Condition-Action (ECA) format, a method commonly used for controlling adaptable reactive systems, as seen in Chapter 2 [17, 33-38, 44, 45, 47, 67, 68, 71, 78, 79, 98, 104, 120, 143].

Adaptation in the Chisel dynamic adaptation framework is driven by adaptation policy rules in this ECA format. These adaptation rules can be in one of two formats, reactive rules and proactive rules. Reactive rules are specified by selecting an event that is fired in response to a context change, an adaptation target, and an adaptation to apply to that target. In order to focus the application of the rule, a set of guard statements may be included in the rule, to be evaluated when the rule is triggered. Proactive rules, which also specify an adaptation target, an adaptation to apply, and a set of conditions, are triggered immediately upon parsing, instead of waiting for an event.

The same rule-based approach can also be used to perform event manipulations instead of adaptations. This would allow events to be dynamically fired to indicate a contextual change in a responsive manner.

The Chisel rule manager is responsible for the interpretation of these rule conditions and the initiation of adaptation or event manipulations if required. The contents of these rules will be completely unanticipated, so the rule manager must act as a general-purpose rule interpreter. The control logic embedded in the adaptation rules may be arbitrarily complicated, so the rule manager must have substantial support to interpret any rule specifications that can embedded in the rule formats. Further information on the types of rule formats that the rule manager must interpret is provided in section 3.6, where the Chisel policy language is introduced. Since these rules may change at runtime, the rule manager must support the dynamic loading and unloading of these rules. The rule manager will also be required to interact with the event manager to enable these rules to be triggered by events. If any runtime interpretation error occurs during the evaluation of any rule, the rule

manager should provide a warning that the adaptation directive is faulty and should be replaced.

Any actual event manipulations are performed by the event manager. Any adaptation is performed by the Chisel behaviour manager, with the Chisel service manager given the responsibility to find the target object or class that will be adapted, and to find any fields or methods of arbitrary objects or classes used as context information within the rule conditions.

3.3.4 How to find the object or class to adapt?

As described, it must be possible to dynamically identify which objects or classes will be adapted since it is a key requirement of the Chisel framework, that it must be possible to allow the location of any particular adaptation to remain unanticipated. Additionally, as a result of the deliberately wide definition of context, and that any runtime state may be considered important contextual information, a requirement exists that arbitrary application classes and objects can act as context information sources. Since there may be no way to identify which objects or classes will be queried, the Chisel framework must allow any object or class to be queried.

The Chisel service manager is responsible for finding the named class or object to query or adapt. Application classes can easily be found with the support of the runtime environment since they can be referred to and looked-up by name at any time. For individual objects, this is more difficult to achieve since objects do not usually have any human readable reference at runtime.

The Chisel named object store is used to associate a name with individual application objects, so that they too can be used in the adaptation policy script. To do this for any object, in an unanticipated manner, is indeed a challenging requirement as it becomes difficult to find the individual object in the runtime execution environment. The Chisel named object store provides an extensive logging and profiling metatype that allows the user to selectively profile application classes to identify individual instances of that class. Once the user has identified a particular object of interest, that object can have a user-friendly names associated with it. Once this name is associated with the object, the name can then be used to refer uniquely to that object via the Chisel named object store. This name can be used in any policy rule, either as an adaptation target itself, or as an information source to direct the operation of the policy rules for other classes, objects, and events.

As discussed in section 3.2.4, when using profiling and logging mechanisms to inspect the operation of executing software, it is a requirement that the user interprets the operation of the application in order to establish the roles and behaviours of individual classes and objects, and so decide which classes should be adapted or which objects should be named in order that they can be adapted. Although the Chisel framework provides a number of mechanisms to assist the user in this regard, the operation of identifying and naming an object remains the ultimate responsibility of the user. A naïve or inexperienced user may have difficulty with such a task, and as such the use of the Chisel framework to probe and inspect a compiled software module would prove difficult. However, it is judged that such a tool would be very useful to an experienced or determined user.

In order to adapt or query an application object or class, the user must know which object or class is to be adapted. If the user needs to search for this object or class using the introspection and monitoring techniques described in section 3.2.4, this often takes some time, as the application objects or classes need to be executing for a time to be profiled and inspected. For this reason, if a user needs to adapt an unknown object or class, it is necessary that the user anticipates the need to find that class or object in order to adapt that object or class. However, this need for anticipation of adaptation does not break the requirements for completely unanticipated dynamic adaptation since there is no requirement for this anticipation to occur before the start of execution, only that this anticipation occurs before the unknown object or class must be adapted.

In the current design, the user must individually name each object of interest. No mechanism currently exists to allow the user to either have an object automatically named, or have a group of objects selected for naming. Research on such a mechanism was seen to be outside the scope of this thesis. In addition, the binding of a name to an object is only valid as long as that object is in context. Once the object becomes unused and garbage collected, or the application exits, the name to object binding will become invalid. Any use of an invalid or out of context name will prompt a warning to the user. However, for the same reason, as long as a named object stays in use, it can be shared by different users.

Once the user has identified which classes needs to be adapted or queried, or has selected which objects to name, these named objects and classes can be retrieved using the Chisel service manager, which uses the Chisel named object store if necessary.

3.3.5 How is the new behaviour applied?

In the Chisel adaptation framework, dynamic metatype association is used as the adaptation mechanism. The Chisel framework must also support the dynamic creation, loading, and identification of arbitrary metatype code at any time during runtime. Section 3.2.3 describes how a named metatype can be defined offline as the target application executes. Iguana/J provides the mechanisms to compile this named metatype in a separate parallel process alongside the executing application. Policy rules can then be dynamically created and passed to the policy manager to have this named metatype associated with an arbitrary application object or class.

When an adaptation operation is required, the Chisel behaviour manager performs that adaptation. The behaviour manager will first request the target class or object from the service manager. The behaviour manager will then load the meta-level class containing the metatype behaviour and associate an instance of it with the target class or object. According to the metatype model, described in section 3.2 above, this metatype association may fail if the rules that govern metatype association are not followed. In this case, the target class or object will keep their current metatype and continue operation. A runtime exception will inform the rule manager, which will then inform the user that the metatype association rules have been broken and the rule that caused the adaptation may need to be replaced.

According to the metatype model, if the adaptation target is a class, all subclasses of that class, and all current and future instances of the target class and its subclasses will also have their metatype changed. For this reason metatype association with classes should be performed with care. When a metatype is associated with a class or object, if that metatype is a parent metatype of the currently associated metatype, its current metatype will be replaced immediately with the new metatype, thereby discarding the old metatype. This mechanism, where another metatype is applied to a class or object, is the mechanism used in the metatype model to remove an applied adaptation, within the constraint that the newly applied metatype is, or is derived from, the metatype of the class of the object, or the metatype applied to any parent class.

3.3.6 Summary of the design and operation of the Chisel dynamic adaptation framework

This section has described the requirements and outline design of the Chisel adaptation manager and its constituent parts and how dynamic adaptation is accomplished in the Chisel

framework. The requirements described in section 3.1 can all be fulfilled with this design. The Iguana/J reflective framework supports the dynamic association of metatypes with arbitrary objects and classes at runtime, thereby allowing the location at which an adaptation is applied to remain unanticipated until runtime. The Chisel service manager supports the dynamic identification and specification of these adaptation locations. The Chisel policy-based specification of adaptation rules allows the location at which an adaptation is applied, when it is applied, and how it applied to remain unanticipated until runtime. The Chisel rule manager then interprets this changeable control logic at runtime. The service manager also supports the dynamic lookup of arbitrary contextual values of any object or class in the application, which is used by the rule manager when evaluating rules containing context-aware logic. The Chisel behaviour manager, using the Iguana/J runtime component, supports the dynamic binding and unbinding of adaptations by supporting dynamic associations of arbitrarily named metatypes, according to the dynamic interpretation of the control logic. Introspection of arbitrary compiled code is supported using the profiling metatype mechanisms described in section 3.2.4, with the Chisel framework supporting the dynamic association and disassociation of these metatype. These profiling techniques are also used by the Chisel named object store.

As shown in Figure 3.3.2, the operating mode of the Chisel adaptation manager requires the cooperation of all constituent parts of the adaptation manager. Firstly an adaptation event is fired, possibly by the context manager in response to a change in a monitored context value, or automatically by the event manager as specified in the triggering specification of a newly defined event, or perhaps fired by the evaluation of an event manipulation rule. As the event is fired, the rule manager is alerted. The rule manager then requests the set of fired events from the event manager. Once received, the rules that are triggered by each event are retrieved. For each rule in this set, the conditions section of the rule is evaluated, in cooperation with the service manager and named object store if named objects or classes are used in the condition set. If the condition set evaluates successfully, the rule evaluation continues. If an event manipulation operation is required, the operation is invoked via the event manager. If an adaptation request is required, an adaptation request is sent to the behaviour manager. The behaviour manager then retrieves the target named object or class to be adapted from the service manager via the named object store if necessary, performs a dynamic metatype association on that adaptation target.

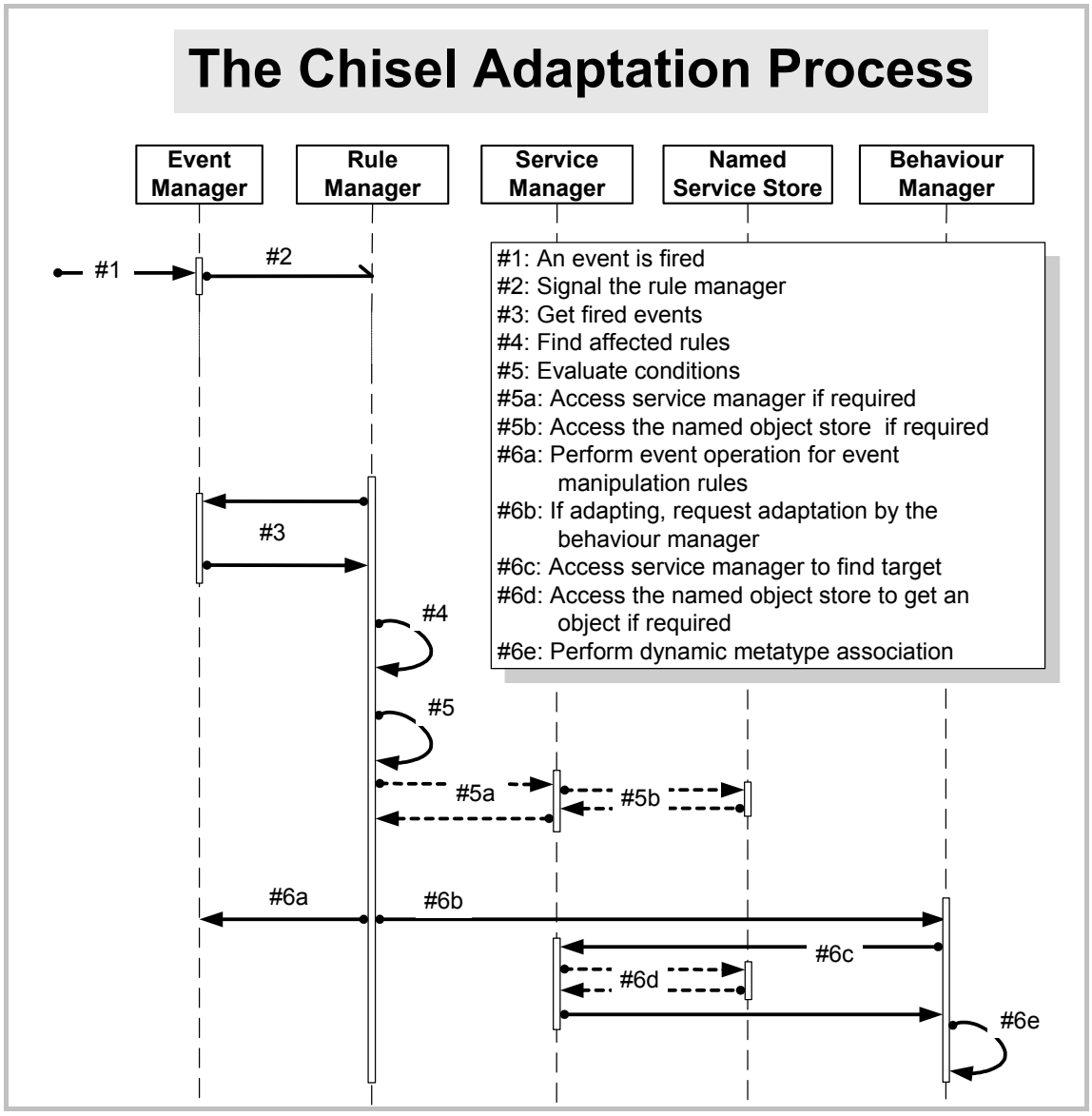

Figure 3.3.2 Overview of the Chisel adaptation process

The following section contains more detail about the Chisel event model, the Chisel context model, and the Chisel policy-based management model. The implementation of each part of this design is provided in the following section, along with in depth information about how these constituent parts of the Chisel dynamic adaptation framework work together to support the general-purpose completely unanticipated dynamic adaptation of arbitrary compiled software in a context aware manner.

3.4 The Chisel event model

Section 3.3.2 above describes the need for an event-based monitoring mechanism in the Chisel adaptation framework, and a set of requirements that must be fulfilled. In particular, this event model must support the dynamic definition of events, and support the dynamic firing of events. The Chisel event system operates in conjunction with the Chisel context system, which is described in the next section, by using events to signal changes in context and communicate the possible need to adapt.

Although the event model used in the Chisel framework must meet the requirements above regarding dynamic definition and naming of event types, and support the dynamic specification of triggering logic, the event model need not otherwise be very complex, since the Chisel adaptation manager is the only consumer of events in the Chisel framework. In addition, in this initial design of the Chisel framework, only context relevant to the local application is supported, so no support for a distributed event model is required. No easily usable event mechanism that fulfilled these requirements without requiring extensive runtime support could be found. For this reason it was necessary to design and implement a custom event mechanism. One resultant design requirement was added, that the event mechanism be of a general-purpose nature to support later enhancement, particularly to add support in a later design for a distributed event model; for these reasons it should be a modular and extensible system.

The Chisel event manager is responsible for all aspects of event handling and management in the Chisel framework, and so provides the Chisel event model. In the Chisel event model, all events have a unique name to allow them to be referred to by name. All event operations are performed in conjunction with the event manager. The Chisel event model does not support a publish-subscribe mechanism, since all objects within the adaptation framework and the application being adapted can be considered publishers of events (event sources). The adaptation manager is the only event subscriber, and subscribes to all events (event sink).

In the Chisel framework, each event type is implemented as one named event object. These individual named event objects are created by the event manager each time a new event type is registered. These event objects are maintained by the event manager, which provides an interface for each named object to be repeatedly fired. Each object representing a named event type must be of, or extend, the generic `ChiselEventObject` type, seen in figure 3.4.1. The class `ChiselEventObject` can also be extended to define event types that

contain extra methods and data fields. New event types can be dynamically created, registered, and deregistered using the event manager's programmatic interface. Each event type can be fired, cleared, enabled, and disabled, again via the event manager. Each instance of the Chisel adaptation manager has just one event manager so access to the event manager is via a single system-wide interface, so objects do not need to maintain a reference to the event manager in order to define or use Chisel events.

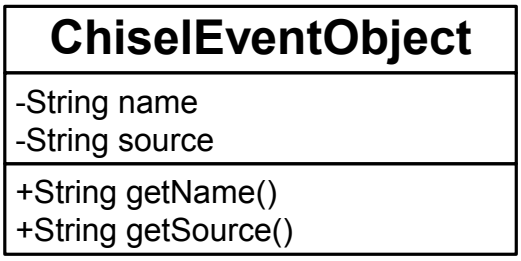

Figure 3.4.1 The ChiselEventObject class

Using the Chisel policy language, events can also be dynamically defined, fired, cleared, enabled, and disabled. Figure 3.4.2 below shows the simplest form of event definition..

```
NEW NetworkDisconnected
```

Figure 3.4.2 Simple example of a dynamic event definition

In figure 3.4.2 above, the event type *NetworkDisconnected* is defined. When parsed, this policy directive will cause a new instance of *ChiselEventObject*, called "NetworkDisconnected" to be created registered with the event manager. For the event objects created programmatically and the event objects created using the policy directives, the unique name associated with each event type can be used in any adaptation policy rule.

```
ON Event1 Event2.FIRE
```

Figure 3.4.3 Simple of an event manipulation rule

Once the event fires, any rule that uses that event as a trigger will be evaluated. As described in section 3.6.3, rules can be defined to fire, clear, enable, and disable any named event, as shown in figure 3.4.3 above, where an event of type *Event2* is fired every time an event of type *Event1* fires. This mechanism provides elementary support for the conditional specification of composite events, and provides a mechanism for event combinations and manipulations such as event filtering etc.

Also described in full in section 3.6, the Chisel policy language design provides a mechanism to specify automatic triggering specifications for events that are defined by policy directives. These automatic triggering specifications can be based on the evaluation of a set of context conditions, or based on a single or periodic time specification. If based on context conditions, the event will be repeatedly fired automatically while the conditions evaluate successfully. The time-based automatic triggers must be defined in a format that is intuitive and easy to understand, so extensive support for the translation and interpretation of multiple timing formats is required. This requirement for time-based automatic triggering of events is inspired by the contributions of the GEM [97] event specification language, where the `at` and `every` operators are used for automatic event triggering. For every operator a time period is specified. For the `at` operator a particular time is specified. Similar support is provided in the design of the Chisel policy language, as described in section 3.6, to allow the user to convey in intuitive and simple formats an abstract concept such as timing, and then have it translated into a strictly defined set of timing specifications. These mechanisms together support the definition and controlled use of events in a dynamic manner, and in a manner that can remain completely unanticipated until during runtime.

Based on this design, the Chisel event model supports the dynamic definition of named event types and the dynamic firing of events, using both a programmatic interface for use in compiled code, and via the Chisel policy manager to fully support the unanticipated definition and use of events. With this design, the Chisel event model satisfies the event model requirements described in section 3.3.2, and satisfies the requirement for completely unanticipated dynamic adaptation, that the specification and use of unanticipated control logic must be supported. A more detailed description of the implementation, operations, and architecture of the Chisel event manager, and how it combines and operates with the other parts of the Chisel framework is provided in the following chapter. Further examples of how dynamically-defined events are specified, their automatic triggering specification, and the specification of event manipulation rules using the Chisel policy language are also presented in section 3.6.3.

3.5 The Chisel context model

As stated in sections 1.1 and 3.1, the primary objective of the Chisel framework is the support of unanticipated adaptation, in response to unanticipated context change, where context refers to the changing state, resources, and requirements of the operating

environment, the application, and the user. This definition of context is designed to be wide ranging to encompass all characteristics of the user, application, and environment that can be measured and queried. This is required since what may be regarded as important context cannot be anticipated when the application being monitored and adapted is designed, or indeed when this adaptation framework is designed. The Chisel context model describes how these context values can be specified, measured, and automatically monitored. This is achieved in two ways.

Firstly, the Chisel service manager supports the dynamic querying of arbitrary fields and methods of arbitrary classes and objects. Since each field and method return value might be considered an unanticipated context variable, this is one mechanism to query context values. This mechanism to query context values is used extensively in the rule directives that specify how the rule manager should initiate adaptations and event manipulations. In particular the use of events to trigger rules, which can contain complex context queries to direct the operation of the rule, and which can then manipulate other events, provides a powerful mechanism to signal the changing context of the user, application, and execution environment. These context change signals can then be used to initiate adaptation rules, which may themselves contain context-aware adaptation control logic. And in addition to this, the rules that direct these context-aware adaptations are themselves dynamically updateable to support both the users changing requirements, and the ability to dynamically specify what should be considered an important context variable. The format of adaptation rules and event manipulation rules is described in the next section.

Secondly, the Chisel context manager supports the programmatic definition of context variables as instances of the `ContextVariable` class, shown in figure 3.5.1. These context variables contain details on a context source, and a context value belonging to that source, whose value can be retrieved. In the default design for the `ContextVariable` class, only fields of classes and named objects can be queried. However, this class can be overridden to extend its operation to support queries of a more complicated nature.

The Chisel context manager also provides an automatic context monitoring mechanism, with support for the automatic firing of events in response to monitored context changes. This is achieved by registering a context variable check condition, and a set of tolerance values for that context variable with the context manager. An event type can be associated with this alert condition, to be fired if the context value moves outside the specified tolerance range. These automatic context alert monitors are instantiations of the `ContextCheckCondition` class shown in figure 3.5.1. These objects are passed to the

Chisel context manager, which performs the monitoring operations and fires the relevant event when the context variable moved outside of tolerance values.

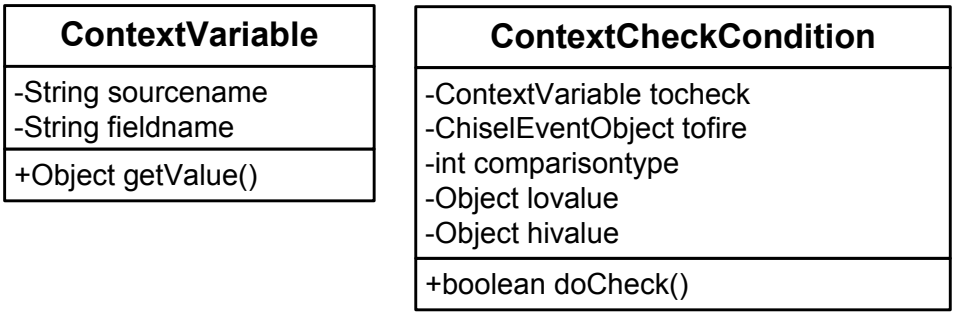

Figure 3.5.1 Data representations of context variables and context alert conditions

The current Chisel policy language design does not support the dynamic specification of `ContextVariable` objects or `ContextCheckCondition` alerts. These can only be defined in source code. This source code may be part of the adapted application, but only if the context variable is anticipated to be important at the design time of the application. Since the Chisel framework is primarily designed to support completely unanticipated dynamic adaptation, this method is not discussed further. The source code containing these context variable definitions and alert definitions can also be contained in metatype code. Since metatypes and their constituent meta objects can be designed and built after the target application has started execution, this is the method favoured. How these context variable definitions and alerts are defined is left to the metatype designer.

The Chisel context model is closely linked with the Chisel event model, since events in the Chisel framework describe context changes. In this respect, the Chisel adaptation framework was designed in a manner that supports unanticipated dynamic definition and manipulation of events, in a context-aware manner. These events are then used to trigger unanticipated dynamic adaptations. So in effect, the Chisel framework supports unanticipated dynamic adaptation in a context-aware manner.

More details on the implementation, architecture, and operation of the Chisel context manager, the Chisel service manager, and the Chisel event manager are given in the following chapter.

3.6 Policy-based management in Chisel

This section describes the use of policy-based management techniques in the Chisel dynamic adaptation framework. The Chisel policy language is also described, which is used to specify adaptation policies for use by the Chisel dynamic adaptation manager to control possibly unanticipated dynamic adaptations in a manner that exploits user-, application-, and environment-specific information.

3.6.1 Why use the Chisel policy language

Despite the requirements for the Chisel adaptation framework to be flexible and support general-purpose use, the type of operations supported by the Chisel framework is quite small. The Chisel framework is designed to allow the user to control the initiation of adaptation operations in a policy based manner. To achieve this, the user must be able to define both proactive and reactive adaptation rules and event manipulation rules. The reactive rules must specify an event that would trigger the rule. These rules could be used to either associate a named metatype with a named class or object, or perform one of a small set of event operations on a named event. These rules should also support the definition of arbitrarily complex condition statements that can be evaluated when the rule is triggered to determine if the adaptation operation or event operation should proceed. The language must also support the definition of new named events and a set of automatic triggers for these events.

Many policy-based management frameworks exist that could have been incorporated into the Chisel framework to provide this support, e.g. Ponder [33-35, 47] or REI [78, 79] as discussed in Chapter 2, however it is apparent that these policy frameworks focus more particularly on security aspects and enterprise level management than on adaptation control. General purpose event-based languages were also considered, particularly Esterel [6, 51] and Jess [137] with their support for Java since this was the programming language to be used in the Chisel framework. However each of these systems require an extensive runtime environment, and for each, the user would be required to learn a new language, just to control adaptation.

It was decided that a fully functional scripting language, or a general-purpose policy-based management framework would be excessively over expressive and heavyweight for the limited requirements of the Chisel framework. It was also felt that the benefits of designing a new lean adaptation policy language would allow easier use of the Chisel framework since

the user would only be required to learn a few simple language constructs rather than a full expressive language.

The Chisel policy language performs as the interface between the Chisel adaptation manager and the user adapting the application software. In order for the Chisel language to be used in a flexible manner, it is necessary that this interface is as easy as possible for the end user to use. It is also necessary that the policy management method used supports the dynamic loading and unloading of new policies to support unanticipated adaptations.

Since the Java programming language was to be used to create the metatype constructs for unanticipated dynamic adaptations, it was also decided that the policy language should resemble Java where possible. This is especially the case in the conditions specification section of rules, where support is required for access to methods and fields of arbitrary objects and classes, calculations and comparisons, and the interpretation of user-specified data values.

The reason for using a human-readable declarative language approach was to allow the user to understand how the adaptation manager controls adaptation. Rather than use a machine-readable language such as XML, the language must display, in an obvious and comprehensible way, what will happen when adaptations are necessary. The most complex section of a rule is the specification of conditions, for which a standard programming language such as Java is ideally suited, since language-level constructs such as operators, the naming of classes, methods, fields etc. are needed. Tool support could have been added to a machine-readable language such as XML to specify and display rule and condition information in a comprehensible manner. However, this type of display of rule logic would still only provide the same experience that a human-readable language would provide without any support. The Chisel policy parser is responsible for taking this human-readable format and converting it to a data format for use by the adaptation manager. This mapping from language to adaptation directives data would still be necessary for a machine-readable language, without added benefit. Any tool support for the display of adaptation logic could still be provided, based on the parsed representation of the control rules.

3.6.2 Alternatives to policy-based management of unanticipated adaptation

There are a number of alternatives to using a rule-based or policy-based approach to control dynamic adaptation. If an adaptable program presents an interface (either graphical or console based) that allows the user to find and identify an adaptation location, and then

provide an unanticipated adaptation for that location, no policy-script is required to drive the adaptation process. This is the approach used in several adaptable systems, some supporting unanticipated adaptation, e.g., the console solution in Iguana/J, the Java HotSwap Client Tool, the Doctor interface in DynamicTAO etc. However, these mechanisms force the user to adopt a "hands-on" approach to adapt the application since, whenever the adaptation is required, the user must be available and ready to initiate the adaptation. Like a policy-based management system, this mechanism requires anticipation of the possible need to adapt, but without anticipating individual adaptations, before the application is started as the interface will require runtime support.

Another alternative is the use of a dynamically loaded agent component to perform and control the adaptation. In this case a software module or agent is created, possibly dynamically, that embeds an adaptation and the logic to install and control that adaptation. That agent is then incorporated into the application so that the agent can then intelligently update the application by deploying its adaptation code. This mechanism is seen in systems such as DynamicTAO, 2K, aspects in Wool, management components in OpenORB, the use of install scripts to update software, or more malicious forms of mobile code seen with computer viruses. These mechanisms are often difficult to use and may pose a security risk to the application. These modules are inflexible and require more anticipation of adaptation since the adaptation logic must be anticipated before the adaptation is deployed and, once the adaptation is compiled, the logic cannot be changed. This mechanism also requires that there exists some mechanism to support the dynamic acceptance and execution of these modules, so again, like policy-based management, this mechanism anticipates the possible future need to support adaptation.

3.6.3 The Chisel policy language

There are three distinct parts to the Chisel policy language. The first part of the language is a means of specifying reactive adaptation rules, as described in figure 3.6.1 below:

ON *EventType NamedClassOrObject* **.** *NewBehaviourMetatype* **IF** *Conditions*

Figure 3.6.1 Format of a reactive behaviour adaptation policy rule

As part of the Chisel event model, every type of event has a unique name. In reactive rules, the name of the triggering event is specified, in place of *EventType* in the rule specification above. When an event of that type is fired, in conjunction with the Chisel event manager, the Chisel rule manager will select the rules triggered by that event to be

evaluated. Adaptation rules are used to specify which metatype should be associated with which application object or class. In the rule specification above, *NamedClassOrObject* is replaced with the name of an arbitrary object or class. If a class, this name is the fully qualified class name, e.g., *java.io.InputSream*. If the adaptation target is an object, the name assigned to that object, as entered in the Chisel named object store, is used. *NewBehaviourMetatype* is replaced with the name of a metatype, e.g., *NullProtocol*, which the behaviour manager should associate with the target object or class. A set of arbitrary conditions can also be specified to constrain the operation of the rule, replacing *Conditions* in the rule specification above. All named objects and classes, all events, and user specified values can be queried and compared to evaluate a series of constraints or guard statements to control the operation of this adaptation policy rule. The Chisel rule manager is responsible for dynamically evaluating the condition section of the rule, which should result in a boolean value. If the conditions successfully evaluate to true, the rule manager will then request that the Chisel behaviour manager perform the adaptation operation.

Reactive event manipulation rules can be used to perform more complex event operations such as event filtering, the format of which is described in figure 3.6.2 below.

ON *EventType* *EventType* **.** *EventOperation* **IF** *Conditions*

Figure 3.6.2 Format of a reactive event manipulation policy rule

The reactive rule above is also triggered by fired events of a named event type. Here, when the event fires, any named event type, including the triggering event type, can have an event operation performed on it. In place of *EventOperation* in the example one of four operations can be specified: *FIRE*, *CLEAR*, *DISABLE*, or *ENABLE*. *FIRE* is used to fire the named event. *CLEAR* is used to clear a named event that may have been already fired, thereby preventing it from triggering the evaluation of a following rule. *DISABLE* disables the event type so that it cannot be fired. *ENABLE* is used to reenable an event type that was previously disabled.

The second part of the Chisel policy language supports the definition of proactive adaptation policy rules. These rules are very similar to the reactive rules described above, but the rule is triggered immediately, and just once, as the policy script is loaded, instead of waiting for a triggering event. The conditions section, the adaptation of managed objects and classes, and the manipulation of named events are all supported as specified in the reactive rules. Figures 3.6.3 and 3.6.4 below show the format used to specify proactive rules.

INITIALLY *ClassOrObject* **.** *NewBehaviourMetatype* **IF** *Conditions*

Figure 3.6.3 Format of a proactive behaviour adaptation policy rule

INITIALLY *EventType* **.** *EventOperation* **IF** *Conditions*

Figure 3.6.4 Format of a proactive event manipulation policy rule

The third part of the policy language provides a mechanism to support the dynamic definition and triggering of new events.

NEW *EventType* *Trigger* *TriggeringSpecification*

Figure 3.6.5 Format of a dynamic event specification rule

Figure 3.6.5 describes how a new event is declared and will be automatically set to fire according to its triggering specification. In the example above, *EventType* is replaced with the name that should be associated with the new event type. A number of different automatic triggering specifications are included in the design of the Chisel policy language. Trigger can be replaced with the keywords *EVERY, AT*, or *WHEN*. If the *EVERY* keyword is used, a time period specification replaces *TriggeringSpecification* in the rule above. If the *AT* keyword is used, a certain time or date is specified to have the event automatically fired at that time or date. If the *WHEN* keyword is used, a condition block similar to the adaptation rules and event manipulation rules above is specified. In this case, the event will be repeatedly fired when the condition evaluates to true.

Each of these parts of the Chisel policy language is described in more detail in the following sections.

Specification of new events

This mechanism is used to define new events that could not have been anticipated at design time, possibly because the event indicates a condition that was unknown at design time, or one that was not anticipated to be of importance by the application designer. In order to support adaptation in response to unanticipated stimuli, the Chisel policy language supports the dynamic declaration of new triggering events.

Figure 3.6.6 shows how a new event called *UnluckyDay* is defined dynamically, and will be fired every Friday. If the trigger *EVERY* is used, a set of time specifications is specified by means of a time interval. If the trigger *AT* is used, a single time point or date is specified. As with the GEM event specification model, partial timing specifications result in the use of default values for unspecified parts of a timing specification. If no seconds values are

specified, the value *0* is assumed. If no day is specified, the current day is assumed, etc. For example, "*Friday*" in the rule specification above, results in the automatic triggering of the event every Friday at *00:00:00*, so the event *UnluckyDay* will be next fired immediately after midnight, in the morning of the next occurring Friday. The event will then be fired at the same time every following Friday. Along with the names of each day, the other descriptive language constant values supported are the names of the months, *midday* and *midnight*, *day*, *week*, *month*, *hour*, and *minute* are provided in the framework design. Here the *minute* keyword represents *60* seconds, the *hour* keyword represents *3600* seconds, *week* refers to *604800* seconds, i.e., the number of seconds in a week, the *midnight* keyword refers to the time *00:00:00* etc. The full specification of a time point or date is in the form *dd/MM/yy@hh:mm:ss* where *dd* is the day of the month, *MM* is the month number, *yy* is the abbreviated year number, *hh* is the hour, *mm* is the minute, and *ss* is the seconds value. As stated, if the time is omitted, midnight is assumed. If the date is omitted, today is assumed. If either the date or the time is omitted, the *@* symbol can also be omitted. Individual parts of the time/date specification can also be omitted values, but only in order from right-to-left. For example, *dd/MM* can be specified without the year value since this year will be assumed, *hh* can be specified for a time without the minute or second value since *0* will be assumed for both. The full specification of a time interval must be given in seconds, or using one of the language constants described above. Arithmetic operators are currently not supported in this design of the Chisel language.

```
NEW UnluckyDay EVERY Friday
```

Figure 3.6.6 Example dynamic event definition

Standard conditions, as used in proactive and reactive rules and described in the following section, are evaluated for use with the *WHEN* trigger. These conditions must be continuously checked in the background, and when these conditions evaluate successfully, the event is fired via the event manager. An alternative approach to using the *WHEN* trigger in a rule specification, is to use the Chisel context manager in a programmatic manner to monitor a context-variable condition and fire the event when that variable goes outside specified bounds, as described in section 3.5.

As discussed earlier in this chapter this ability to define new event types is crucial to the design of the Chisel framework. This mechanism allows the unanticipated definition of what context information is of importance to the adaptation management, and how the Chisel

event model can be used can represent possibly changing context conditions with events fired to alert the adaptation management of the occurrence of these context conditions.

Specifying rule conditions

The rule conditions specification block, used in reactive rules, proactive rules, and *WHEN* automatic event triggers, contains arbitrary code that is interpreted at runtime by the Chisel rule manager. The final result of the evaluation of this code must be a boolean value, to indicate if the adaptation operation or event should proceed or not. This code is specified in standard Java notation. Conditions can contain calls to any field or method, of any class or named object, and constant string, integer, floating point numbers, boolean, and character values.

Strings are specified using quotation marks " ", integers as ordinary numbers, floating point numbers as numbers with a decimal point, boolean values as *true* or *false*, and characters using single quotation marks ' '.

Field accesses are specified in the format: *NamedObjectOrClass.FieldName* where *NamedObjectOrClass* can be replaced by any fully qualified class name or by a named object as registered in the named object store, and *FieldName* can be replaced by the name of a declared public field for that class or objects. Method calls are specified like Java in the format: *NamedObjectOrClass.MethodName(Parameters,…)* with *NamedObjectOrClass* replaced by any fully qualified class name or by a named object as registered in the named object store. *MethodName* is replaced by the name of the declared method to be called in that class or object. Parameters can be replaced by one or more parameters for that method call. Other classes, objects, method return values, field access, and user defined constant values can be used as parameters in a recursive manner. Object and classes returned by field accesses and method invocations can also by used as the target object for other field access and method calls, using the following format: *(NamedObjectOrClass.MethodName(Parameters,…)).FieldName* with parentheses used to separate the different parts of the call.

Standard arithmetic operators, including subtraction ("-"), multiplication ("*"), addition ("+"), division ("/"), and unary boolean negation ("!") are included in the design. Standard boolean comparison operators, including equals ("=="), not equals ("!="), greater than (">" and ">="), less than ("<" and "<="), with multiple conditions combined using the boolean combination operators AND ("&&") and OR ("||") are also supplied. The conditions block can be arbitrarily complex provided each block is enclosed in brackets. If the condition

section is omitted, the condition "`IF true`" is presumed, so the condition will always be satisfied.

The most difficult task of the Chisel rule manager is the dynamic interpretation of these arbitrarily complex condition specifications to determine if the rule's adaptation operation or event manipulation operation should proceed. Full details on the implementation of the rule manager are given in the following chapter. Examples of rule condition blocks are given in the example rules used throughout this thesis.

Specification of new reactive rules

Reactive rules are the mechanism that the Chisel framework uses in order to perform user defined event manipulations and initiate adaptations in response to context changes. This section introduces a series of examples to illustrate how these reactive rules are specified.

```
ON WirelessNetworkDisconnect   NetworkConnectionService.WiredConnectionBehaviour
        IF (NetworkConnectionService.WiredConnectionAvailable  == TRUE  &&
                WirelessNetworkDisconnect.IsTemporary == FALSE  )  ||
            (UserPreferences.getPreferredComms()).compareTo("Wireless") != 0
```

Figure 3.6.7 Example reactive adaptation policy rule

In figure 3.6.7, *`WirelessNetworkDisconnect`* is a named event type, for example, fired by a network resource monitor or by the Chisel context manager to signify that a wireless connection may have become disconnected. For this example, if the *`WirelessNetworkDisconnect`* event is fired, this rule will be triggered. The rule manager will then request that the behaviour manager associates the named *`WiredConnectionBehaviour`* metatype to the object or class named *`NetworkConnectionService`*, but only if the condition block evaluates to true. As can be seen, the fields and methods specified in the fired event object, *`WirelessNetworkDisconnect`*, can also be used in the conditions, as well as method calls and field accesses of arbitrary classes and named objects. However, as described in section 3.4, this is only possible for event types that extend the *`ChiselEventObject`* class to include additional field or methods.

In figure 3.6.8, *`UnluckyDay`* is an event, fired by the example rule given in figure 3.6.6 above. When it fires, event *`ReallyUnluckyDay`* will also be fired if it is the 13th day of the month when the rule is triggered, as determined by the Java code specified in the condition block. Even though a rule of this nature may seem contrived and have little

anticipated practical use, this rule shows that particular user requirements and resources cannot be anticipated by the software's original designers and developers.

```
ON UnluckyDay  ReallyUnluckyDay.FIRE
        IF (((java.util.Calendar).getInstance()).getTime()).getDate() == 13
```

Figure 3.6.8 Example reactive event manipulation policy rule

Specification of proactive rules

Chisel proactive adaptation policy rules are very similar to reactive rules, except they are not triggered by events but rather they triggered immediately, and just once, when the policy script is loaded and interpreted. Proactive rules are used where the user requires that an adaptation is applied immediately instead of waiting for a context change. This mechanism is appropriate where the need for adaptation has already arisen and is being addressed by providing a new policy script for the Chisel framework. Again, proactive rules can contain arbitrarily complex conditions in a manner similar to the reactive policy rules.

```
INITIALLY MyNamedApplicationObject1 . ChiselVerboseOperation
```

Figure 3.6.9 Example proactive adaptation policy rule

Figure 3.6.9 shows how a metatype, *ChiselVerboseOperation* will be applied to the named object *MyNamedApplicationObject1* immediately when the policy rule has been parsed. This new behaviour will be unequivocally applied since the condition "*IF true*" is assumed, as stated above. The Chisel event manager will make use of the Chisel named object store to find the object that corresponds to the name *MyNamedApplicationObject1*. This demonstrates that useful and recognisable object names, applied to any object via the Chisel service manager and the Chisel named object store, can be used in Chisel policy rules. More information on the Chisel named object store is presented later in the following chapters, including, how names can be applied to arbitrary objects, and how these names are used to find the object for use in rules as shown in the example above.

Passing parameters to metatypes

The current design of the Chisel policy language does not support the specification of metatype parameters. Although this powerful technique of passing parameter data when metatypes are associated, allows metatypes to be more flexible and reusable, this version of the Chisel programming language does not support metatype parameters, since this would

affect the readability and simplicity of the language. However, parameters can be embedded directly in the metatype code and then dynamically recompiled offline. This current design is not ideal, but the support for metatype parameters was not seen as a core requirement of the Chisel policy language. Metatype parameters can also be specified if metatype associations are performed from within other metatypes or other arbitrary execution code.

3.6.4 Summary of policy-based management in the Chisel architecture

This section has described the use of policy-based management techniques to control adaptation in the Chisel framework. The use of this policy-based control model allows the clean decoupling of adaptation logic from the adaptation mechanism used by the Chisel framework. The dynamic loading and interpretation of policy directives can also be used to support the management of new unanticipated adaptations, by allowing those new adaptations to be referred to dynamically, along with where they should be applied and what management logic should be used to control how and when those adaptations are applied.

This section has also described the Chisel policy language, and how it can be used to define new events, specify proactive and reactive event manipulation rules and adaptation rules, and how complex control logic can focus and direct the operation of these rules. It is important that the Chisel policy language is easy to use and easy to understand, allowing users to manipulate the target application software in ways that cannot be anticipated by the software designers and developers.

The Chisel policy manager supports the dynamic loading of policy scripts containing a set of rules defined using the Chisel policy language. The Chisel rule manager is responsible for controlling the adaptation process by processing these rules, either when the script is loaded for proactive rule, or in response to events for reactive rules. As described in the Chisel event model and context model, these events signify a change in the state, resources, or requirements of the user, application, or execution environment. This mechanism allows adaptations to proceed in a controlled, reactive, context-aware manner.

The use of the Chisel policy language to specify possibly unanticipated adaptation control logic, and the Chisel adaptation manager to have those specifications parsed and interpreted at runtime, is the key design characteristic that allows the Chisel framework to support completely unanticipated dynamic adaptation in a context-aware manner.

However, the use of the Chisel policy language does restrict the generality of the Chisel framework to support general-purpose adaptations. The Chisel policy language, and the Chisel framework, is specifically designed to allow new rules to be specified dynamically, with adaptation triggered by events which can be dynamically defined and manipulated, and with adaptation initiations and event manipulations constrained by rule conditions specified in a manner as close as possible to a general-purpose programming language. In this respect, every effort has been made to maintain the generality of the Chisel framework. Only with further research and design could alternative or supplementary dynamic adaptation control mechanisms be leveraged to enhance the generality of the framework.

3.7 How context-aware general-purpose completely unanticipated dynamic adaptation is achieved

This chapter has described the design of the Chisel dynamic adaptation framework. As stated the primary goal of the Chisel framework is to support completely unanticipated dynamic adaptation. This can be achieved if the all aspects of individual adaptations remain unanticipated until during the execution of the application being adapted. To achieve this the contents, location, timing, and control logic of the adaptation must remain unanticipated.

3.7.1 Unanticipated adaptation contents achieved

Since the dynamic association of metatypes is the adaptation mechanism used by the Chisel adaptation framework, it is necessary that metatypes can be created in a dynamic manner at any time as the target application is running. To support unanticipated behaviour adaptation, there can be no a-priori knowledge about what the dynamically created metatype will do, or how the adaptive behaviour will be achieved.

The Chisel framework supports this dynamic definition and use of metatypes by using a runtime lookup mechanism to find the metatype, only after it has been referred to by an adaptation policy rule, which may itself have been dynamically defined in an unanticipated manner. As discussed in section 3.2, this design of the Chisel framework makes use of the Iguana/J reflective architecture to provide the mechanism to support dynamic metatype association. At any time as the application to be adapted is executing, in the background a user can dynamically define a new metatype for unanticipated use by the adaptation manager. This means specifying the parts of the object model to reify, then dynamically declaring, developing, and compiling behavioural meta objects for these reifications, and

then combining these meta objects to form a meta-level class embedding the metatype behaviour, all as the target application is executing. This metatype can contain any adaptive behaviour that can be supported using the metatype model.

Since the metatype is implemented as a standard Java class, it can be dynamically loaded using Java dynamic classloading techniques. Therefore, this new and unanticipated metatype class can then be dynamically loaded by the Chisel framework's behaviour manager and associated with any class or object in the application, thereby adapting either the functional or non-functional behaviours of that class or object. If necessary, the Chisel framework can be used to pause the execution of the application while waiting for the metatype to be compiled.

3.7.2 Unanticipated adaptation locations achieved

In the Chisel framework, dynamic metatype association can be used to dynamically adapt any class or named object, which acts as the location. If the object or class already has a metatype associated with it, it will be replaced by the new metatype, unless the new metatype is derived in some manner from the original metatype, in which case it will already contain the behaviour of the old metatype. Also, if a metatype is to be associated with an object, that metatype must be derived from the metatype associated with the object's class. The same restriction applies to the dynamic association of metatypes with classes; the metatype must be derived from the metatypes of its superclasses. However, all objects and classes initially have a default null protocol associated with them, and all metatypes are derived from this null protocol.

The design and operation of the Chisel service manager and the Chisel named object store has been discussed in a number of sections of this chapter. The Chisel service manager is responsible for finding the target object or classes for adaptation, and so any object or class that can be found at runtime can be dynamically adapted. For any dynamically applied adaptation, if its location is to remain unanticipated, there must exist some mechanism to have this location information specified at runtime. From this location specification, it must also be possible to find one distinct target object or class to adapt. The location specification is made in the adaptation policy script, which may be changed at any time during the execution of the target application. It is then the responsibility of the Chisel service manager to find this location at which to apply the adaptation. If the adaptation location is a named object, this name must first have been associated with an individual object using the Chisel object store interface. These mechanisms combined mean that the location of any adaptation

can remain unanticipated until during the execution of the target application, and until the time the control logic for that adaptation is interpreted by the Chisel policy manager. More information on the implementation of the Chisel service manager, the Chisel object store, and the Chisel policy manager is presented in the next chapter.

3.7.3 Unanticipated adaptation control logic achieved

The Chisel policy language, described in section 3.6, in conjunction with the Chisel policy manager allow users to dynamically decide what logic should guide the process by which an arbitrary adaptation is applied to an arbitrary target. Any time the user wishes to update the policy script, the Chisel policy parser will accept the new script, parse the control directives and pass the rules to the Chisel rule manager and the Chisel event manager for processing. There are no requirements to control what the user should have in these policy rules. The user can refer to any context variable, any field or method in any class or named object, any event, or can define their own requirements when creating the condition set for a proactive or reactive rule. The user can dictate that the adaptation be applied immediately as a proactive adaptation directive, or choose any event as the trigger for a reactive adaptation rule. The user can also dynamically specify the control logic of when this event will fire. These supports combined allow the user to define arbitrarily complex rules about how individual adaptation should be applied, and all in an unanticipated manner, as the target application runs.

3.7.4 Unanticipated adaptation timings achieved

The time that an adaptation is applied is strongly related to the adaptation logic declared in the adaptation policy rules. In the Chisel adaptation framework, an unanticipated adaptation can be applied as the result of a proactive rule or a reactive rule.

A proactive rule will result in the application of an adaptation immediately as the policy script is interpreted, but only if the conditions specified in the rule are satisfied. This means that the timing of these adaptations is bound to the time that the script is changed. However, as described above, this script can be changed in an unanticipated manner, so the time at which these proactive adaptation rules are applied, can remain unanticipated until during runtime, i.e., until the user decides that an adaptation requirement exists, a therefore uploads a new policy script for the policy manager to interpret and then apply the adaptation.

A reactive rule will result in the application of the adaptation at an even later stage. A reactive rule will only be triggered if the event used as the rule trigger is fired. Only then

will the condition section of the reactive rule be evaluated. The condition section can contain arbitrarily complex combinations of operations, arbitrary context lookups, and arbitrary constant values. Only if these conditions evaluate successfully and result in a true boolean value will the adaptation be applied. Again the event that triggered the rule can be fired as a result of another arbitrary policy rule, by the Chisel context manager, by the event manager as part of an automatically triggered event, or indeed by any part of the application or adaptation manager that makes use of the event manager programmatically. Combined, this means that the time at which adaptations caused by reactive adaptation rules are applied, is indeed unanticipated.

3.7.5 General-purpose dynamic software inspection and adaptation achieved

As seen, each of the requirements for the support of completely unanticipated dynamic adaptation can be fulfilled by the Chisel dynamic adaptation framework, thereby meeting the primary goal of the Chisel project. However, it should be possible to use this framework to adapt any general-purpose application and not just those designed especially with the possible need for adaptation anticipated.

Metatype association can be used to adapt any object, since all state and method accesses, both into and out of the object, can be intercepted. There are no requirements about how the object's source code was written, and indeed no requirement for access to the source code of the object. All that is required is a reference to the object, in order to associate any arbitrary metatype with that object.

If the functional behaviours of the object are to be adapted, then it may be necessary to have some form of information about how those functional behaviours are implemented. This is difficult without adequate documentation and if the source code for the object is unavailable. However, as described in section 3.2.4, Chisel supports the dynamic attachment of introspective and probing metatypes with the object to facilitate attempts to deduce as much information as possible about the internal implementation of the object. Although this mechanism is likely to produce a significant amount of useful information about how the object works, this mechanism is restricted by the lack of support in the metatype model to directly access the functional code of the target object. However, the separate use of a number of other techniques, such as decompilation, the use of bytecode manipulation tools such as Javassist [24, 26], BCEL library [32], or the use of debug tools to extract any debug

symbolic information remaining in the code would likely lead to more information about the operation of the object.

If the non-functional behaviours of an object are to be adapted, there is often less need for access to the implementation details of the object, since these behaviours can often be applied by completely wrapping or redirecting the operations of the object, without regard to the type of the object, what the functional behaviours of the object may be, or how these behaviours are achieved. However, this cannot always be the case since some non-functional behaviours require tight integration with the internal operations of the object, e.g., memory management or debugging.

Since the Chisel framework can be used to dynamically adapt software that was designed and developed without any a-priori knowledge that these adaptation would be performed, the Chisel framework satisfies the requirement that arbitrary objects and classes, of arbitrary applications, can be dynamically adapted in a general-purpose manner.

By making use of the inspection mechanisms described in section 3.2.4, the Chisel framework can be used to dynamically inspect and probe the operation of any general-purpose compiled software application, providing vital information about how that module can be adapted and tailored, and all without requiring any access to the application's source code. Therefore, these inspection mechanisms can be used to open up software modules that have been designed according to the black box principle of software engineering.

However, the generality of the Chisel dynamic adaptation framework is restricted by it use of Iguana/J as its dynamic adaptation mechanism, and by the degree of control provided by the Chisel policy language to drive the adaptation process.

3.7.6 Context-aware dynamic adaptation achieved

As described throughout this chapter, the Chisel context model uses a very wide-ranging description of context to include all resources, requirements, and state of the user, application, and execution environment. The main reason why this definition is so wide-ranging is that there is no way to anticipate what would be considered context and what would not. Different applications and different users would all have different context requirements and would consider definitions of context ranging from very specialised and focused, to very wide-ranging and vague. The Chisel framework is designed to act as a general-purpose context-aware adaptation framework, so the widest definition of context must be used.

Since there is no way to anticipate or determine prior to execution, what context values the user may require access to during runtime, the Chisel context model supports the dynamic specification of what should be monitored, whether as a direct value lookup, or using events to signal changing context values. This is achieved in the Chisel framework by supporting the dynamic definition of event types, and the dynamic specification of rules to be evaluated based on those event types. The rules can also query any public field or method of any arbitrary named object or class at runtime, by making use of the service manager.

For context lookups that can be anticipated prior to runtime and so inserted in application code, or lookups that are anticipated as metatypes are written and so embedded in metatype code, the Chisel context manager can be used to register, and automatically monitor, named context variables. These context monitors then fire events when the context variable changes in an interesting manner.

So once events signalling interesting changes in context are being fired, the Chisel policy language can be used to specify rules for context-aware dynamic metatype associations in a completely unanticipated manner, with further context lookups incorporated into the condition block of these rules.

Unanticipated location (*where*)	• Ability to dynamically create adaptation hook locations at runtime *or*, Have already inserted a large set of hook locations, suitable for any adaptation • Ability to identify and refer to adaptation location at runtime	✓
Unanticipated control (*how*)	• Ability to dynamically specify adaptation control logic, and have this control logic interpreted at runtime • Ability to include arbitrary context monitoring in this control logic, and support the dynamic specification what should be monitored.	✓
Unanticipated timing (*when*)	• Ability to bind and unbind adaptations at specified points at any time during execution, in a timely manner	✓
Unanticipated adaptation contents (*what*)	• Ability to dynamically create arbitrary executable code at runtime • Ability to load this newly created code refer to this executable code after it is loaded	✓
Introspection	• Ability to dynamically inspect, probe, and profile the operation of arbitrary compiled software at runtime • Ability to perform this introspection without access to the source code of the software • Ability to perform this introspection in an unanticipated manner	✓
Metatypes	• Make use of metatypes as a dynamic adaptation mechanism, to demonstrate their abilities and usefulness	✓

Table 3.7.1 Meeting requirements in the Chisel dynamic adaptation framework

3.8 Conclusion

This chapter described in detail the requirements, concept, and design of the Chisel dynamic adaptation framework. The metatype adaptation model and the use of dynamic metatype association as a dynamic adaptation mechanism are also presented.

An overview of the design and operation of the Chisel adaptation framework first introduced the adaptation manager and its constituent sub-managers: the event manager, the rule manager, the behaviour manager, the service manager, and the context manager. The design of each of these managers was then described in detail, with their key operations introduced.

The Chisel policy language, used to dynamically drive the Chisel adaptation manager was also described in detail. The use of the Chisel language to specify rules to manage the application of dynamic adaptations in a proactive or reactive manner, and to manage the dynamic definition and manipulation of events for use in these reactive rules, was also described.

How the Chisel framework can be used to perform completely unanticipated dynamic adaptation is also presented, by describing how all aspects of individual adaptation can be specified after the target application has started execution.

The next chapter describes in depth a prototype implementation of the Chisel adaptation framework and the implementation of the Chisel policy parser. Chapter 4 also describes in detail how the Chisel event model and the Chisel context model are achieved.

Chapter 4

CHISEL FRAMEWORK IMPLEMENTATION

This chapter describes in detail a prototype implementation of the Chisel dynamic adaptation framework that conforms to the design presented in Chapter 3.

The majority of the chapter is made up of detailed descriptions of how the individual parts of the Chisel adaptation manager are implemented and how they operate. How these parts operate together to provide the operation of the adaptation manager is then discussed. Finally, this chapter shows how the Chisel adaptation manager is initialised and attached to an application to be monitored or adapted.

4.1 Overview

The Chisel dynamic adaptation framework is implemented based on the design given in the Chapter 3 to support completely unanticipated dynamic adaptation in a context-aware manner. As described, the Chisel dynamic adaptation manager is composed of a number of sub-managers, where the adaptation mechanism, the adaptation logic interpretation, and context monitoring are all decoupled from each other. The adaptation logic, which is specified in adaptation policy rules written using the Chisel policy language, is interpreted by the Chisel policy parser and then handed to the Chisel rule manager for enforcement. The Chisel service manager is responsible for the identification and specification of target objects, and classes, and is used to retrieve those named objects and classes as adaptation targets or for use when conditions are being evaluated as part of the evaluation of adaptation rules. The Chisel named object store is also provided for use by the service manager to support the mapping of recognisable names to individual objects. The Chisel event manager is responsible for managing events, including event creation, specification, triggering, and

monitoring. When events are fired, the Chisel rule manager is signalled so that it can then request a list of all fired events. The rule manager selects all rules triggered by those events and then, in cooperation with the service manager, dynamically evaluates any conditions for those affected rules. If these conditions evaluate to `true`, the Chisel behaviour manager uses the service manager to find the adaptation's target object or class, and applies the specified adaptation to it, dynamically loading the metatype class that contains the adaptation if required.

The following sections provide a more in-depth discussion of the operation of each of these constituent parts of the Chisel adaptation manager.

4.2 Event manager

As described in Chapter 3, the Chisel event manager is responsible for providing the Chisel event model, so the event manager deals with all aspects of the event system in the Chisel adaptation manager. The main operations of the Chisel event manager are shown in figure 4.2.1 below. The event manager maintains a set of registered named event types, where each event type is implemented as a single named instance of the `ChiselEventObject` class. The event manager provides an interface to allow each event to repeatedly be fired, cleared, enabled, and disabled. The event manager is also responsible for retrieving method return values and data field values from named event objects that are implemented by extending the `ChiselEventObject` class, and so may contain custom fields and methods. These fields and methods can be used in the conditions block of policy rules, as described in section 4.3. The event manager is also used by the Chisel policy compiler when a policy script is validated while being loaded and interpreted, to verify that each event used is first registered.

Each new event type, represented as an instance of the `ChiselEventObject` class or subclass, is registered with the Chisel event manager. As stated, each event type has a unique name, which can be used at any time to refer to the particular event object that represents that event type. The event manager maintains this mapping of names to event objects. When an event is fired, the object representing that event type is copied into a separate data set containing only fired events. When an event is cleared, it is removed from this set. As events are fired or cleared, the rule manager is also alerted. When the rule manager requests the set of fired events, this set of fired events is copied to the rule manager

for processing, and then this set of fired events is cleared. If an event is disabled, it cannot be fired. Disabled events can be re-enabled at any time.

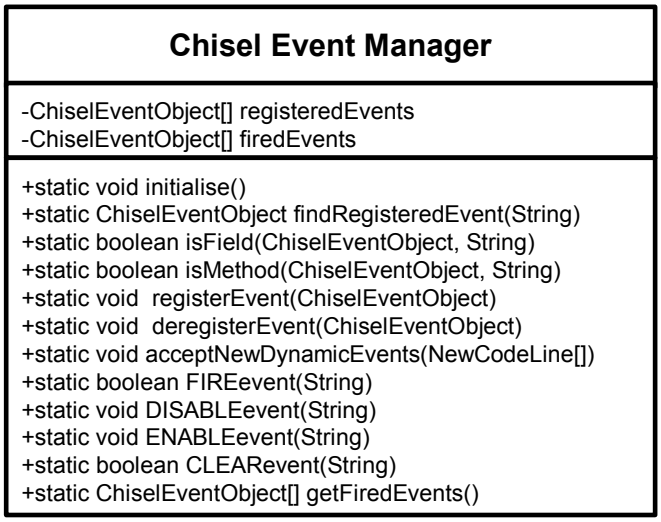

Figure 4.2.1 Key functions of the Chisel Event Manager

If an event is defined in a policy specification using the *NEW* keyword, that event definition is first parsed by the Chisel policy manager, and then passed to the event manager as a *NewCodeLine* object, (see figure 4.8.2). When the specification is first parsed, a new *ChiselEventObject* is dynamically created for that event, and then registered with the event manager. From then on, that newly defined event can be used in the same way as any other event. If an event manipulation policy rule causes an event to be fired, cleared, enabled, or disabled, that operation is translated into a method invocation on the event manager to perform that operation. When the name is passed to the event manager, the event manager first finds the event type (i.e., the corresponding *ChiselEventObject*) for that name, and then performs that operation. *FIREEvent* fires the event by copying that event to the fired events set. *CLEAREvent* removes the event from the set of fired events if present. *DISABLEEvent* first clears the event, and then sets the event status to disabled so that it cannot be fired. *ENABLEEvent* clears the disabled status of the event if set, enabling the event to be fired.

Also described in Chapter 3, the Chisel event model is designed to support the automatic firing of events according to automatic triggering specifications. This automatic triggering is only designed for events defined dynamically using *NEW* in policy specification scripts, and not for event types that are created in source code and are registered programmatically with the event manager. The *AT* keyword is designed to cause the newly defined event to be fired

at a certain future time. The *EVERY* keyword is used to cause events to be fired according to a periodic time specification. The *WHEN* keyword allows a set of conditions to be specified, whereby the event will be repeatedly fired while those conditions are true.

In the current implementation, only the operation of the *WHEN* keyword is implemented. The *AT* and *EVERY* operations are designed, but not currently implemented. The mechanisms to support these time-based automatic triggers will be completed in the immediate future.

The event manager defines internally the *ChiselEveryTimeEvent* event type, which remains in a permanently fired state, however, this event type cannot be used from outside of the Chisel adaptation manager, and so cannot be used as part of a user defined rule. The automatic triggering specifications in *WHEN* rules are internally translated into reactive event-firing rules, with the specified *WHEN* conditions forming the conditions block for that rule, with the *ChiselEveryTimeEvent* event used as the triggering event. This reactive event manipulation policy rule is then passed to the rule manager for evaluation along with the set of other reactive rules evaluated by the rule manager. Since these *WHEN* rules are always triggered, these are evaluated during each rule evaluation pass by the rule manager

For demonstration and testing purposes, a dialog based user interface is provided that allows direct manipulation of events, as shown in figure 4.2.2. This graphical interface is implemented by the *Eventmaker* class. This dialog box provides support to use the event manager to dynamically define new events, or fire any registered event.

Figure 4.2.2 The Chisel "Eventmaker" dialog

4.3 Rule manager

The Chisel rule manager forms the core of the Chisel adaptation manager. After new adaptation policy rules have been parsed by the Chisel policy parser, the rules' data are passed to the rule manager. The Chisel rule manager acts as both a rule repository and a rule

evaluator within the Chisel framework. The main operations of the rule manager are shown in figure 4.3.1.

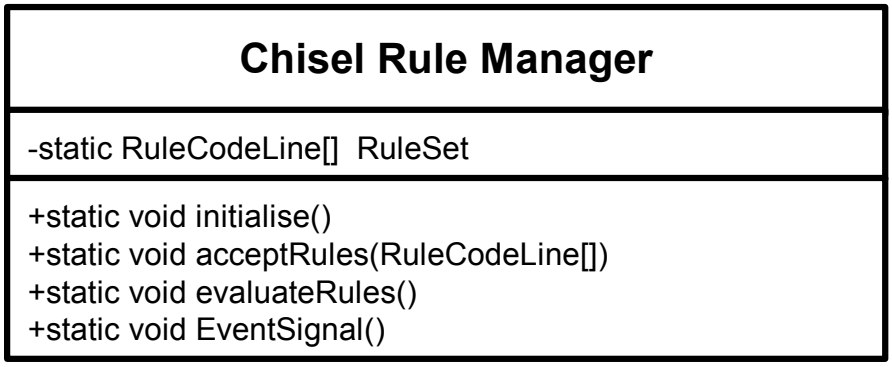

Figure 4.3.1 Key functions of the Chisel Rule Manager

Once an event is fired or cleared using the event manager, the rule manager is notified. The rule manager, operating in a separate background thread, then requests the current set of fired events from the event manager. The rule manager must then select and evaluate the rules that are triggered by one of the events in this set of fired events. The rule manager iterates sequentially through the rules, in the order they were specified in the policy script. For each pass of the rules, each rule is only checked once. An event remains fired for one entire pass over the rules, at which point it is cleared. If one of the triggered rules causes a fired event to be cleared, that event is then removed from the set of fired events, and so will not cause any following rule that would be triggered by that event to be evaluated. If an event is fired, enabled, or disabled by the evaluation of a rule, that event operation is carried out by the event manager. When an event operation occurs, the rule manager pauses its pass over the rule set and requests an update of the set of fired events from the event manager, with any newly fired events added to the rule manager's list of currently fired events. The rule manager then continues its pass over the rule set to find triggered rules. This enforces a design choice that the order of rules specified in the policy script file when it is parsed is important, such that events fired or cleared by policy rules, will only affect rules further down in the list of rules. An event cannot be fired or cleared to affect a rule further up the list of specified rules.

This sequential evaluation of rules based on a single pass over the rule-set, can result in a situation where dependencies between rules and events are difficult to specify, but it does mean that a ping-pong situation resulting from a circular dependency, where one rule triggers another rule that in turn re-triggers the first rule, is not possible.

Once an event triggers a rule for evaluation, the primary role of the rule manager is the interpretation and evaluation of condition statements for that rule. The rule manager contains a comprehensive runtime interpreter to evaluate the rule's conditions, which are specified in a manner similar to the Java programming language. As described in the Chisel policy language specification in the previous chapter, these rule conditions must evaluate to `true` to indicate that the rule operation should be performed. Rule conditions can contain user-specified constant values, arbitrary method calls and field accesses to named events, to arbitrary application classes, or to named individual objects. These constants, field values, and method return values can act as parameters for other method invocations, or as the target objects or classes for further field accesses and method calls. These values can be compared to other values using boolean comparison operators including, "==" (equals), "!=" (not equals), ">"(greater than), ">=" (greater than or equals), "<" (less than), and "<=" (less than or equals). Boolean values and expressions can be combined using boolean combination operators, "&&" (and) and "||" (or).

While evaluating rule conditions the rule manager's condition interpreter makes extensive use of the service manager and event manager to retrieve named event objects, named classes, and named objects to act as the invocation targets for method and field access. Once the object or class is retrieved using its name, the Java reflective API is used to find the appropriate named method using `java.lang.Class.getDeclaredMethod`, or the appropriate field using `java.lang.Class.getDeclaredField`. An extensive collection of operations is provided to support runtime type comparisons and casting, which are required to allow constant values and arbitrary objects and classes of different types to be used as invocation targets or as method parameters. Once the correct method or field is found for the named event, class, or object, the method call or field access is invoked, with the return value then available for casting or comparison operations for use with other condition subsections. The direct specification by the user of type casting code is not supported in the Chisel policy language, so this support for dynamic type checking and casting is vital for the interpretation and evaluation of the conditions code. Yet another collection of operations is provided to support the dynamic comparison of arbitrary objects, which may be of different types.

The Chisel policy language, and as a result the Chisel rule manager does not support the use of the Java `new` keyword to instantiate new objects in a rule's condition block. However, instantiations can be implemented using the `java.lang.Class.NewInstance()` operation provided by the Java reflective API. The Chisel policy language was also

designed to support arithmetic operations on values within the condition block of a rule, however, this is not supported in the current implementation of the Chisel framework.

Once the conditions section for a rule has been fully evaluated, the result must be a boolean value. This value dictates whether the rule operation will be performed. Here the rule operation will be either an event manipulation, translated into a call to the event manager, or a behavioural adaptation, which is translated into a call to the behaviour manager.

So far in this section, only reactive adaptation rules and reactive event manipulation rules have been discussed. As policy scripts are parsed, an event type `ChiselInitialEvent` is defined and registered with the event manager. This event is fired once only, after the rule manager has been passed its set of rules, after which the event type is then disabled. Proactive rules are translated into reactive rules with this event type used as the rule trigger. This means that proactive rules are then triggered only once, immediately after the policy script has been parsed. These rules are then evaluated only once, as standard reactive rules, thereby providing the behaviour of proactive rules

The object casting and comparison operations supplied in the rule manager are also used by the Chisel policy parser to verify newly loaded policy specification scripts. Although many comparison and casting errors may only arise when the conditions are actually evaluated, these parse-time consistency checks are important to identify syntax errors in newly specified rules. The use of brackets in rules is also checked at rule parse time.

This implementation of the Chisel rule manager, with its extensive support for the dynamic interpretation of arbitrary code expressed in a manner very similar to Java, supports the evaluation of unanticipated dynamically specified adaptation control logic, one of the key requirements to support completely unanticipated dynamic adaptation. The ability to dynamically interpret which adaptations should be applied, and where and when they are applied, means that it is no longer necessary to embed adaptation directives in compiled code. This fulfils two more key requirements for completely unanticipated dynamic adaptation, that the location and timing at which an adaptation will be applied can remain unanticipated until after the target application has started execution. The dynamic interpretation of event manipulation directives means that context changes, signalled as events, can be directed in a customisable manner, allowing the unanticipated incorporation of context requirements, thereby supporting the Chisel context model, to allow the use of context information that may not be anticipated to be of importance until during runtime.

4.4 Behaviour manager

The Chisel behaviour manager encapsulates the dynamic adaptation mechanism, i.e. dynamic metatype association. Although the Chisel behaviour manager implements the most important operation of the Chisel framework, i.e., performs dynamic adaptation, the design and operation of the behaviour manager is not complicated. The interface to the Chisel behaviour manager, illustrated in figure 4.4.1 below, is characterised by only two functions, a function to test if a metatype can be associated with a named object or class, and a function to perform that dynamic association. As discussed in Chapter 3, Iguana/J was chosen as the mechanism to support this dynamic metatype association.

The operation `tryChangeBehaviour` simply wraps Iguana/J's `ie.tcd.iguana.Meta.associate` function after retrieving the named target object or class from the service manager, and loading the named class that implements the adaptation behaviour. As described in the previous section the behaviour manager is used by the rule manager to handle adaptation requests. Like the other managers, the behaviour manager is also used by the Chisel policy compiler when the policy script is validated during the policy loading and interpretation process, to verify that the required metatype can be associated with the specified class or object, according to the metatype association rules described in Chapter 3. This however does not ensure that an unanticipated dynamic metatype association will succeed at runtime since the metatype of the target object, its class, or the superclasses of that object's class may have changed to one that is not compatible with the metatype requested by the adaptation operation. If an error occurs, the user is warned and advised to remove the offending rule. Even if this rule is not removed, the rule manager continues operation, and continues attempting to use the rule in case the metatype association later becomes valid.

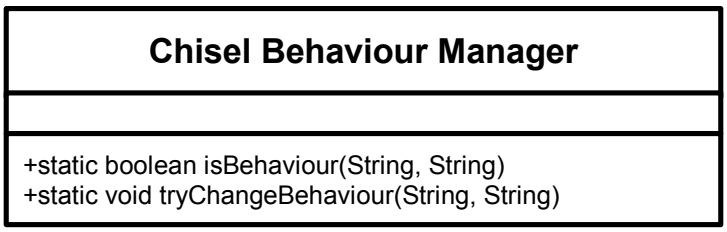

Figure 4.4.1 Key functions of the Chisel Behaviour Manager

Although the use of the metatype model is a requirement of the Chisel framework, all use of the metatype model is encapsulated inside the behaviour manager. Since the behaviour

manager is designed and implemented as a modular element of the Chisel framework, so if replaced, the Chisel framework could be used to support any dynamic adaptation mechanism that supports the dynamic naming, loading, and application of individual adaptations to individual named objects and classes. In the same manner, any other mechanism that implements the metatype adaptation model by supporting dynamic metatype association could be integrated in the behaviour manager without difficulty.

4.5 Service manager

The Chisel service manager, shown in figure 4.5.1, is responsible for finding appropriate named objects and classes for use by the other managers, either to perform adaptations or for use in evaluating rule conditions. In the Chisel framework, all objects and classes are referred to by name, so the Chisel service manager is responsible for the mapping between these names and the classes and objects they represent. For named classes, this is achieved using the Java reflective API, simply using the `java.lang.Class.forName` function. For objects, this is achieved using the Chisel named object store, which maintains a mapping between individual objects and unique names for those objects. The implementation and operation of the Chisel named object store is described separately in the following section. In this regard, the internal class-specific operations of the service manager are largely provided by the Java reflective API, while the object-specific operations are largely provided by the Chisel named object store, so the service manager implementation is relatively uncomplicated.

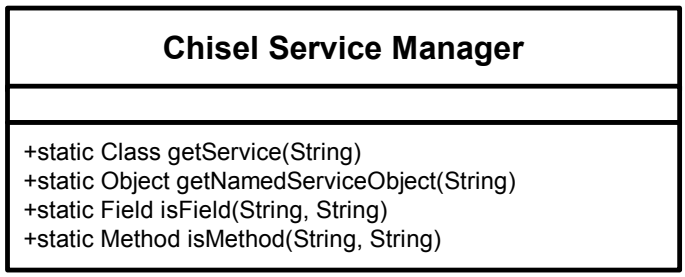

Figure 4.5.1 Key functions of the Chisel Service Manager

Like the event manager, the service manager is also used by the Chisel policy compiler when a policy script is validated while being loaded and interpreted, particularly to verify that object and class names are valid, and that named method and field accesses are valid for their target objects or classes. The service manager is also responsible for using the Java reflective API to retrieve these fields and methods for use by the Chisel rule manager as it

interprets rule conditions. If the field request, but particularly the method request, would return more than one method or field, this search functionality is overridden in the rule manager. It should be noted that the standard Java rules regarding permissions and usage scopes apply here, so only public methods and fields can be accessed using the service manager and rule manager. It is also possible to retrieve protected (but not private) methods and fields of objects and classes in the same package as the rule manager, i.e., those objects and classes used in the Chisel adaptation framework in the `ie.tcd.Chisel` package.

As described in section 3.5 in the previous chapter, any named object, or named class can act as a context source. In this respect, the Chisel service manager also plays an integral role in the Chisel context model, alongside the Chisel context manager. The service manager allows the user to add unanticipated context lookups, in the form of arbitrary field and method accesses, to the conditions section of arbitrary adaptation rules and event manipulation rules. This allows the adaptation process to operate in a context-aware manner, where the specification of what is considered an important context value can remain unanticipated until during runtime, at which point the user can specify what context value to look up in Chisel policy rules. This can all occur as the requirements for adaptation change dynamically, in unanticipated and erratic manners.

4.6 Named object store

A key requirement of the Chisel framework is that individual objects can be dynamically adapted in a completely unanticipated manner, and without access to the source code where that object is instantiated. However, in the Java execution environment, there is no built-in mechanism to specify and refer to individual object instances in a human-readable manner at runtime without associating a name with that object and maintaining a mapping between the name and the object referred to. This behaviour is implemented in the Chisel named object store.

The Chisel named object store supports the programmatic insertion of name-object mappings. This insertion of name to object mappings can be used either in an anticipated manner by adding a call to this function in the application source code, or more importantly, in an unanticipated manner from within a dynamically loaded metatype. This registration of a name to object mapping is accomplished using the `addObjectReference` function provided by the named object store, as seen in figure 4.6.1. Within the Chisel named object store, each object registered in the store is referred to using a weak reference to that object.

If the weak object reference becomes stale, e.g., if the object is deleted, the object reference and its associated name are removed from the named object store.

Once the easily recognisable name is coupled with the object reference, that name can be used at any time in the adaptation policy rules, either as a target for adaptation itself, or as part of a condition block in any adaptation rule. As discussed in the section above, the Chisel service manager makes extensive use of the Chisel named object store to find individual named objects that are used in adaptation rules and event manipulation rules. The `getObject` operation, shown in figure 4.6.1, is used to perform this query.

A prototype implementation of an object naming mechanism for the Chisel named object store provides logging and profiling behaviours that can be associated dynamically with any class, thereby logging all instantiations and invocations, along with the parameter values and types, of all objects of that type. By presenting these logs in a clear and filtered format, the user or administrator can select which object is to be named. Once the object is selected for naming, the user provides a user-friendly name, at which point that name to object mapping is registered in the named store. This profiling behaviour, implemented as a metatype, can be dynamically disassociated from the target class at any time, while the named objects retain their names. This implementation of the Chisel named object store is treated in detail as a case study in the following chapter.

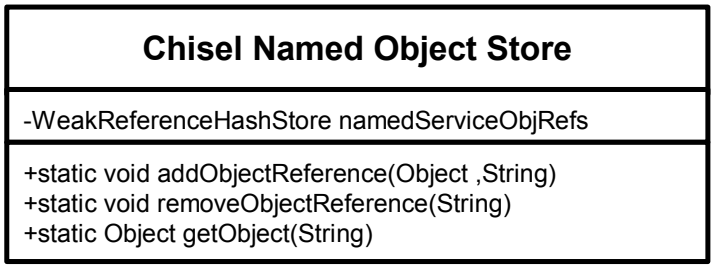

Figure 4.6.1 Key functions of the Chisel Named Object Store

Section 3.2.4 has discussed the requirement that the user must first decide which object or class to adapt or use in rule conditions, before that object or class is used in policy rules. If the user wishes to refer to a named object, that object must first be named by using this Chisel named object store. Whether the programmatic interface is used directly from meta-level code, or the profiling mechanism described above is used, the user must first anticipate at some stage prior to adaptation which objects must be named. The amount of time that it takes the user to identify the object or class that needs to be adapted or queried is dependent on the user, the complexity of the application, and the type of adaptation required. However,

this monitoring and understanding of the operations of the target application does not need to occur before the application has started execution, so this anticipation remains post-runtime anticipation.

4.7 Context manager

As discussed in Chapter 3, the Chisel context model is partly provided by the Chisel context manager, and partly provided by the Chisel service manager, as described above. The roles of the Chisel context manager are twofold: to provide a mechanism to identify and query the values of context variables, and to monitor these variables for change according to context monitor alerts. Named context variables can be dynamically defined by creating an instance of the `ContextVariable` class. Alerts can be created by creating an instance of the `ContextCheckCondition` class also shown in figure 4.7.1. The design and operation of these context variables and context alert monitors is also discussed in section 3.5 of Chapter 3.

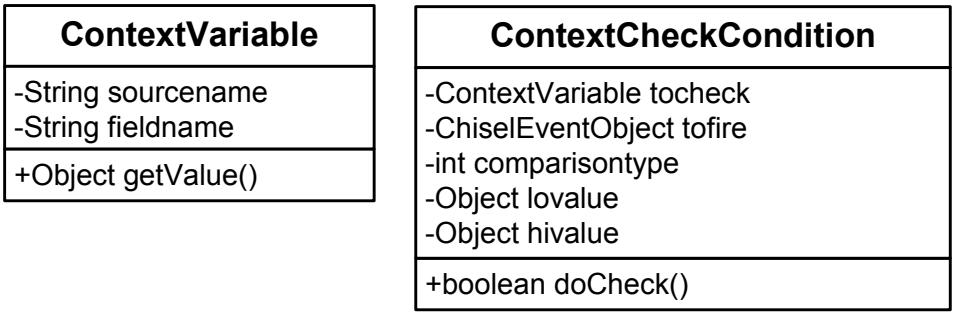

Figure 4.7.1 Data representations of context variables and context alert conditions

Once a `ContextVariable` object is created, it is registered with a name in the context manager using the `addContextVariable` function, as shown in figure 4.7.2. The context manager can then be used to query the value of named context variables using the `getContextVariable` function, which uses the service manager to retrieve this context variable value.

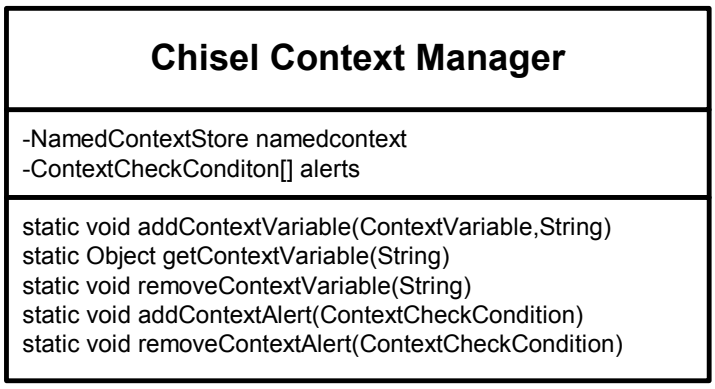

Figure 4.7.2 The principal functions of the Chisel Context Manager

After a ContextCheckCondition object is added to the context manager, the context manager constantly monitors its context variables, and performs the specified test on the context variable's value and fires the event via the event manager if required. The comparison test types supported in ContextCheckConditions are *BETWEEN*, *LESSTHAN*, *MORETHAN*, *NOTBETWEEN*, *EQUALS*, and *NOTEQUALS*. If only one boundary value is required, e.g., for the *EQUALS* comparison, the *hivalue* object is not used. The context manager makes use of the comparison and casting operations implemented in the rule manager to evaluate these conditions. If the context variable comparison evaluates to a value outside of the bounds defined by the *lovalue* and *hivalue* objects, the event manager is used to fire the event specified in the *tofire* field. This monitoring operation could also have been translated into reactive event manipulation rules, as for the *WHEN* keyword described in section 4.2 above.

Section 3.5 in Chapter 3 also states that these named *ContextVariable* objects and *ContextCheckCondition* alert monitor objects cannot be created using policy rules, but instead must be implemented in source code. Context queries from within policy rules are supported using the Chisel service manager.

4.8 Policy parser / policy manager

The Chisel policy parser is responsible for taking the policy specification file and parsing its event definitions, the adaptation rules, and the event manipulation rules, and converting them to a data format that can be used by the rule manager and the event manager. In order to support dynamic policy specifications, policy scripts can be loaded at any time into the

policy parser at runtime, at which point the policy manager is then responsible for propagating the new policies to the other parts of the Chisel adaptation manager.

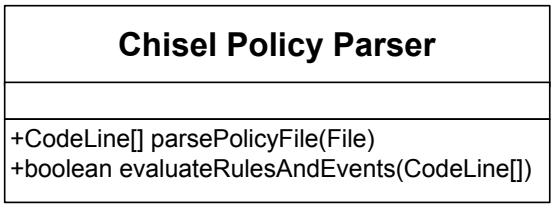

Figure 4.8.1 The principal operations of the Chisel policy parser

Although not created by a parser generator, the Chisel policy parser contains a complex set of internal operations to first tokenize the policy file, identify the type of each token in two passes, and convert each token into a `CodeUnit` object, shown in figure 4.8.1. Each line is made up of a series of `CodeUnits`, grouped into a `CodeLine` object. A parsed policy script file is represented as an array of `CodeLine` objects.

The Chisel parser is responsible for the correct parsing and verification of the rules specified using the Chisel policy language. This parsing and verification is performed in a practical and informal manner, whereby each rule is extensively verified to ensure that all events used are correctly registered with the event service, that all adaptations (metatypes) are compatible with the target objects or classes according to the metatype association rules described in section 3.2.5 in Chapter 3, that all method calls and field accesses used in the conditions section are valid, and that the syntax of the rules is correct. This verification is to aid the user to correctly specify adaptation logic and to minimise the possibility of runtime interpretation errors when rules are evaluated.

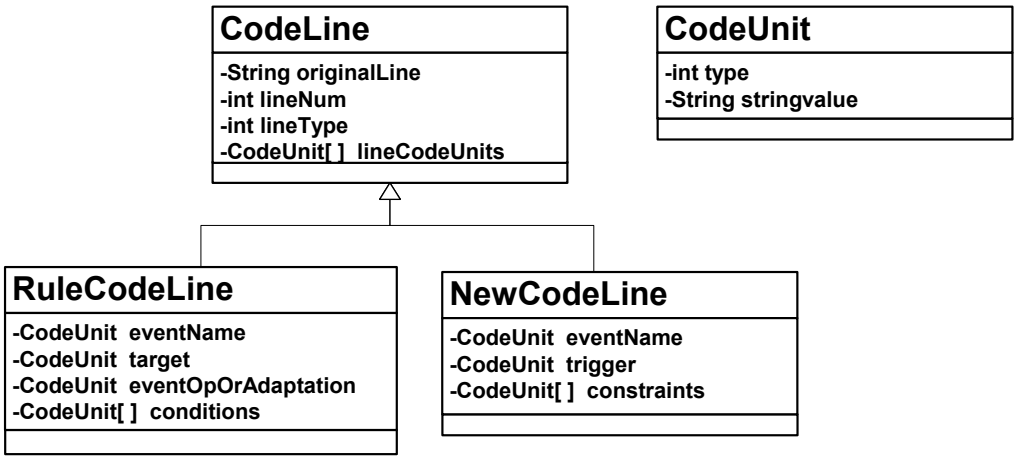

Figure 4.8.2 Data representations of policy rules

When the script is verified, the array of `CodeLine` objects that represent the policy script file is split into two arrays, where new event declarations are converted to objects of type `NewCodeLine`, and all reactive and proactive rules are converted to objects of type `RuleCodeLine`, both shown in figure 4.8.2 above. The `NewCodeLine` objects are then passed to the event manager, as described in section 4.2 above, with the `RuleCodeLine` objects passed to the rule manager, as described in section 4.3 above. When the event manager and the rule manager have been updated the event manager and rule manager resume operation as normal.

By supporting the dynamic specification and dynamic parsing of policy scripts, the Chisel policy language and the Chisel policy parser together support the specification of new, unanticipated, and context-aware adaptation control logic during runtime. This fulfils the requirement for the Chisel adaptation framework to support the unanticipated specification of adaptation control logic, and the unanticipated specification of where and when unanticipated adaptations should be applied.

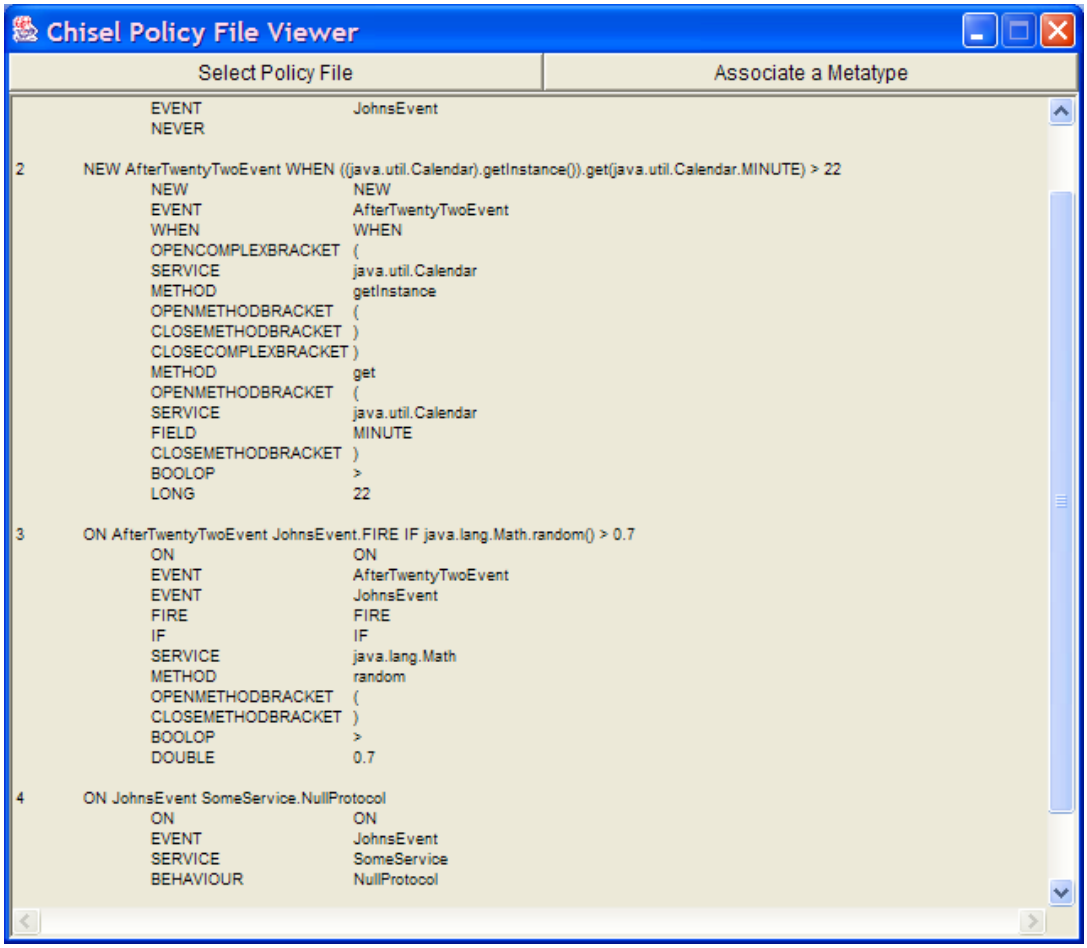

Figure 4.8.3 The Chisel policy file viewer demonstration

As part of the Chisel framework implementation, a demonstration graphical interface was developed to allow the user to view the currently selected adaptation policy script, and to allow the user to dynamically provide a new policy script file for the policy manager to parse. This viewer allows the user to view the policy rules, as they are specified in the policy script, and how they are represented once parsed. This viewer, shown in figure 4.8.3 above, is provided by the `PolicyFileViewer` class.

4.9 Summary of the operation of the Chisel framework

The sections above describe in detail the implementation and operation of the constituent parts of the Chisel dynamic adaptation framework, i.e., the event manager, the rule manager, the behaviour manager, the service manager and named object store, the context manager, and the policy parser. When combined, these constituent parts work together to allow the Chisel dynamic adaptation framework to support context-aware, completely unanticipated dynamic adaptation, in an open and extensible manner.

This is achieved by providing an event model provided by the Chisel event manager and a context model provided by the Chisel service manager and Chisel context manager, which signal dynamic changes in possibly unanticipated context variables using events. The Chisel policy parser supports the dynamic and unanticipated specification of adaptation control logic using the Chisel policy language. This adaptation policy specification is based on the dynamic monitoring and manipulation of context events, and the evaluation of arbitrarily complex condition evaluations and context lookups, with these adaptation policy specifications interpreted and evaluated using the Chisel rule manager. Finally, if these rules specify that an adaptation should be performed, the Chisel behaviour manager performs the dynamic association of possibly unanticipated metatypes to arbitrary application classes and named objects, thereby performing unanticipated adaptation in a context-aware manner.

Figure 4.9.1 below, based on figure 3.3.2 in Chapter 3, demonstrates the operation of the Chisel adaptation manager. Firstly, an event is fired, using the event manager's `FIREEvent` function. The rule manager is signalled that an event has fired using the rule manager's `EventSignal` function. By default, this operation causes the rule manager's thread to request the set of fired events from the event manager via the `getFiredEvents` call. The rule manager then iterates through its rule set to find the first rule triggered by any of the fired events. When it finds a rule that has been triggered, the rule manager begins evaluating the conditions for that rule, if any. During the evaluation of these conditions, it

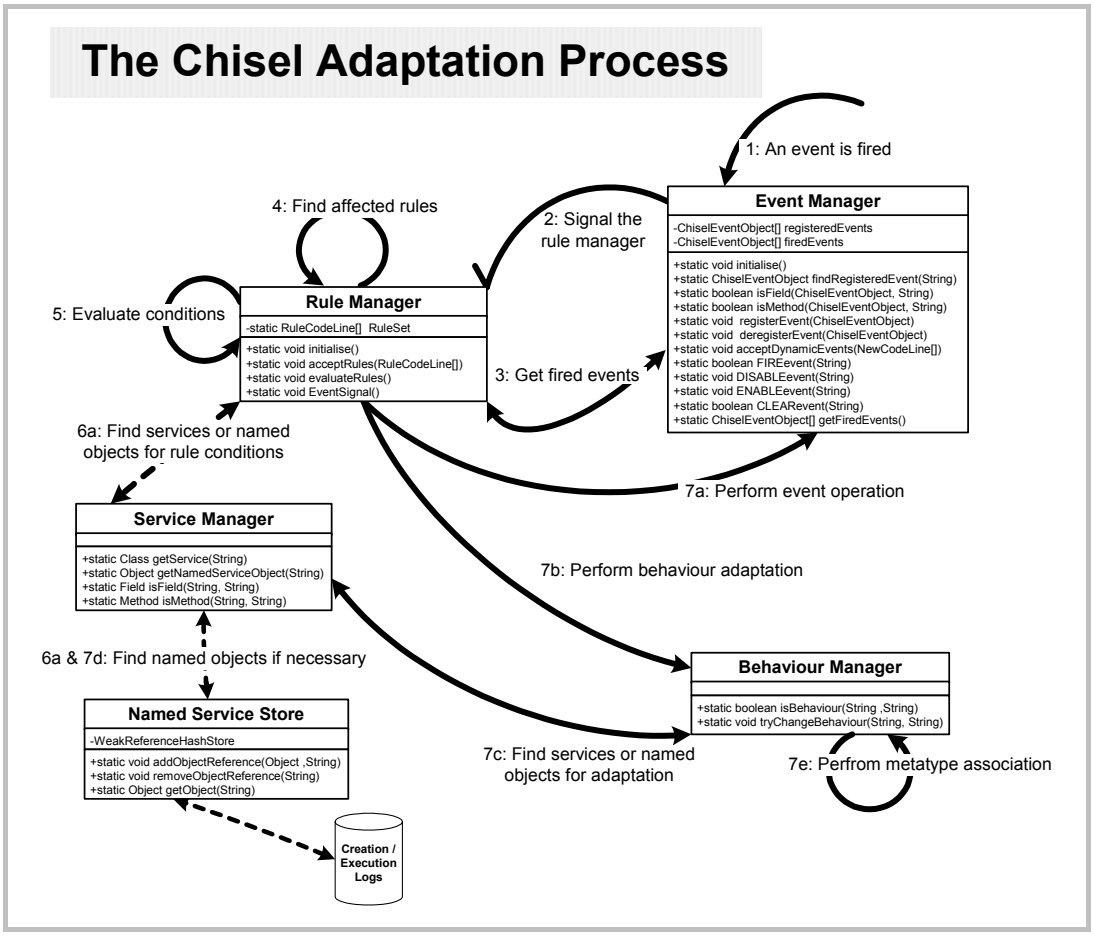

Figure 4.9.1 The detailed operation of the Chisel dynamic adaptation manager

may be necessary to lookup a named class from the service manager, using the *getService* call, or if this indicates that the name refers to a named object, the *getNamedServiceObject* method is used, which in turn invokes the named object store's *getObject* method. Once the rule manager has evaluated all of the rule's conditions, and if successful, the rule manager must either perform an event manipulation operation or initiate an adaptation. If an event manipulation is required, the event manager's *FIREEvent*, *CLEAREvent*, *ENABLEEvent*, or *DISABLEEvent* methods are used. If a behavioural adaptation is required, the behaviour manager's *tryChangeBehaviour* function is called with the name of the service object or class to adapt and the name of the metatype to associate with that object or class. The behaviour manager then requests the named class or object from the service manager using the *getService* or *getNamedServiceObject* method as above. The behaviour manager then locates the class implementing the metatype, checks that it is a valid metatype, and then attempts to associate the metatype with the class or object. The evaluation of this rule is then finished,

so the rule manager continues with the evaluation of the next triggered rule, but first requesting an update of fired events from the event manager if the previous rule's outcome was an event manipulation operation. Once all of the rules are evaluated, the rule manager waits for another event, or until a timeout has occurred.

While the operations described above are being performed, the context manager monitors named context variables specified in registered `ContextCheckCondition` objects, and raises context events if the values of those context variables move outside specified bounds. Meanwhile the policy parser waits for the user to pass in a new policy script specification file. At the same time the event manager should be checking automatic event triggering specifications, but this is not currently implemented. The managed application also executes at the same time, along with any other executing software that the user launched when the adaptation manager was launched. For example, the user may have started the execution of additional context monitoring modules, which can register and manipulate new event types, register named context variables and monitoring alerts, register object names with the named object store, or perform any other action, including access the programmatic interfaces of the adaptation manager.

4.10 The programmatic interface and the policy-based interface

As mentioned throughout this chapter, there are a number of operations that can only be performed either using programmatic interfaces in code, or using the policy-based interface using the Chisel policy language.

New objects cannot be instantiated using the `new` Java keyword in policy scripts. This can be frustrating when specifying parameter objects for method invocations. This restriction, along with the restriction that the conditions section of a rule must result in a boolean value, was introduced to try to prevent the use of the rule manager as a general-purpose Java language interpreter, rather than as a mechanism solely for the evaluation of rule conditions. It is intended that all behavioural changes be implemented as metatypes, rather than as arbitrarily executed code specified in rule conditions.

It is currently not possible to define named context variables or register context-monitoring alerts with the Chisel context manager using the Chisel policy language. Again, this is can only be achieved using the context manager's programmatic interface. To add the feature to

the Chisel policy language would require the definition of a new type of rule, since these operations should not form part of a rule condition block. The specification of rules to dynamically fire events in response to the evaluation of a context monitoring operation is already supported by the use of *WHEN* for newly defined events. In this respect, it is possible to provide the behaviour of context monitoring alerts.

The specification of automatic triggering conditions is currently only possible when using the Chisel policy language to define new event types. This automatic triggering mechanism is not supported for event types defined in code. The triggering of events defined from source code should be handled by the code used to define them. Of these automatic triggers, only the *WHEN* keyword is currently supported. The implementation of support for the *AT* and *EVERY* keywords is planned.

4.11 Attaching the adaptation manager

There are a number of mechanisms mentioned in Chapter 3 to attach the Chisel adaptation manager to the operation of arbitrary application classes. The Chisel adaptation manager is created as an instance of the *ChiselC1* class. This object simply initialises the rule manager, the event manager, and the context manager. This object also provides the interface for the policy manager. The code to create a *ChiselC1* object must be attached to the application code for execution in some manner. This *ChiselC1* object can then have new policy scripts occasionally passed in for parsing, or instead an instance of the sample *PolicyFileViewer* class described above can be created which presents a user interface to the user to support the dynamic location, loading, and parsing of policy scripts.

```
import ie.tcd.Chisel.*;
…
      ChiselC1 chiselc = new ChiselC1();
      PolicyFileViewer pviewer = new PolicyFileViewer ( chiselc, …);
…
```

Figure 4.11.1 Example of code needed to start the Chisel dynamic adaptation framework

Whichever way the Chisel adaptation manager is attached to the application being monitored, the Java execution environment must be started in a manner to enable the Iguana/J runtime component. This is accomplished using a command-line option, described in the Iguana/J documentation [124].

4.11.1 In the application source code

If the application source code is available, the adaptation manager can be started from within the application, with the application recompiled, and executed. This simply requires adding the code described in figure 4.11.1 into the application source code. However, this solution is far from ideal since the application source code may not be available, and challenges the objective of the Chisel framework to perform unanticipated dynamic adaptation of arbitrary applications without damaging the original application.

4.11.2 As a custom application launcher

A custom launcher class `ChiselLauncher` was developed to start the Chisel adaptation manager, and then start the application class. To use this launcher, the user must execute the `ChiselLauncher` class instead of the application class, but pass the name of the application class and all of its command-line arguments to the `ChiselLauncher` class. As shown in the code excerpt in figure 4.11.2, the launcher implementation simply locates the `main` entry point into the application class and invokes it, passing in the original command-line arguments. This results in the normal execution of the application in all respects, except that the Chisel adaptation manager is also enabled.

```
import java.lang.reflect.Method;
public class ChiselLauncher {
  public static void main(String args[]){
      …
      String args2[] = new String[args.length-1];
      System.arraycopy(args, 1, args2, 0, args2.length);
      Class application = Class.forName(args[0]);
      Method mainmethod = application.getDeclaredMethod("main", new Class[]{args.getClass()});
      /********* Enable Chisel **********/
      ChiselC1 chiselc = new ChiselC1();
      PolicyFileViewer pviewer = new PolicyFileViewer ( chiselc, …);
      /**********************************/
      mainmethod.invoke(null, new Object[]{args2});
      …
  }
}
```

Figure 4.11.2 The implementation of the ChiselLauncher class

4.11.3 As a statically assigned metatype

Another mechanism to attach the Chisel adaptation framework to an application in an unobtrusive manner is to embed the initialisation of the Chisel adaptation manager in a metatype. Figure 4.11.3 and figure 4.11.4 show the metatype `EnableChisel`, and its meta object class `MetaObjectExecuteEnableChisel`. This metatype performs no

behavioural change except to disassociate itself from the class with which it is associated. However, when the meta object class *MetaObjectExecuteEnableChisel* is first loaded, it initialises the Chisel adaptation manager in static code, i.e., as the class is loaded.

```
class MetaObjectExecuteEnableChisel extends MExecute {
      static{
          /********* Enable Chisel **********/
          ChiselC1 chiselc = new ChiselC1();
          PolicyFileViewer pviewer = new PolicyFileViewer (chiselc, … );
          /*********************************/
          /********* Any other initialisation code **********/
          …
          /************************************************/
      }

      private String newprotocol = "ie.tcd.iguana.NullProtocol";     //default value
      public MetaObjectExecuteEnableChisel(String newMetatype){
          super();
          newprotocol = newMetatype;
      }
      public Object execute(Object o, Object[] args, Method m) … {
          …
          Class theclass = m.getDeclaringClass();                    //find the application class
          ie.tcd.iguana.Meta.associate ( theclass, newprotocol );    //disassociate this metatype
          …
          return proceed(o, args, m);                                //continue
      }
}
```

Figure 4.11.3 Meta object class that initialises the Chisel adaptation manager

```
protocol EnableChisel (){
      reify Execution : MetaObjectExecuteEnableChisel();
}
protocol EnableChiselEx ( String newProtocol ){
      reify Execution : MetaObjectExecuteEnableChisel ( newProtocol );
}
```

Figure 4.11.4 The EnableChisel metatypes that initialise the Chisel adaptation manager

This metatype can be associated with the main application class using static metatype association, specified in an initialisation file for the Iguana/J runtime component. It is reasonable to require that the name of the main class of the application is known, since this name is required by the Java execution component. In the example shown in figure 4.11.5, the main application class *SomeService*, has one of the *EnableChisel* metatypes statically associated with it. In this case, as the *SomeService* class is loaded to start the application, the *EnableChisel* metatype class will be loaded, so the meta object class *MetaObjectExecuteEnableChisel* will be loaded, and so the Chisel adaptation manager will be initialised. When the main method of the *SomeService* class is called, the class will have the Iguana/J *NullProtocol* metatype associated with it, i.e., the metatype that would have been associated with it by default if this mechanism were not

used. If the user wishes to have different metatype associated with the class, this metatype name can be passed as a parameter to the static metatype association specification.

```
SomeService==>EnableChiselEx ("ie.tcd.iguana.NullProtocol");
```

Figure 4.11.5 Static association of the EnableChisel metatype with an application class

4.12 Summary

This chapter described in detail the implementation of the Chisel dynamic adaptation framework. This chapter also discussed how the adaptation manager operates and how the Chisel framework can be used to dynamically adapt general-purpose compiled software. Also included are descriptions of a number of mechanisms to attach the Chisel adaptation manager to an arbitrary compiled application, and how this can be achieved in an unobtrusive manner, and in a manner that remains unanticipated until the start of execution. This requirement to anticipate the need to enable and initialise the Chisel adaptation does not break the requirement to support completely unanticipated dynamic adaptations, since only the possible need to adapt is anticipated, but no part of any individual adaptation needs to be anticipated until during the execution of the application.

The following chapter describes in detail two case studies to evaluate the Chisel dynamic adaptation framework and its ability to support completely unanticipated dynamic adaptation of arbitrary applications, in a context-aware manner, without requiring access to the source code of the application.

Chapter 5
USING THE CHISEL FRAMEWORK: CASE STUDIES AND EVALUATION

This chapter describes two case studies used to evaluate the capabilities and limitations of the Chisel dynamic adaptation framework. The objective of these case studies is to substantiate the contributions of the Chisel framework and evaluate its usability to perform completely unanticipated dynamic adaptation on arbitrary general-purpose software in a context-aware manner. While demonstrating these case studies the strengths and weaknesses of the metatype model for dynamic adaptation will also be discussed where appropriate. This chapter also discusses the performance penalty of using the Chisel dynamic adaptation framework in terms of the time taken to perform some of the key functions of the Chisel adaptation manager.

5.1 Evaluation criteria

This general-purpose support for context-aware completely unanticipated dynamic software adaptation will be evaluated using a number of criteria. These criteria are presented in table 5.1.1, and discussed below.

Since completely unanticipated dynamic adaptation is the primary objective of the Chisel dynamic adaptation framework, the degree of support for unanticipated adaptation will be discussed for both case studies. This discussion of completely unanticipated dynamic adaptation will be broken into 4 subcategories: firstly, the ability to dynamically perform adaptations where the actions of the adaptation are unanticipated; secondly, the ability to perform adaptations at arbitrary locations which have not been explicitly prepared; thirdly,

the use of adaptation policies that are specified and interpreted at unanticipated times, with those policies containing unanticipated control logic; and, finally, the ability to perform adaptations at unanticipated times.

While making use of unanticipated control logic, this control logic should be able to exploit contextual information about the state, resources, and requirements of the user, application, and execution environment, some of which could not be anticipated to be of importance by the application designers and developers. The ability to specify which changing context values are of importance, and how this context information can be exploited to tailor the adaptation process will also be evaluated.

Also discussed is the degree to which the Chisel framework can be used to open up a closed system and expose the operation of that system to allow it to be adapted and tailored, without requiring access to the source code of that application.

The case studies in this chapter both perform dynamic adaptation using the metatype model for adaptation, particularly by using dynamic metatype association to change the behaviour of compiled software, either by changing the functional behaviours of the software, or by inserting new non-functional behaviours into the software. The degrees to which these case studies demonstrate the usefulness of the metatype model, and expose the limitations of the model will also be analysed.

- Support for completely unanticipated dynamic adaptation:
  - Dynamic adaptation where the adaptation performed was unanticipated
  - Dynamic adaptation at unanticipated and unprepared locations
  - Use of unanticipated control logic and adaptation policies to drive adaptation
  - The unanticipated timing of the application of adaptations
- Support for awareness of changing and possibly unanticipated context
- The ability to adapt general-purpose software in an open manner, without requiring access to source code
- The use of the metatype model for dynamic adaptation, demonstrating its usefulness and limitations.

Table 5.1.1 Evaluation criteria for the Chisel dynamic adaptation framework

The case studies presented demonstrate the usefulness of the Chisel dynamic adaptation framework to support particular adaptations, but the case studies should also demonstrate how other general-purpose adaptations could be performed in a similar manner. These case studies are representative of a range of possible adaptations, and the extent to which this is demonstrated by each case study will be addressed.

The particular case studies selected are: an implementation of the Chisel named object store where support for profiling and naming individual objects was implemented as an unanticipated dynamic adaptation; and the application of a dynamic adaptation to support the operation of an off-the-shelf network application to operate in a mobile computing environment. These case studies were chosen to demonstrate and evaluate the Chisel framework from two distinct but complementary viewpoints.

The first case study, an implementation of the Chisel named object store was particularly selected to demonstrate that any application can be probed, profiled, and inspected without any preparation of the target application. The fact that the Chisel named object store is completely independent of the target application demonstrates the general-purpose nature of the Chisel framework. This case study also demonstrates the unanticipated nature of the naming of individual objects since the location of an object to be named, when it is named, or how the named object will be later used, demonstrate some of the features of completely unanticipated dynamic adaptation. The ability to name random objects and use them either as targets for adaptation, or as context sources, further demonstrates the flexibility of the Chisel framework.

The second case study, the unanticipated adaptation of a network application to operate in a mobile computing environment, demonstrates what may be an unanticipated need to adapt a third-party application to operate in an environment for which it was not designed. No assistance is required from the application developers, or indeed access to the source code of the application, to allow it to be adapted to operate in an environment requiring frequent and unanticipated changes. The mobile computing environment was chosen specifically because of its erratically changing context. A middleware for mobile computing, ALICE, was chosen because of its availability and the author's familiarity with its operation. A telnet application was chosen because of the telnet protocol's particularly inflexible approach to disconnection and reconnection and so its incompatibility for use in a mobile computing environment.

5.2 Case Study: The Chisel named object store

Although a key component of the design of the Chisel framework, the Chisel named object store is itself implemented and enabled using the Chisel framework. The case study demonstrates both the operation of the named object store, and how the association of non-

functional behaviours with arbitrary application classes is supported by the Chisel dynamic adaptation framework, and therefore demonstrates its usefulness and flexibility.

5.2.1 Motivation

A requirement of the Chisel framework is the ability to perform unanticipated dynamic adaptation on individual application objects. The Java programming language and execution environment provides no mechanism to identify or refer to individual objects from outside the source code of the application. In order to perform completely unanticipated dynamic adaptation of individual objects, without access to the application source code or the ability to recompile the application to embed hooks to identify individual objects, individual objects have to be identified, or have their identification associated with them, as or after the objects are created. At runtime, the identification of individual objects is handled internally by the execution environment, which cannot be inspected or altered at runtime without substantially altering the runtime environment itself. In Java, it is possible to obtain a unique identifier for each object, using the `System.identityHashCode` operation, but there is no mechanism to obtain a reference back to the object using this identifier.

In the Chisel framework, the adaptation directives regarding how objects are to be adapted are specified using the Chisel policy language. In order to adapt an individual object, it must be possible to uniquely refer to that particular object in the policy rules. In order to make the Chisel framework and the Chisel policy language easy to use, these object identifiers should be human-readable and should clearly identify individual objects within the application. For this reason, it was decided that the user should be able to provide the name used for an individual object since the user can associate a useful and memorable name to the object for later use in adaptation policy rules. Since the Chisel framework was designed to support completely unanticipated dynamic adaptation, it is impossible to anticipate which objects will be adapted; therefore, it is impossible to anticipate which objects will require names and which will not. It would be infeasible to request a name for every object created during the execution of an application, as this level of required constant interaction with the application as it operates would make the application frustrating, if not impossible, for the user to use.

5.2.2 Design

Since objects do not exist, and so cannot be used until after they are created, it is not necessary that an object be named until some time after it is created. Once an object is

created, the main mechanisms that can be used to differentiate an object from an object of the same type are: its constructor parameters; the operations performed on it, along with the parameters passed and values returned; when the object was created; and the state of the object. If all operations, along with their parameters and return values, that are performed on each object of that type, the constructor arguments for each object, and the time each object is created, are all profiled and clearly presented to the user in a filtered manner, the user can then select which objects of that type should be named. Once the user has selected the objects of interest, recognisable names can then be specified for those objects. Once a mapping exists between the name and the object, the user can use that object name in policy adaptation scripts, either as adaptation targets themselves, or for use in the conditions section of rules to direct the adaptation of another object or to manipulate events.

This profiling mechanism is embedded in a metatype, which can then be associated with any class. When this metatype is associated with a class, the metatype is also associated with all current and future instances of the class and its subclasses. All method calls to all of those objects are then profiled and logged, including information on the method called, object invoked, the time, the number of parameters along with their types and values, and the type and value of the result. When a new instance of the class is created, the metatype logs the constructor parameters and the creation time.

The first step to locating an object of interest is the association of this profiling metatype, called the `ChiselBaseLogging` metatype, with the object's class. It was seen as a reasonable requirement, that if a user wished to name an object, they must first know the type of the object. This association can be performed using the Chisel policy interface, by creating a new proactive adaptation rule as described in figure 5.2.1 below.

```
INITIALLY ApplicationClassName.ChiselBaseLogging
```

Figure 5.2.1 Association of the ChiselBaseLogging metatype with an application class to profile operation of all of its instances

This profiling information is then logged to a database for inspection by the user. A web-based user interface was developed to allow the user to view and filter the contents of this database as operations continue on the profiled objects. Once the user has selected an object to be named, a name is requested and then applied to that object. Immediately afterwards that name can be used in adaptation policy rules. At any time, the profiling metatype can be disassociated from the application class to discontinue logging the operations of its instances. The named objects will maintain their names even after the disassociation of this metatype.

Figure 5.2.2 shows this web-based interface, where the operations of an object of type `SomeService` are profiled. This view is for one object, which the user has selected from a set of profiled objects. In this example, `SomeService` is a simple Java class with one method, `sayHello`, which passes back a string containing the current time. In a sample application, a number of `SomeService` objects are instantiated with the `sayHello` method repeatedly called at random. The Chisel web service also allows the user to specify a name for any object once the user has selected the relevant object. Once this name is set, the object name is reflected back to the named object store. This store is then used by the Chisel service manager to locate object references when names are used in adaptation policy rules.

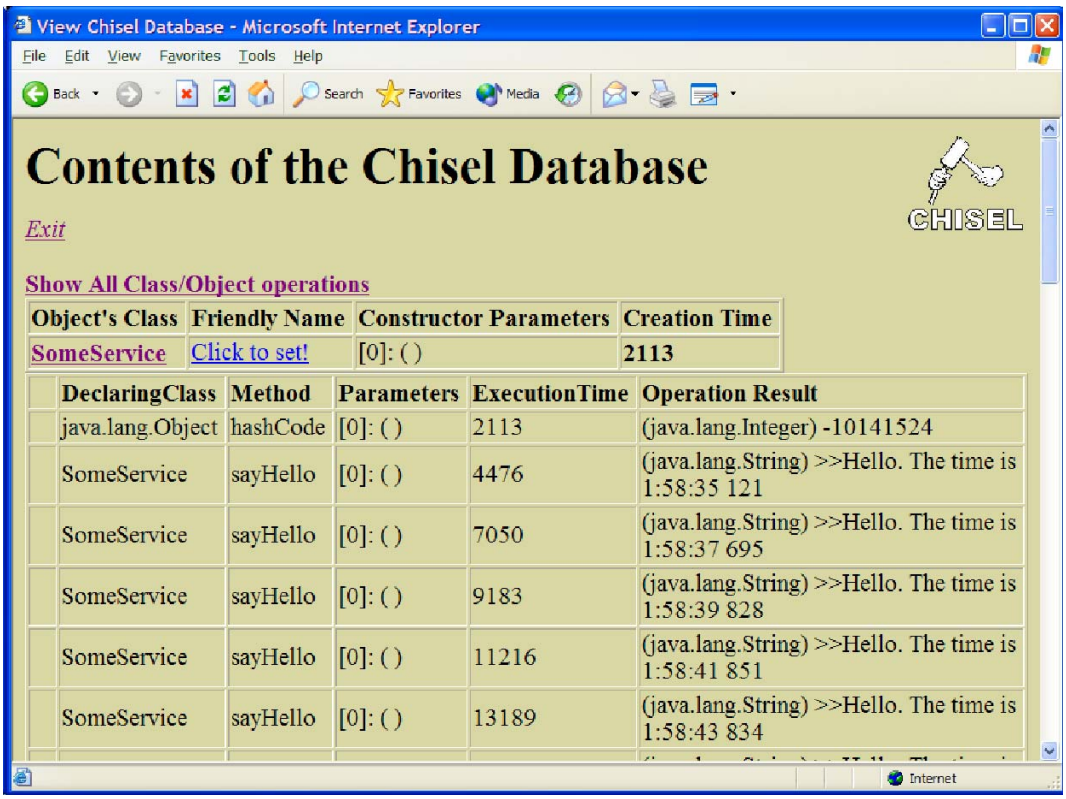

Figure 5.2.2 Filtered view of the Chisel webservice database containing profiling data about arbitrary application objects

From figure 5.2.2, the user can see that the `sayHello` method of this instance of the `SomeService` object is repeatedly called with no parameters. Once the user has decided that this object should be named, the user simply clicks the "*Click to set*" hyperlink to be presented with a simple dialog to enter a name that will be assigned to this object using the Chisel named object store's programmatic interface to register the name with the chosen

object. As shown in figure 5.2.3 below, copied from section 4.6 in Chapter 4, the method `addObjectRefence` is provided for this registration operation.

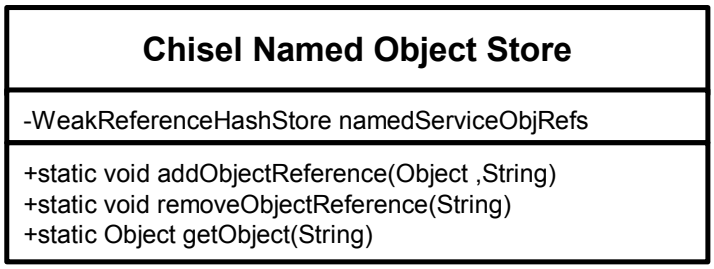

Figure 5.2.3 Key functions of the Chisel Named Object Store

Once a name-to-object mapping is passed to the Chisel named object store, an internal mapping between the name and a weak reference to the named object is created. Inside the named object store, this information is stored in a `ReferenceHashStore`, which maintains the internal data mapping between weak object references and names. This use of weak references to each object allows each object to be removed and garbage-collected when it is no longer in use by the application. Once garbage-collected, the weak object reference becomes stale, so the name no longer maps to a valid object. When this happens the name is disassociated from the reference so no further use of the object is possible. If a rule contains the name of an object that has been deleted, a warning is presented to the user by the Chisel rule manager to request that the user removes the offending rule, after which the application and rule manager continue normal operation. If the user is unavailable, or chooses not to remove the rule, the rule manager will continue to warn the user, but keep operating as normal without evaluating that rule. If at any time the user reuses the name for a different object, the rule will then operate normally again as before.

As discussed in the previous chapters, although this is a supporting mechanism for the user, the responsibility of identifying the object, by interpreting its behaviours and roles, remains with the user. The amount of time it takes the user to decipher the behaviours and roles of an object is dependent on the complexity of the object and the abilities of the user. This time taken to decipher the behaviour and roles of a particular object requires that the need to name this object must be anticipated before the object can be named. However, there is no requirement that this anticipation should occur before the profiled application has started execution. In this respect, runtime unanticipated naming of individual objects is achieved.

This mechanism supports the identification of individual application objects, in any general-purpose application, without any requirement to access the source code of the application,

stop the application, or interfere with its normal operation in any way. The only requirement is that the class of the object to be named is known, either by name, or identified by other meta-level code.

5.2.3 Implementation

The first step to implementing the Chisel named object store is the specification of the *ChiselBaseLogging* metatype using Iguana/J. This metatype is composed of two meta object classes, one to reify object creation *MetaObjectCreateBaseLogging*, and one to reify method execution *MetaObjectExecuteBaseLogging*.

From the source code extract in figure 5.2.4 for the *MetaObjectCreateBaseLogging* meta object class, the Iguana/J operation *proceed* creates the object and places the return value in the *result* object. An unnamed reference to this object is then added to the *ReferenceHashStore*. If the user decides later to specify a name for this object, the *ReferenceHashStore* will be updated to include this name, and so provide a mapping between the name and the object. The class *DBConnection* provides the operation to have the creation of this object logged to the database used by the Chisel webservice, along with the object's type, constructor parameters, and time of creation. The *result* object is then passed back to the application for normal use.

```
class MetaObjectCreateBaseLogging extends ie.tcd.iguana.MCreate {
  public Object create(Constructor cons, Object[] args)  …  {
    …
    Object result = proceed(cons, args);                    // create the object
    …
    ReferenceHashStore.addObjectReference(result, … ); // add an unnamed reference to the store
    DBConnection.logCreation(result, cons, args, … );       // log creation to the database
    return result;
  }
};
```

Figure 5.2.4 The MetaObjectCreateBaseLogging meta object class

Since the *ChiselBaseLogging* metatype will be associated with the application class, it will also be associated with every instance of that class. The meta object class, *MetaObjectExecuteBaseLogging*, shown in figure 5.2.5 below, reifies the execution of all method calls to each of these instance objects. When a method is invoked, the *ReferenceHashStore* is first checked to determine if the target object is registered in the store, and if not it is added. If the object was created before the *ChiselBaseLogging* metatype was associated with the object's class and the current method call is the first since that association, and so the object's reference will not registered

in the *ReferenceHashStore*. The Iguana/J operation *proceed* then performs the method invocation on the base object, passing the method parameters untouched, and then stores the return value, if any, in the *result* object. The class *DBConnection* also provides the operation to have the execution of this object's methods logged to the database, including the method called, the number of arguments along with their types and values, the type and value of the method's result if any, and the start time of the execution of the method. The *result* object is then passed back to the application for normal use.

```
class MetaObjectExecuteBaseLogging extends ie.tcd.iguana.MExecute {
  public Object execute(Object obj, Object[] args, Method meth ) ... {
      ...
      if( ! ReferenceHashStore.containsObjectReference(obj) )   /* If the object's not in the store */
          ReferenceHashStore.addObjectReference(obj, ... );     /* add it to the reference store */
      ...
      Object result = proceed(obj, args, meth);                 /* execute the method */
      ...
      DBConnection.logExecution(obj, meth, args, result ... );  /* log creation to the database */
      return result;
  }
};
```

Figure 5.2.5 The MetaObjectExecuteBaseLogging meta object class

These two meta object classes are then combined into a single metatype and compiled by the Iguana/J MOP compiler to produce the metatype class *ChiselBaseLogging* as seen in figure 5.2.6 below.

```
protocol ChiselBaseLogging {
  reify Creation:MetaObjectCreateBaseLogging();
  reify Execution:MetaObjectExecuteBaseLogging();
}
```

Figure 5.2.6 Definition of the ChiselBaseLogging metatype

As was seen in figure 5.2.1, this *ChiselBaseLogging* metatype can then be associated with any application class during the operation of the application by using a proactive adaptation rule, specified using the Chisel policy language which is passed to the Chisel policy parser. In figure 5.2.7 below, the application class *SomeService* has the *ChiselBaseLogging* metatype associated with it.

```
INITIALLY SomeService . ChiselBaseLogging
```

Figure 5.2.7 Association of the ChiselBaseLogging metatype with application class SomeService

As introduced in the previous section, *SomeService* is a simple Java class with one method, *sayHello*, which passes back a string containing the current time. The operations of each object of this class are then profiled, with the profiling information for each object displayed to the user, as seen from figure 5.2.2 in the previous section. A similar view of the

collected profiling data is available for each object of each class that has the `ChiselBaseLogging` metatype associated with it. The user can then select an object, associate a name with that object, and then use that name in any policy rule.

Once the name is used in a policy rule, the Chisel service manager will check if the name is registered with the named object store using the `getObject` method shown in figure 5.2.3. The named object store will check its store of name-object mappings stored by the `ReferenceHashStore` to locate a weak object reference. Once found, the object referenced by the weak object reference will be retrieved if the object is still available. If the particular object is not available, the named object store will alert the service manager, which will alert the rule manager, which in turn warns the user the rule cannot be evaluated.

5.2.4 Alternatives

A number of alternative methods to assign unique names could have been used instead of the method described above. One method would involve using the Java RMI Registry [150] to store the names of objects described as Java RMI components. However, only objects that implement the `java.rmi.Remote` interface can be added to and retrieved from the RMI registry. This mechanism is therefore unsuitable for use in the Chisel named object store since it is required that arbitrary application objects can be added to the store.

Another method would be the use of the Java Naming and Directory Interface (JNDI) [149]. JNDI provides a standardised Java language interface for the use of naming and directory service implementations. This interface supports the insertion and querying of name to object bindings. Currently JNDI provides standardised interfaces for the LDAP [163], DNS [103], and Java RMI Registry directory services. In order to provide a naming mechanism required for the Chisel named object store, a new backend naming service would still have to be implemented. The implementation of the Chisel named object store described above could be ported to act as this service provider and so export the standardised JNDI interface. This is a topic for further work in the implementation of the Chisel named object store.

It is also possible to store an object's name in the meta objects associated with the object. However, this mechanism has a number of shortfalls. When a metatype is disassociated from an object, the metatype's meta objects, and all state in those meta objects are lost. This means that once the name storage metatype is disassociated from the object, its name is lost. This mechanism also provides no central storage of names that can be searched in order to provide a mapping of names to objects. The first version of Iguana [58] supported the

association of a name with individual objects in this manner; however, this mechanism was removed in later versions of Iguana.

5.2.5 Evaluation and discussion

This implementation of the Chisel named object store is provided to demonstrate the power of the metatype model to provide a useful non-functional service, such as extensive profiling support and object naming, in a dynamic and unanticipated manner for any arbitrary application class or object. This case study was provided mainly to demonstrate completely unanticipated dynamic adaptation.

Neither the individual application objects that are named, nor their classes which have the `ChiselBaseLogging` metatype associated with them, are prepared in any way to have this adaptation applied to them since the location at which this adaptation is applied is completely unanticipated.

What the adaptation does can also be unanticipated until during runtime. At any time, the `ChiselBaseLogging` metatype can be changed, for example to provide a mechanism to log different aspects of the intercepted operations, or redirect the database connection to a different database. If recompiled before used in an adaptation policy rule, the `ChiselBaseLogging` metatype can provide this extended behaviour. Otherwise, a new metatype can extend the operation of the `ChiselBaseLogging` metatype by extending the operation of either the `MetaObjectExecuteBaseLogging` or `MetaObjectCreateBaseLogging` meta object classes and create a new metatype class that implements an extended behaviour.

For example, if the user also wishes to log all instantiations of a class to a local data file as well as to the Chisel webservice database, a new metatype can be created. There are two options available to create this metatype. Firstly, the user can create the new metatype that just performs the logging to file behaviour, and then create a third metatype that inherits from both this new metatype and the `ChiselBaseLogging` metatype, thereby combining the behaviour of both metatypes. An alternative is to create a new meta object class that inherits from the `MetaObjectCreateBaseLogging` meta object class to incorporate the database logging behaviour, and extends it to perform local file logging. A new metatype can then be created that uses this new meta object class. Figure 5.2.8 demonstrates this second example.

```
class MetaObjectCreateBaseLoggingEx extends MetaObjectCreateBaseLogging {
  public Object create(Constructor cons, Object[] args)  …  {
      …
      /* let the superclass handle object creation, database logging, reference registration etc */
      Object result = super.create (cons, args);
      FileLog.logCreation(result, cons, args, … );          /* log creation to a local file */
      …
      return result;
  }
};
```

Figure 5.2.8 The MetaObjectCreateBaseLoggingEx meta object class

Here a new meta object class is created, extending the operation of the class *MetaObjectCreateBaseLogging*. This class simply defers the operation of the *create* method to its superclass, then logs the creation of the object *result* using the *FileLog* class. A new metatype *ChiselBaseLoggingEx* is then defined as shown in figure 5.2.9 below, which is then compiled in a separate process using the Iguana/J compiler.

```
protocol ChiselBaseLoggingEx {
  reify Creation:MetaObjectCreateBaseLoggingEx();
  reify Execution:MetaObjectExecuteBaseLogging();
}
```

Figure 5.2.9 Definition of the ChiselBaseLoggingEx metatype

Immediately after the Iguana/J compiler has finished compiling the metatype class, this metatype can be used in any policy rule, again perhaps as a proactive rule as seen in figure 5.2.7 above. The time when this profiling metatype is associated with arbitrary application classes is completely dependent on the timing at which the user passes the triggering policy rule to the Chisel policy parser, and so is unanticipated until during runtime. As described previously in this thesis, these adaptation policy rules can contain arbitrary and unanticipated condition blocks to control the association of this profiling metatype, or any other metatype. This mechanism is useful to support customised dynamic association and disassociation of the profiling metatype, and so dynamically enable and disable the profiling operation of the Chisel named object store, thereby allowing the specification of unanticipated control logic to manage the application of this adaptation. In summary, this mechanism supports the completely unanticipated profiling and naming of any arbitrary object in any arbitrary application.

This case study also demonstrates the mechanism and the usefulness of creating and parsing new adaptation policy rules to dynamically adapt the operation of both the target application and the Chisel framework as the application runs. Only when the user decides that an object should be located and named is it necessary to create the policy rule to enable or disable this

new non-functional behaviour. This policy rule can then be passed into the Chisel policy parser whenever the user is ready to enable or disable the behaviour.

This case study also demonstrates the ability of the user to expose the implementation of any application by profiling in detail the operation of individual objects within the application. The user need not have any information about the operation of the target application, and no access to the application source code. This method does not alter or interrupt the execution of the application in any way, nor does it require any preparation of the application beforehand. This is primarily achieved by the use of dynamic metatype association, which wraps the behaviour of arbitrary application objects and classes, allowing the behaviours of those objects and classes to be thoroughly inspected or adapted.

However, the use of a database to store the profiling data, and so requiring individual database accesses for each profiled object creation and execution request, imposes a considerable computational overhead and slows down the operation of the application being adapted[2], as can be seen from table 5.2.10 below. This prototype implementation of the Chisel named object store and the Chisel framework in general, was implemented to provide flexibility and ease of use, to demonstrate the usefulness of the metatype model for dynamic adaptation, and to support the completely unanticipated dynamic adaptation of general-purpose software, rather than provide a high performance solution.

Time taken to profile the creation of an object	1.98 ms (std. dev. 0.41)
Time taken to profile the execution of an method invocation	4.19 ms (std. dev. 0.75)
Time taken to name an individual object	2.33 ms (std. dev. 0.91)
Time taken to retrieve an object given its name	3.21 ms (std. dev. 0.62)

Table 5.2.10: Time taken to access the Chisel named object store

5.2.6 Wider applicability

The Chisel named object store case study, as described, forms a fundamental part of the Chisel framework and provides support for the adaptation of arbitrary named application objects. However, the use of the metatype model to implement an extensive non-functional behaviour, and then have this behaviour associated with arbitrary application classes in a completely unanticipated manner, demonstrates the power of the Chisel dynamic adaptation framework to dynamically adapt any class or object in a flexible and managed manner.

[2] All timings were taken on a lightly loaded Dell Inspiron 8600 laptop computer, 2 GHz Pentium M processor, 1GB RAM, Windows XP SP1. Using JDK version 1.3.1_06.

This case study particularly demonstrates the support provided by the Chisel framework to allow the user to arbitrarily decide to adapt a random application class in a manner where the adaptation can be applied, and then removed, as the user wishes, all without interrupting the application, and without requiring access to the source code of the application.

5.2.7 Summary

This case study has both provided an in depth description of the implementation and operation of the Chisel named object store, and discussed the usefulness of the Chisel dynamic adaptation framework to dynamically inject non-functional behaviours into an arbitrary application. This case study also demonstrates the ability to harness the Chisel framework to expose the internal operation of black-box application, not only showing how the class and its instances operate and interact with other classes and object, but also expose the locations where the classes and objects can be adapted or expanded as the user requires.

5.3 Case Study: Adaptation for mobile computing

This case study introduces how an application can be adapted in an unanticipated manner to enable its use in a mobile computing environment.

5.3.1 Motivation

This section describes the difficulties encountered by both applications and middleware in a mobile computing environment, and so elaborates on the need for adaptation.

What is mobile computing?

Physically mobile computing devices that work in standalone mode can be termed “nomadic computing” [102]. “Mobile computing” however can be considered an extension of distributed computing, whereby portable devices have access to (possibly remote) information services regardless of their movement or physical location [72], thereby making use of distributed services while also supporting the notion of nomadic computing. The main difficulty with mobile computing is the inherent scarcity and variability of resources available for use by mobile computers as they move. The primary resource requirement of a mobile device, when it is working as part of a distributed system, is its network connection, usually some form of wireless connection, which when used by a device that is physically moving, suffers from unanticipated and possibly prolonged disconnections [54]. The reason why this issue is such a major problem for mobile computing is that the applications

currently being developed are being built as distributed systems applications, assuming a stable interconnection, and do not sufficiently account for these disconnections and reconnections [119].

Middleware for mobile computing

"Middleware can be viewed as a reusable, expandable set of services and functions that are commonly needed by many applications to function well in a networked environment" [1]. Traditional middleware for distributed systems, residing above a network-enabled operating system but below the application, provides a homogeneous interface and programming model for the application regardless of the uncertain local execution environment. It abstracts away the complexities of the underlying environment, communication subsystems and distribution mechanisms, thereby providing a single view of the underlying environment, and sheltering applications from the underlying environment, communication subsystems and distribution mechanisms. Such features are seen in traditional middleware systems such as COM+ [100] Java RMI [150] and CORBA [106].

On the other hand, a middleware system for mobile computing must be flexible in order to provide such a homogeneous and stable interface and programming model to make use of possibly erratic execution contexts. A key requirement for middleware for mobile computing is the ability to adapt to drastic changes in available resources, especially network connection availability [61]. To achieve such a stable execution environment it is often necessary that an adaptable middleware for mobile computing be 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.

Difficulties with applications and middleware for mobile computing

As environment conditions change, to values unknown unforeseen by the application designer, the middleware that provides abstractions for these environmental resources must dynamically adapt to support the applications that run on top of that middleware. As stated, one of the primary services provided by the middleware is the ability to supply network communications services as these resources change. The characteristics of the available connections can range from an inexpensive, very high-bandwidth, low-latency connection such as a high-speed wired LAN connection, to a very expensive, low-bandwidth, high-latency connection such as a GSM connection, where each communication protocol may make use of different communication models and addressing modes.

Mobile computing applications should also be able to handle periods of disconnection, supported by the middleware underneath. The difficulties that are associated with such a range of connection characteristics are further compounded by the fact that these characteristics can change in an unanticipated manner. For example, these disconnections occur when the device moves out of range for wireless connections, or an interface device is suddenly disconnected, as seen when a user suddenly disconnects the device from a synchronisation cradle or removes a networking device currently in use.

Despite the presence of an adaptable middleware that can function in such a dynamic environment, applications running on this middleware must be able the cope with the differing levels of service provided. If an application is built without taking account of the changeable nature of the execution environment as presented by the middleware, i.e., a mobile-transparent application [75], it must adopt the most conservative choices for resource availability and connectivity. If an application is written in a mobile-aware manner [75], the application can support changes to available resources, but this adaptation support must be embedded in the application source code.

A further issue with such a varied collection of communication technologies that can be leveraged for mobile computing is that the user may not wish to fully use the available resources in an eager or greedy manner to maintain data connectivity. For example, even if a GPRS connection is available, it is generally much more expensive than available wired or wireless LAN connections. A further example is the case where although currently disconnected, with connections available, the user may be aware that cheaper or more convenient connection resources will soon be available, i.e., something that cannot be anticipated in a generalised manner by the adaptable middleware platform. For these reasons, it is imperative that the added potential of the user's own resources, preferences, and intelligence are exploited. This model reinforces the wide-ranging interpretation of "context-awareness" taken by the Chisel framework to allow all queryable characteristics of the user, the application, and the execution environment to be used as sources of contextual information.

5.3.2 ALICE

ALICE (Architecture for Location-Independent Computing Environments) [7, 29, 61-63, 156, 157], developed in Trinity College Dublin, is an architectural framework that supports network connectivity in a mobile computing environment by providing a range of client/server protocols (see figure 5.3.1). ALICE allows these protocols to provide their own

support for location management, disconnected operation, and connection management. In ALICE, Mobile Hosts are mobile devices, which may interact with fixed computers called Fixed Hosts. These connections are tunnelled through Mobility Gateways, which are also fixed computers. The mobile host can become disconnected from a mobility gateway and later become reconnected to a different mobility gateway without interfering with the virtual connection to the fixed host. A fixed host can also be made up of a combination of a mobile host and mobility gateway. Both the fixed host and the mobile host can act as client or server for any protocol.

ALICE is made up of a series of layers. The Mobility Layer (ML) handles communications between devices by overriding socket functions 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. Protocol-specific Swizzling Layers reside above the ML and support mobility of servers by translating server references and redirecting client connections to more up-to-date server locations according to these references. When a disconnection occurs, the ALICE ML will synchronously queue unsent data between the mobile host and the mobility gateway until a connection is re-established. ALICE has been implemented in C [29, 61-63] and in Java [7, 156, 157]. Versions exist for CORBA [29, 61-63] and Java RMI [7, 61]. A version supporting SOAP is also planned.

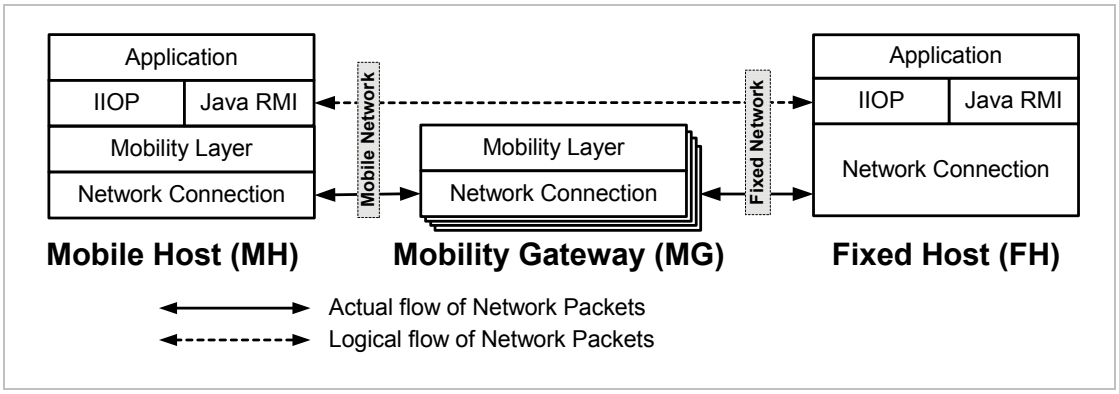

Figure 5.3.1 Overview of the ALICE middleware framework

At present, ALICE does not provide support to force disconnections in order to enable the selection of a different communication connection than the one that is employed at a given time. 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 can also be specified to handle caching of server object replicas in a protocol-specific or application-specific manner. When addressing the requirement that the user and the application may wish to drive or constrain the adaptation of the middleware, the main issue that is not addressed by ALICE is the relative difficulty to control which connection to use and the incorporation of semantic information to make a more informed choice about how the ML should reconnect the mobile host and the mobility gateway.

5.3.3 Design

This need for context-aware unanticipated dynamic adaptation of both mobile computing middleware and mobile-aware applications was seen as an ideal case study for the Chisel dynamic adaptation framework. The operation of middleware and applications on mobile computers should not be interrupted just because the resources or requirements of the user, application, or operating environment have changed in an unanticipated manner. Ideally, it should be possible to adapt an application for mobile computing in a mobile-aware manner, but without requiring direct access to the application source code in order to embed this behaviour.

To demonstrate the use of the Chisel dynamic adaptation framework, an off-the-shelf network application, "The Java™ Telnet Application/Applet" [77], was adapted to operate in a mobile computing environment by dynamically adapting it to use the ALICE mobility layer. Once adapted, the application can operate freely in a mobile computing environment since the ALICE middleware framework presents a socket interface that does not fail when a network disconnection occurs.

5.3.4 Implementation

For this case study a full Java implementation of the ALICE mobility layer was completed, based on the work presented in [156, 157]. It provides a class `MASocket` that contains the ALICE connection behaviour, which implements a socket interface similar to the standard Java socket class, `java.net.Socket`. When the `MASocket` class is used instead of the standard Java socket, all messages from a mobile host to a fixed host are redirected via a mobility gateway. When the connection between the mobile host and the mobility gateway breaks, all network data are cached at the mobile host and the mobility gateway for later reconnection. This disconnection and reconnection happens without the application being made aware of the disconnection.

This dynamic adaptation of the telnet application was achieved using the Chisel dynamic adaptation framework, without stopping the application and without changing the source code of the application in any way. Although the source code for this telnet application is available, it was not used to adapt the application. The only assumption made about the programming of the application was that a standard Java socket, or a subclass of `java.net.Socket`, is used to connect the client and the telnet server, a reasonable assumption for any network-enabled Java application.

A metatype, `DoAliceConnection` (figure 5.3.2), was developed to intercept the creation of the socket connection to the telnet server and replace the socket in use with an instance of the ALICE `MASocket`. This redirection behaviour was embedded in the meta object class `MetaObjectCreateALICEConn` as shown in figure 5.3.3.

```
protocol DoAliceConnection  {
  reify Creation: MetaObjectCreateALICEConn ();
}
```

Figure 5.3.2 Definition of the DoAliceConnection metatype (MOP) class

```
class MetaObjectCreateALICEConn extends ie.tcd.iguana.MCreate {
  public Object create(Constructor cons, Object[] args)  …  {
      …
      if ( … /* this is not a localhost connection, or a connection used by ALICE */ … ) {
          // Change the constructor to be called, from a java.net.Socket to a MASocket constructor
          cons = ….  // find the MASocket constructor
      }
      Object result = proceed(cons, args);     /* create the socket */
      …
      return result;                           /* result is either a normal socket or an MASocket */
  }
};
```

Figure 5.3.3 The MetaObjectCreateALICEConn meta object class

The redirection behaviour is accomplished by first checking that this intercepted creation operation is for a socket connection between the client application and a remote computer, and that it is not a connection used by ALICE itself. If this object creation operation is a valid instantiation that should be redirected to a `MASocket` object creation, the constructor for `MASocket` is found using the Java reflective API and passed to the Iguana/J `proceed` operation. This operation passes the intercepted object creation operation to any other meta objects that are also intercepting the operation, then finally creates the object.

This adaptation can be applied to the telnet application in a number of ways. The first is to use a proactive adaptation rule to associate the `DoAliceConnection` metatype with the `java.net.Socket` class as seen in the previous case study, in figure 5.2.7. An alternative is to perform this adaptation in a context-aware manner, i.e., only perform the

metatype association if the application is being used in a mobile computing environment, where the network connection is known to be intermittent. In the adaptation policy rules seen in figure 5.3.4 below, the *DoAliceConnection* metatype is only associated with the *java.net.Socket* class if the *UsingDodgyNet* event fires. When the connection moves to a stable network connection the *UsingGoodNet* event is fired, thereby re-enabling the use of standard Java sockets.

```
ON UsingDodgyNet java.net.Socket.DoAliceConnection
ON UsingGoodNet java.net.Socket.NullProtocol
```

Figure 5.3.4 Enabling and disabling the DoAliceConnection metatype in a context-aware manner

The event *UsingDodgyNet* can be fired automatically by the Chisel event manager using an automatic rule definition and trigger rule, or by the Chisel context manager when a wireless connection is detected for example, by the user using another event manipulation policy rule, etc. Figure 5.3.5 shows how this event is fired from meta-level code. Similarly, the *UsingGoodNet* event can be fired when the network connection is deemed stable, by another policy rule, some network monitoring code, or by the context manager. Figure 5.3.6 shows how the user causes the event to be fired explicitly when it is known that the network is stable.

With Java, the only ways to detect that a socket data-link disconnection has occurred is if the *write* function of the socket's *SocketOutputStream*, or the *read* function of the socket's *SocketInputStream* causes an *IOException* exception, or if *read* returns the error integer value *-1*. Each of these error conditions is checked in the meta object class *CheckNetworkSendAndReceive* shown in figure 5.3.5 below.

This meta object class, *CheckNetworkSendAndReceive*, is then used in a metatype that is associated with the classes *java.net.SocketInputStream* and *java.net.SocketOutputStream*. An alternative is to associate the metatype directly with a particular socket's *InputStream* object and *OutputStream* object as that socket object is being created or used.

```
class CheckNetworkSendAndReceive extends ie.tcd.iguana.MExecute {

  public Object execute(Object obj, Object[] args, Method meth )
                  throws  InvocationTargetException {

          Object result;
          try{
              result = proceed(o, args, m);
          }
          catch(InvocationTargetException e){
              if(e.getTargetException().getClass()==IOException.class)
                  EventManager.FIREevent("UsingDodgyNet");
              throw e;
          }
          if(m.getName().compareTo("read") == 0)
              if(((Integer)result).intValue() == -1)
                  EventManager.FIREevent("UsingDodgyNet");

          return result;
  }
};
```

Figure 5.3.5 Detecting a network error by intercepting network operations

When the user later decides that the available data connections are stable, the user explicitly fires the `UsingGoodNet` event as shown in the proactive event manipulation rule in figure 5.3.6 below. The user could also include a number of more complex context lookups in the conditions section of the rule.

```
INITIALLY UsingGoodNet.FIRE IF TRUE
```

Figure 5.3.6 Explicitly firing the UsingGoodNet event

To test the operation of this case study implementation, the telnet application was copied onto a standard laptop[3] with the Chisel framework installed. A jar file containing a version of the ALICE mobility layer was also available. The application was then started, but no telnet connection started. The Chisel policy rules described above were then written and parsed by the Chisel adaptation manager. A telnet connection was then made, with the connection automatically redirected over an ALICE `MSocket` connection. At a later stage, once the connection was operating for some time the laptop was carried to a location where a wireless networking signal was unavailable. At this stage the wireless networking card was completely shut down and removed. While disconnected all data was cached at the mobile host and the mobility gateway. At a later stage, in a location where a different wireless networking connection was available, a different wireless networking card with a

[3] Dell Inspiron 8600, 2 GHz Pentium M processor, 1GB RAM, Windows XP SP1. Two wireless networking cards were also available, a Cisco Aironet 350 card and a Compaq WL110 card.

different address was inserted. At this stage the telnet connection relayed and received all cached data and continued its normal operation.

5.3.5 Alternatives

A number of alternatives to this version of the ALICE mobility layer could have been used. A Java version of the ALICE mobility layer is presented in [7] which uses a Java Native Interface (JNI) [145] approach to replace the native implementation of the Java socket library with a re-implementation based on the original ALICE mobility layer written in the C programming language. Using the Chisel adaptation framework, these alternative socket classes can be used in place of the standard Java sockets, but since the mobility layer is implemented in native code, the Chisel framework cannot be used to adapt or inspect the internal operation of this mobility layer implementation; for example, its connection strategies or state, or its caching behaviour, etc. A number of other transport layer level approaches, such as Mowgli [88, 89] or MSOCKS [96] could have been used despite having only limited support for disconnections. However, since these systems are also not implemented in Java and so would require a JNI interface, they are unsuitable for this case study. Another possible approach is the use of MobileSockets [107, 108], a full Java re-implementation of a socket library that supports disconnection and host mobility similar to the ALICE mobility layer used above. Again, to use this mechanism, the creation of a socket object can be redirected to use MobileSockets instead of standard Java sockets. Since MobileSockets are fully implemented using Java, the MobileSocket implementation could also be dynamically adapted using the Chisel framework.

5.3.6 Further adaptations

Using the Chisel framework further adaptations are made possible, to both the application and the ALICE middleware framework.

When a disconnection occurs between the mobile host and the mobility gateway, the class `MGatewayConnection`, which is responsible for maintaining this connection, calls its method `reestablishConnection`, which in turn calls the `connectAttempt` method with the host name and port number of the mobility gateway as parameters. If the user wishes to connect to a particular mobility gateway, the parameters of this call can be changed. If the `reestablishConnection` method is intercepted, an event can be fired to indicate to the entire ALICE framework and the application that a disconnection has occurred, thereby allowing mobile-aware adaptations of the application. If the user wishes to

maintain a disconnected state, this *reestablishConnection* method can be set to fail, thereby forcing the ALICE mobility layer to continue caching the connection data until the user re-enables the operation of the method. If the user wishes to force a disconnection, the *MGatewayConnection* method *hobbleConnection* can be called at any time to close the connection between the Mobile Host and the Mobility Gateway.

For example the user wishes to switch off the caching of data while the socket is disconnected, the user can create a new metatype that intercepts the sending of data on the *MASocket's OutputStream* object, and if the *MGatewayConnection* shows a disconnected state, the data to be sent can be discarded, as shown in figure 5.3.7 below.

```
class NoDisconnectedCaching extends ie.tcd.iguana.MExecute {

  public Object execute(Object obj, Object[] args, Method meth ) ... {

      if ((obj.getClass()==MOutputStream.class) && (meth.getName().compareTo("write")==0))
          if (MGatewayConnection.broken == true || MGatewayConnection.sock == null
              ||  MGatewayConnection.input == null || MGatewayConnection.output == null )
          {
                  return null;
          }
      }
      return  proceed(obj, args, meth);
  }
};
```

Figure 5.3.7 Disabling caching while disconnected

In this example, *MOutputStream* is the type of the *OutputStream* used by the application's *MASocket* to be used when sending data using the mobility layer. *MGatewayConnection* maintains a number of fields that can be used to determine the state of the underlying connection between the mobile host and the mobility gateway. These include: *broken* to indicate that the connection is currently broken, *sock* is the actual *Socket* object used for the connection, and *input* and *output* are the actual *SocketInputStream* and *SocketOuputStream* objects used to send and receive data over the connection. If any of these indicate an error, the socket connection is considered broken. If this connection is broken, the mobility layer is in disconnected mode, so any attempt to send the data will result in that data being cached. If that data should be discarded, the write method can be disabled. Otherwise, the operation should proceed as normal.

These same mechanisms could also be used to support the context-aware adaptation of the application. If the user wishes to adapt the application in a mobile-aware manner, the user must first know when the connections being used are connected or disconnected. The rule

definitions in figure 5.3.8 below, define the events *CurrentlyConnected* and *CurrentlyDisconnected* one of which will be fired repeatedly depending on the current state of the mobility gateway connection.

```
NEW CurrentlyConnected      WHEN      ( MGatewayConnection.broken != true)  &&
      ( MGatewayConnection.sock != null ) &&  ( MGatewayConnection.input != null ) &&
       ( MGatewayConnection.output != null )

NEW CurrentlyDisconnected   WHEN      ( MGatewayConnection.broken == true ) ||
      ( MGatewayConnection.sock == null ) || ( MGatewayConnection.input == null ) ||
       ( MGatewayConnection.output == null )
```

Figure 5.3.8 Automatic definition of events for context-aware adaptation

This mechanism of dynamically redirecting Java socket connections to ALICE *MASocket* socket connections could also be used to dynamically adapt the Java RMI middleware model similar to the approach discussed in [7, 61], but in an unanticipated manner. This possible approach could enable the adaptations described in [7], by intercepting the instantiation of both the *java.net.Socket* and *sun.rmi.server.UnicastRef* classes. An alternative approach could intercept the operations of the *java.rmi.server.RMISocketFactory* interface when it is requested to create the actual sockets used to perform remote object invocations, as described in [156, 157].

5.3.7 Evaluation and discussion of the metatype model

This dynamic adaptation enabled the telnet application to operate as normal while the network connection used was repeatedly disconnected and reconnected, using a variety of network adapters and services. However, due to the nature of the operation of sockets, a connected socket cannot be adapted to use the ALICE mobility layer. To do so would require breaking the connection between the client and the server and replacing this connection with an ALICE socket. However, this cannot be accomplished without the client and the server applications detecting the disconnection, closing their communication sessions, and producing an error. Therefore, this particular adaptation can only be applied before the connection is made. This requires the user to anticipate the need for adaptation before the first connection is made, or else suffer at least one disconnection before applying the adaptation.

In general, it is difficult to adapt an application where object creation is used as the interception point to perform an adaptation. Only objects that are created after the adaptation is applied will be adapted. This is only appropriate when objects are created throughout the operation of the application, as seen here where new socket objects are created for each new

connection created. Adaptations applied at method invocation time, for either classes or individual objects come into effect immediately for all instantiated objects once the adaptations are applied, and are available at the next method call. For this reason adaptation by interception of invocation is a more general-purpose approach and requires less anticipation than adaptation at object creation time.

5.3.8 Evaluation and wider applicability

This case study is designed to demonstrate the usefulness and usability of the Chisel dynamic adaptation framework rather than to quantitatively evaluate the performance of any particular adaptation. For this reason this case study is not evaluated by measuring the performance of the ALICE mobility layer. For more information on the performance characteristics typical of this type of adaptation see related work in [7, 61, 88, 89, 96, 107, 108, 156, 157].

This case study is designed to demonstrate the use of the Chisel dynamic adaptation framework to adapt both applications and middleware for use in a mobile computing environment. Although a mobile computing scenario was chosen to demonstrate the operation of the Chisel dynamic adaptation framework, this case study equally applies to any environment or operation mode where unanticipated dynamic adaptation is a requirement for satisfactory operation. A mobile computing environment was seen as a perfect example since the state, resources, and requirements of the application, the environment, and the user can all change to extreme values in an unanticipated manner.

This case study was also designed to demonstrate the effectiveness of the Chisel dynamic adaptation framework to adapt third-party applications, without requiring access to their source code. This case study demonstrates how the operation of a complex compiled application can be changed dynamically according to the needs of the user and the environment. It also demonstrates how applications can be adapted as they run, without any requirement to change, interrupt, or restart the application.

5.4 Performance

Although not of primary concern for the evaluation of the Chisel framework, support for general-purpose completely unanticipated dynamic adaptation does come with a performance penalty. However, since the Chisel adaptation manager is designed to operate in an asynchronous event-based manner in the background, this performance penalty is

difficult to quantify. To confound this difficulty, the penalty is related to the complexity of the policy rules interpreted by the adaptation manager, with almost every rule performing differently. This section will focus specifically on the Chisel framework in terms of startup time, time to parse a rule, time to interpret a rule once it has been triggered, time to perform an event operation, and time to initiate an adaptation via dynamic metatype selection[4]. This section will not discuss the performance penalty of the adaptation itself since the Chisel framework is built in manner to be somewhat independent of the adaptation mechanism used, and the performance of the Iguana/J framework has been evaluated in depth in [125].

Firstly, the time taken to start and initialise the Chisel adaptation manager is given in table 5.4.1 below. The first row demonstrates just the time taken to initialise the adaptation manager, with the second row showing separately the time taken to start the Chisel policy viewer (seen in figure 4.8.3) and the Chisel "Eventmaker" dialog (seen in figure 4.2.2). This initialisation was performed with an empty rule set and without starting an application for adaptation. The time taken to start and initialise the Chisel adaptation is a real delay to startup of any target application since the adaptation manager must be fully initialised before the target application is started or initialised.

Time taken to start and initialise Chisel	26.24 ms (std. dev. 2.37)
Time taken to start the graphical management interfaces	306.64 ms (std. dev. 15.35)

Table 5.4.1: Time taken to initialise the Chisel adaptation manager

Secondly the time taken to parse a number of policy directives is given in table 5.4.7 below. However, these times should not be considered to be actual delays to the operation of the application being adapted since the parsing of any policy directives can be performed in parallel to operation of the target application at a configurable priority. In the timing examples below, the rules were parsed while the target application was paused to give a more reliable time measurement. To differentiate between different types of policy directives a number of trivial rule sets were parsed. Figure 5.4.2 contains a policy directive to define and register a new event type with the Chisel event manager. The time taken to parse this directive is given on the first row of table 5.4.7 below.

```
NEW Event1
```

Figure 5.4.2: Policy directive to create and register a new event type

[4] All timings were taken on a lightly loaded Dell Inspiron 8600 laptop computer, 2 GHz Pentium M processor, 1GB RAM, Windows XP SP1. Using Iguana/J version 0.36, JDK version 1.3.1_06.

Figure 5.4.3 contains a reactive policy which would cause the *SomeService* class to have the *ChiselBaseLogging* metatype unconditionally associated with it if an event of type *Event1* is fired. The time taken to parse this rule is given on the second row of table 5.4.7 below.

```
ON Event1 SomeService.ChiselBaseLogging
```

Figure 5.4.3: An unconditional reactive adaptation policy rule

Figure 5.4.4 contains a reactive policy which would cause the *SomeService* class to have the *ChiselBaseLogging* metatype associated with it if an event of type *Event1* is fired and the condition containing one static field lookup evaluates successfully. The time taken to parse this rule is given on the third row of table 5.4.7 below.

```
ON Event1 SomeService.ChiselBaseLogging IF ( SomeService.intvalue == 5 )
```

Figure 5.4.4: A reactive adaptation policy rule with a single field comparison as a condition

Figure 5.4.5 contains a similar reactive policy which would cause the *SomeService* class to have the *ChiselBaseLogging* metatype associated with it if an event of type *Event1* is fired and the condition containing one trivial static method invocation evaluates successfully. The time taken to parse this rule is given on the fourth row of table 5.4.7.

```
ON Event1 SomeService.ChiselBaseLogging IF ( SomeService.getInt( ) == 5 )
```

Figure 5.4.5: A reactive adaptation policy rule with a single method invocation in its condition

Figure 5.4.6 contains another similar reactive policy which would cause the *SomeService* class to have the *ChiselBaseLogging* metatype associated with it if an event of type *Event1* is fired and the condition containing a boolean combination of one static field access and one trivial static method invocation evaluates successfully. The time taken to parse this rule is given on the fifth row of table 5.4.7.

```
ON Event1 SomeService.ChiselBaseLogging IF
        ( SomeService.intvalue == 5 ) && ( SomeService.getInt( ) == 5 )
```

Figure 5.4.6: A reactive adaptation policy rule with a combination of comparisons as a condition

Time taken to parse a policy directive to create and register a new event type	3.74 ms (std. dev. 0.50)
Time taken to parse an unconditional reactive adaptation policy rule	7.59 ms (std. dev. 0.47)
Time taken to parse a reactive adaptation policy rule with a single field comparison as a condition	50.13 ms (std. dev. 1.67)
Time taken to parse a reactive adaptation policy rule with a single method invocation in its condition	50.96 ms (std. dev. 2.08)
Time taken to parse a reactive adaptation policy rule with a combination of comparisons as a condition	53.74 ms (std. dev. 2.20)

Table 5.4.7: Time taken to parse Chisel policy directives

Finally, in order to measure the times taken to actually evaluate the rules in figures 5.4.3 to 5.4.6 above, the event `Event1` was fired to trigger the rules. In each case the conditions evaluate successfully, so the timing data given in table 5.4.8 additionally reflects the time taken to perform the metatype association. Unlike the time taken to initialise the Chisel adaptation manager, or the time taken to parse policy directives, rule evaluations are always performed parallel to the operation of the target application. However, again to accurately determine the time taken to perform such evaluations the application was forcefully paused while the particular measurements given in table 5.4.8 were performed. The time taken required to perform an adapted operation is dependent on the contents of the adaptation and is not measured here, however, comparative details on the runtime performance of different types of adaptations using the Iguana/J runtime are given in [125].

Time taken to evaluate a triggered unconditional reactive adaptation policy rule	29.36 ms (std. dev. 0.83)
Time taken to evaluate a triggered reactive adaptation policy rule with a single field comparison as a condition	33.72 ms (std. dev. 2.05)
Time taken to evaluate a triggered reactive adaptation policy rule with a single method invocation in its condition	37.27 ms (std. dev. 2.30)
Time taken to evaluate a triggered reactive adaptation policy rule with a combination of comparisons as a condition	40.90 ms (std. dev. 1.66)

Table 5.4.8: Time taken to evaluate triggered reactive adaptation policy rules

Although the evaluated version of the Chisel adaptation manager was not specifically optimised for speed during implementation, the figures above show that the use of Chisel dynamic adaptation framework itself is not prohibitive. As with any rule-based or event-based system, the performance of the Chisel framework implementation will vary with the number and complexity of the rules used and the frequency at which events are fired. In addition, the number and nature of any adaptations applied will affect the performance of the targeted application. The primary evaluation of the Chisel dynamic adaptation framework is with respect to its flexibility and usefulness rather than its runtime performance penalty. Due to the subjective nature of such an evaluation, and the number of necessary additional considerations, it should be determined on a case by case basis if the

Chisel framework should be used to dynamically adapt a piece of software for intermediate or long term use, or if that software should be rewritten.

5.5 General discussion

The main objective of the Chisel dynamic adaptation framework is to support completely unanticipated dynamic adaptation of arbitrary compiled software in a general-purpose manner. This is demonstrated by both case studies, where arbitrary application objects can be adapted in a completely unanticipated manner, and an arbitrary network application can be adapted to operate successfully in a mobile computing environment. In both of these cases, where the adaptation was applied, what the applied adaptation did, when it was applied, and how its application was controlled, were all unanticipated until after the application had started execution.

The leverage of contextual information to adapt an application was also demonstrated by both case studies, but particularly in the mobile computing case study. Here in particular, the user's contextual knowledge of how the environmental conditions are likely to change, and information from the execution environment as these conditions change, can be used with beneficial effects to drive the adaptation process. In the named object store case study, the knowledge and intelligence of the user is also leveraged to allow individual objects to be identified for later adaptation, or for use in the adaptation control logic.

The ability to inspect and adapt the operations of third-party applications, without requiring access to their source code, is also demonstrated in both case studies. In both cases, arbitrary applications are adapted, with absolutely no requirement for either a-priori knowledge of these adaptations, or preparation of the application to support these adaptations. In both cases the internal operation of the application is exposed, allowing the user to examine the roles and behaviours of the internal parts of the application, and apply arbitrary adaptations both at the interface to the application software, and in the internals of the application, thereby breaking down the black-box design model used to develop the software.

Both case studies also demonstrate some of the capabilities and limitations of the metatype model to perform dynamic adaptations by changing or inserting functional or non-functional behaviour. The object naming case study demonstrates the non-invasive nature of non-functional behavioural changes using the metatype model to perform complex and useful adaptations. The mobile computing case study demonstrates the ability to adapt the functional behaviour of an application, by either adapting the current behaviour of

individual objects or classes, or replacing that a particular class or object in entirety to completely change the resulting behaviour of the application. With respect to the limitations of the metatype model, the mobile computing case study in particular, demonstrates the limitations of the use of the metatypes to adapt at object creation time. However, this is only a limitation with respect to unanticipated dynamic adaptation, this mechanism remains a flexible and feasible approach to perform anticipated dynamic adaptation, or even unanticipated adaptation where objects are created throughout the lifetime of the adapted application.

These two particular case studies were chosen to be representative of a wide range of possible adaptations and so demonstrate the diverse applicability of the framework. The Chisel named object store demonstrates both the extensibility of the Chisel framework and the general-purpose nature of its usability. If any arbitrary application object can be identified, then that object can be arbitrarily inspected or adapted, either by adapting its functional or non-functional behaviours, by introducing before- and after- processing of its creation, deletion, state accesses, method invocations, etc., or indeed by entirely replacing or redirecting these behaviours. The mobile computing example demonstrates the ability to adapt the core functionality of individual objects within an application in a context-aware manner, and so adapt the entire operation of the application to operate in a required manner.

Overall, with the use of these two case studies the usefulness of Chisel dynamic adaptation framework has been demonstrated according to the evaluation criteria discussed in section 5.1. The Chisel framework has been seen to support all of the requirements for completely unanticipated dynamic adaptation, in a context-aware manner, for general-purpose adaptations of arbitrary software. This has also been shown to be possible without requiring access to the source code of the adapted applications, and indeed providing mechanisms to expose the internal behaviours and operations of these applications if the source code is unavailable. This has been achieved primarily by exploiting the useful features of the metatype model to support general-purpose dynamic inspection and adaptation.

However, a serious drawback of the Chisel dynamic adaptation framework is the requirement for the adaptations themselves to be programmed as meta object classes. These adaptations often require in-depth knowledge of the operation of class or object being adapted and require proficiency in programming ability. In this regard, the development of new metatypes may not be suitable for arbitrary users, but rather for users with significant programming experience. The meta-level programmer must also have some level of experience with the rules for using the metatype model, described in Chapter 3, in order to

ensure that the adaptation can be applied without resulting in association errors. It is also necessary that meta object classes are carefully written in a manner to allow the separate association of other metatypes without interference between them. The application of pre-written adaptations is however quite easy to accomplish using the Chisel dynamic adaptation framework. Again some level of knowledge about the operation of the objects and classes being adapted is required, but the difficult implementation of the adaptation is already provided. This ease of use is accomplished primarily by the use of the Chisel policy language, allowing the user to perform complex adaptations without having to rewrite the source code of the application. For this reason the Chisel dynamic adaptation framework is primarily focused towards advanced users who require more functionality from pre-written software, where existing adaptations can be leveraged. Alternatively, in the case the advanced user who is a proficient programmer, the Chisel framework also provides the ability to define and apply new adaptations.

5.6 Chapter summary

This chapter presented two case studies to demonstrate and evaluate the capabilities of the Chisel dynamic adaptation framework. The implementations of the applied adaptations for each case study were presented, along with an analysis of how these adaptations demonstrate the effectiveness and limitations of the Chisel framework. Each adaptation was also discussed in terms of further adaptations that could be applied in a similar manner, to demonstrate the general-purpose nature of the Chisel framework. In addition a set of timing figures was given to demonstrate the performance penalty of using the Chisel dynamic adaptation framework. A general discussion about the practicality of the Chisel framework concluded this chapter.

The next chapter concludes this thesis with a summary of the contributions of the Chisel project, along with a discussion of open research questions and suggestions for further work in this area.

Chapter 6
CONCLUSIONS

This chapter concludes this thesis with a summary of the thesis, an overview of the contributions presented, and a brief description of a number of open research topics not tackled in this thesis.

6.1 Overview of this thesis

Chapter 1 opens with a discussion of the aims and objectives of this thesis. The varying degrees to which individual software adaptation may be anticipated is discussed, after which completely unanticipated dynamic adaptation is introduced. The introduction continues with an overview of metatypes and policy-based adaptation management. The Chisel dynamic adaptation is briefly presented, followed by a discussion of the contributions of this thesis.

Chapter 2 provides an overview and discussion of the most relevant current research that relates to and influenced the design and operation of the Chisel dynamic adaptation framework. These related works were discussed in terms of their support for unanticipated dynamic adaptation and their management supports to control this adaptation. This chapter demonstrated a lack of generalised support for the managed completely unanticipated dynamic adaptation of general-purpose software.

In Chapter 3, the Chisel dynamic adaptation framework is presented. The Chapter opens with an analysis of the requirements that the Chisel framework must fulfil. A detailed discussion about the metatype model is then followed by examination of how dynamic metatype association can be used to perform dynamic software inspection and adaptation. The Chisel event model and the Chisel context model are introduced as a mechanism to define and capture context-aware adaptation requirements for in a reactive manner. The

Chisel policy language is then presented, showing how this policy language can be used to drive the adaptation mechanism in an unanticipated but controlled proactive or reactive context-aware manner.

Chapter 4 presents an implementation of the Chisel dynamic adaptation framework, with the structure and operations of the constituent parts of the framework examined in detail. This chapter show how the Chisel framework is enabled, and explains how step-by-step completely unanticipated dynamic adaptation is performed.

Chapter 5 presents two detailed case studies to show the usefulness and limitations of the Chisel dynamic adaptation framework. The first case study describes the implementation of the Chisel named object store as an adaptation applied using Chisel framework. This non-functional adaptation also demonstrates in a generalised manner how non-functional behavioural adaptations could be applied in an unanticipated and non-invasive manner. The second case study describes the use of the Chisel framework to dynamically adapt third-party network application to operate in a mobile computing environment. This case study demonstrates how functional behaviour adaptations can also be applied in an unanticipated and context-aware manner in response to rapidly changing requirements and resources, all without requiring access to the source code of the compiled application. The benefits and limitations of the Chisel framework are then discussed further.

Chapter 6, this chapter, concludes the thesis with an overview of the contributions of the Chisel project and a discussion of open research questions and further work.

6.2 Contributions of the Chisel Project

This section briefly summarises the contributions of the Chisel project, as presented in the previous chapters.

This thesis provides an in depth study of unanticipated dynamic adaptation and introduces the term "completely unanticipated dynamic adaptation" to refer to the application of software adaptations to a running application, where the new behaviour of each adaptation, the location where an adaptation will be applied, when the adaptation will be are applied, and the control logic that manages the application of the adaptation, can all remain unanticipated until after the target application has started executing, and until the requirements for those adaptations become apparent. This thesis discussed the requirements that an adaptation framework must satisfy to support completely unanticipated dynamic

adaptation, and discusses the current state of the art research with respect to how completely unanticipated dynamic adaptation can be achieved.

This thesis then describes in detail the design and prototype implementation of the Chisel dynamic adaptation framework, which makes completely unanticipated dynamic adaptation achievable for general-purpose compiled applications. This thesis also discusses the ability to exploit changing contextual information to drive unanticipated dynamic software adaptation. Here context has been chosen to refer to all state, resources, and requirements of the user, application, and execution environment. This deliberately wide-ranging interpretation of context is supported by the Chisel adaptation framework by allowing the state and operation of any application software to be monitored, by allowing the user to interact with the adaptation process using the Chisel policy language, and by the use of the Chisel event model to dynamically specify and capture interesting changes in context values. In this way the Chisel framework can support the dynamic application of adaptations where what the adaptation does, where it is applied, how its application is controlled, and when the adaptation is applied, can all remain unanticipated until after the target application has started execution and until the need for that adaptation has arisen.

This thesis also describes how the Chisel framework can be used in an effective and practical manner to inspect and profile the internal operation of compiled software as it executes, thereby allowing the user to probe, extend, and adapt the functional and non-functional behaviours of that software module, without requiring access to its source code. This use of the Chisel framework challenges the black box model of software engineering by allowing the inspection and manipulation of third-party software in an open and general purpose manner. Such inspection and manipulation allows the combination, reuse, and tailoring of diverse software modules in ways unforeseen by their original designers.

This thesis also provides an in-depth discussion about the benefits and limitations of the use of runtime behavioural reflection to implement dynamic metatype association to perform dynamic adaptation. This thesis demonstrates how the dynamic association of metatypes is employed in the Chisel framework as a dynamic adaptation mechanism, to change either the functional or the non-functional behaviours of arbitrary application objects and classes.

6.3 Further work

This section described a number of related research topics and a number of open research topics that have not been fully researched as part of the Chisel project.

6.3.1 The stability and security of adapted software

As stated in Chapter 1, how arbitrarily interfering with or adapting the operation of running software affects the stability of that software has not been discussed in this thesis. Unanticipated dynamic adaptation usually refers to changing some part of the software that was not intended to be changeable. While this change may not adversely affect the operation of the adapted software, great care must be taken to ensure that the unintended effects of the change are minimised.

Since software is generally written in a manner that does not take into account the ability to perform dynamic inspection and adaptation after the software is released, many software modules hide and obscure sensitive data and operations inside the compiled application. However, with the ability to inspect, profile, and adapt, even private operations and behaviours, these sensitive data and operations can be exposed, intercepted, and changed.

The Chisel dynamic adaptation framework is designed and built as an enabling technology, but it can also be used to disable or damage its target. This thesis places responsibility to use the technology carefully onto the user, rather than restrict its operation.

6.3.2 Tool support

Further work is required to support a more user-friendly method to allow the user to dynamically create adaptation code, and the policy rules that control the application of those adaptations. For even proficient programmers, meta-level programming is acknowledged to be a difficult task. While reflective frameworks, such as those discussed in Chapter 2, have made this task easier by presenting more high-level support for reflective programming, further work is required before meta-level programming will be generally accepted. For example, some form of graphical viewer could show which metatypes are associated with base-level classes and objects at any given time, possibly incorporating tool support for metatype association and disassociation. While the Chisel language is designed to be lean and as easy to use as possible, tool support to assist in the creation and validation of policy rules would be beneficial.

6.3.3 Metatype conflicts

Although the problems associated with the combination of possibly conflicting metatypes has been discussed in [64], the metatype model can be difficult to use if multiple metatypes are to be used at the same time. This is particularly true where objects, classes, and

superclasses may all have different and possibly conflicting metatypes associated with them. Ideally, a metatype should be written in a manner independent of any other metatype that it may be composed with, and preferably independent of what type of object or class it may be associated with. However, in practice, where metatypes are written to perform adaptation of functional behaviours, or application-specific adaptation of non-functional behaviour, this separation of concerns will break down to some extent.

Currently in Iguana/J, there is no mechanism to implement advanced metatypes composition strategies since this is performed automatically by the Iguana/J runtime component. Future versions should address this issue.

6.3.4 Iguana

As discussed throughout this thesis, several versions of the Iguana reflective programming model have been implemented for different languages and using different techniques. The aim of the Iguana reflective model is to provide language independent support for runtime reflection. Ongoing work is investigating a possible design and implementation of the Iguana reflective programming model for the Microsoft® language-independent .NET platform [101]. Such work could be used to evaluate how the Chisel dynamic adaptation framework could be extended for use by other languages using different programming methodologies and paradigms, thereby further demonstrating the general-purpose nature of the Chisel framework.

6.3.5 Policy conflicts

This thesis has not discussed the possibility of policy rule conflicts. This is an active research area, but was seen to be outside the scope of this research. With the Chisel framework, it is the responsibility of the user to ensure that policy conflicts do not occur. This is minimised to some extent by enforcing a rule ordering mechanism in an attempt to reduce the possibility of circular rule references, but this mechanism is by no means sufficient to prevent conflicts in a generalised manner.

6.3.6 Other adaptation mechanisms

As discussed in section 4.4 in Chapter 4, all use of the metatype model and Iguana/J is encapsulated in the Chisel behaviour manager. Much of the functionality of the behaviour manager is provided directly by the Iguana/J framework, and so the behaviour manager contains only the necessary operations to leverage the Iguana/J dynamic metatype

association mechanism. This behaviour manager can be easily replaced to exploit a different dynamic adaptation mechanism. Any mechanism that supports the dynamic application of a named adaptation to a named target class or object could be supported. Since this thesis is primarily focused on the use of the metatype model, this aspect of the Chisel framework has not been discussed. Of particular interest would be the use of a structural reflection adaptation framework to add to the introspection capabilities of the framework, since the metatype model does not support the inspection or adaptation of prewritten application code blocks, only the behavioural invocation, interaction, and redirection of these blocks is supported. Such work would assist in determining how the Chisel framework would further generalise across different domains, language, programming models, and usage patterns.

6.3.7 Use in a distributed environment

The current design and implementation of the Chisel framework is specifically for use on one computer and within a single execution context. Further research is required to determine how Chisel would generalise across a distributed environment. This opens a number of research questions. How can the metatype model be used across a distributed environment, and how could it be used to address end to end concerns? Would a more complex distributed event service be beneficial, or sufficient, for use in such an environment? How would the locality of rule effects be addressed, and in particular, how would security and stability of such a mechanism be enforced where remote users could adapt the operation of a local application? How would such system scale across a large number of nodes? Indeed, would the Chisel framework be useful or feasible in such an environment?

6.4 Conclusions

This thesis presents the concept of "completely unanticipated dynamic adaptation". The thesis first defines a completely unanticipated dynamic adaptation as an individual software adaptation that can be dynamically applied even though what the adaptation does, where it is applied, when it is applied, the specification of the controlling logic to manage its application, can all remain unanticipated and unprepared until after the application to be adapted has started executing. This thesis also discusses the requirements that must be met to in order to support completely unanticipated dynamic adaptations, and then presents a design and prototype implementation of a system that meets these requirements: the Chisel dynamic adaptation framework. The Chisel framework is presented in terms of how it

supports the managed applications of general-purpose behavioural adaptations to arbitrary compiled software, in a context-aware but completely unanticipated manner, all without requiring access to the source code of the application.

This thesis also discusses the metatype model, but particularly dynamic metatype association, and evaluates its benefits and limitations as a mechanism to perform dynamic software inspection and adaptation.

This thesis also challenges the black-box model of software engineering, by enabling dynamic inspection and profiling of third-party software without requiring access to its source code. Once the internal roles and operations of the constituent part of the software have been exposed, they can then be dynamically adapted and extended.

References

[1] Aiken, R., M. Carey, B. Carpenter, I. Foster, C. Lynch, J. Mambreti, R. Moore, J. Strasnner, and B. Teitelbaum. (2000) [online]. Network Policy and Services: A Report of a Workshop on Middleware. (RFC 2768) (http://www.ietf.org/rfc/rfc2768.txt). 9 September 2004 [date accessed]

[2] Andersen, A. "OOPP, A Reflective Middleware Platform including Quality of Service Management", *Dr. Scient. Thesis*, in Department of Computer Science, University of Tromsø: Tromsø, Norway. 2002.

[3] AspectWerkz Project. (2004) [online]. AspectWerkz Homepage: http://aspectwerkz.codehaus.org/. 14 April 2005 [date accessed]

[4] BBN Technologies. (2002) [online]. Quality Objects (QuO) website (http://quo.bbn.com). 27 August 2004 [date accessed]

[5] Berbers, Y., "Handling adaptive behavior in real-time systems", in *Technologies for the Information Society: Developments and Opportunities*, J. Roger, B. Stanford-Smith, and P.T. Kidd, Editors. IOS Press: Amsterdam. 1998.

[6] Berry, G., "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: Cambridge, MA, USA. 2000.

[7] Biegel, G., V. Cahill, and M. Haahr. "A Dynamic Proxy-Based Architecture to Support Distributed Java Objects in Mobile Environments". in *Proceedings of the International Symposium of Distributed Objects and Applications, (DOA 2002), (LNCS 2519)*. 2002. Irvine, CA. Springer Verlag.

[8] Bisbal, J., D. Lawless, B. Wu, and J. Grimson, "Legacy Information Systems: Issues and Directions", in *IEEE Software*, 1999. **16**(5).

[9] Blair, G.S., A. Andersen, L. Blair, G. Coulson, and D. Sánchez, "Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform", in *IEE Proceedings - Software*, 2000. **147**(01).

[10] Blair, G.S., L. Blair, V. Issarny, P. Tuma, and A. Zarras. "The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms". in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000) (LNCS 1795)*. 2000. New York, NY, USA. Springer-Verlag.

[11] Blair, G.S., F.M. Costa, G. Coulson, H.A. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, and J.-B. Stefani. "The Design of a Resource-Aware Reflective Middleware Architecture". in *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection (Refelection 1999), (LNCS 1616)*. 1999. St. Malo, France. Springer-Verlag.

[12] Blair, G.S., G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, and K. Saikoski. "A principled approach to supporting adaptation in distributed mobile environments". in *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*. 2000. Limerick, Ireland

[13] Blair, G.S., G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski, "The Design and Implementation of Open ORB v2", in *IEEE Distributed Systems Online*, 2001. **2**(6).

[14] Blair, G.S., G. Coulson, P. Robin, and M. Papathomas. "An Architecture for Next Generation Middleware". in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. 1998. Lake District, UK. Springer-Verlag.

[15] Bonér, J., "AspectWerkz - dynamic AOP for Java. (http://www.codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf)". 2003.

[16] Booch, G., *Object-Oriented Design With Applications*. 2 ed. Benjamin/Cummings: Redwood City, California. 1993.

[17] Bouraqadi-Saâdani, N., T. Ledoux, and M. Südholt, "A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation (Technical Report 01-7-INFO)". École des Mines de Nantes: Nantes, France. 2001.

[18] Buckley, J., T. Mens, M. Zenger, A. Rashid, and G. Kniesel., "Towards a Taxonomy of Software Change", in *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE). To Appear*, 2005. **17**(5).

[19] Cahill, V., "The Iguana Reflective Programming Model (Report: C1-98)". DSG, Department of Computer Science, Trinity College Dublin. 1998.

[20] Capra, L. "Reflective Mobile Middleware for Context-Aware Applications", *PhD Thesis*, in Department of Computer Science, University College London, University of London. 2003.

[21] Capra, L., G. Blair, C. Mascolo, W. Emmerich, and P. Grace, "Exploiting Reflection in Mobile Computing Middleware", in *ACM SIGMOBILE Mobile Computing and Communications Review.*, 2002. **6**(4).

[22] Capra, L., W. Emmerich, and C. Mascolo, "A Micro-Economic Approach to Conflict Resolution in Mobile Computing (UCL Research Note RN/38/01)". University College London: London. 2001.

[23] Capra, L., W. Emmerich, and C. Mascolo. "Reflective Middleware Solutions for Context-Aware Applications". in *Proceedings of The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION 2001) (LNCS 2192)*. 2001. Kyoto, Japan. Springer-Verlag.

[24] Chiba, S. "Load-time Structural Reflection in Java". in *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000) (LNCS 1850)*. 2000. Sophia Antipolis and Cannes, France. Springer-Verlag.

[25] Chiba, S. (2003) [online]. OpenC++ Home Page (http://www.csg.is.titech.ac.jp/~chiba/openc++.html). 14 April 2004 [date accessed]

[26] Chiba, S. and M. Nishizawa. "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators". in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03) (LNCS 2830)*. 2003. Erfurt, Germany. Springer-Verlag.

[27] Chikofsky, E.J. and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy", in *IEEE Software*, 1990. **7**(1).

[28] Coulson, G., G.S. Blair, M. Clarke, and N. Parlavantzas, "The design of a configurable and reconfigurable middleware platform", in *ACM Distributed Computing Journal*, 2002. **15**(2).

[29] Cunningham, R. "Architecture for Location Independent CORBA Environments", *MSc dissertation*, in Department of Computer Science, University of Dublin, Trinity College.: Dublin. 1998.

[30] Czarnecki, K. and U.W. Eisenecker, "Aspect Oriented Programming", in *Generative programming: methods, tools, and applications*. Addison-Wesley. 2000.

[31] Czarnecki, K. and U.W. Eisenecker, *Generative programming: methods, tools, and applications*. Addison-Wesley. 2000.

[32] Dahm, M., "Byte Code Engineering with the BCEL API (Technical Report B-17-98)". Institut fur Informatik, Freie Universitat Berlin: Berlin. 2001.

[33] Damianou, N. " A Policy Framework for Management of Distributed Systems", *PhD Thesis*, in Department of Computing, Imperial College, University of London: London. 2002.

[34] Damianou, N., N. Dulay, E. Lupu, and M. Sloman, "The Ponder Language Specification (Report DoC 2000/1)". Imperial College: London. 2000.

[35] Damianou, N., N. Dulay, E. Lupu, and M. Sloman. "The Ponder Specification Language". in *Workshop on Policies for Distributed Systems and Networks (Policy 2001)*. 2001. HP Labs, Bristol

[36] David, P.-C. "Une infrastructure pour middleware adaptable", *Rapport de DEA (Equivalent to MSc)*, in Institut de Recherche en Informatique de Nantes, Faculté des Sciences & Techniques de Nantes, École des Mines de Nantes & Université de Nantes. 2001.

[37] David, P.-C. and T. Ledoux. "Dynamic Adaptation of Non-Functional Concerns". in *Proceedings of First International Workshop on Unanticipated Software Evolution, at ECOOP 2002*. 2002. Malaga, Spain

[38] David, P.-C. and T. Ledoux. "An Infrastructure for Adaptable Middleware". in *Proceeding of the 2002 International Symposium on Distributed Objects and Applications (DOA 2002) (LNCS 2519)*. 2002. Irvine, California, USA. Springer-Verlag.

[39] David, P.-C. and T. Ledoux. "Towards a Framework for Self-Adaptive Component-Based Applications". in *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, (DAIS 2003) (LNCS 2893)*. 2003. Paris, France. Springer-Verlag.

[40] David, P.-C., T. Ledoux, and N.M. Bouraqadi-Saâdani. "Two-step Weaving with Reflection using AspectJ". in *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, at OOPSLA 2001*. 2001. Tampa Bay, USA

[41] Dias, M. and D. Richardson, "Issues on Software Monitoring". Department of Information and Computer Science, University of California, Irvine, CA. 2002.

[42] Dmitriev, M. "Safe Class and Data Evolution in Large and Long-Lived Java Applications", *PhD Thesis*, in Department of Computing Science, University of Glasgow: Glasgow, Scotland. 2001.

[43] Dmitriev, M. "Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications". in *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE 2001), in association with OOPSLA 2001 International Conference*. 2001. Tampa Bay, Florida, USA,

[44] Dowling, J. and V. Cahill. "Dynamic Software Evolution and the K-Component Model". in *Workshop on Software Evolution, at OOPSLA 2001*. 2001. Tampa Bay, Florida, USA

[45] Dowling, J. and V. Cahill. "The K-Component Architecture Meta-Model for Self-Adaptive Software". in *Proceedings of The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001), (LNCS 2192)*. 2001. Kyoto, Japan

[46] Dowling, J., T. Schaefer, V. Cahill, P. Haraszti, and B. Redmond. "Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation". in *Workshop on Object-Oriented Reflection and Software Engineering at OOPSLA '99*. 1999. Denver, Colorado, USA

[47] Dulay, N., E. Lupu, M. Sloman, and N. Damianou. "A Policy Deployment Model for the Ponder Language". in *IEEE/IFIP International Symposium on Integrated Network Management (IM'2001)*. 2001. Seattle

[48] Duran-Limon, H.A. "A Resource Management Framework for Reflective Multimedia Middleware", *Ph.D. Thesis*, in Computing Department, University of Lancaster. 2001.

[49] Efstratiou, C., A. Friday, N. Davies, and K. Cheverst. "Utilising the Event Calculus for Policy Driven Adaptation on Mobile Systems". in *Proceedings of the 3rd IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2002)*. 2002. Monterey, CA, USA

[50] Elrad, T., R.E. Filman, and A. Bader, "Aspect-oriented programming: Introduction", in *Communications of the ACM*, 2001. **44**(10).

[51] Fekete, J.-D. and M. Richard, "Esterel Meets Java: Building Reactive Synchronous Programs in Java, (http://www.lri.fr/~fekete/ps/EmeetsJ.pdf)". École des Mines de Nantes: Nantes, France. 1998.

[52] Ferber, J. "Computational reflection in class based object-oriented languages". in *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 1989)*. 1989. New Orleans, Louisiana, United States. ACM Press.

[53] Filman, R. and D. Friedman. "Aspect-Oriented Programming is Quantification and Obliviousness". in *Workshop on Advanced Separation of Concerns, OOPSLA 2000*. 2000. Minneapolis

[54] Forman, G.H. and J. Zahorjan, "The Challenges of Mobile Computing". University of Washington. 1994.

[55] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. 1994.

[56] Golm, M. "Design and Implementation of a Meta Architecture for Java (revised version)", *Masters Thesis*, in Department of Computer Sciences, University of Erlangen. 1997.

[57] Golm, M. and J. Kleinöder. "MetaXa and the Future of Reflection". in *Workshop on Reflective Programming in C++ and Java (OOPSLA'98)*. 1998. Vancouver, British Columbia, Canada

[58] Gowing, B. "A Reflective Programming Model and Language for Dynamically Modifying Compiled Software", *Ph.D. Thesis*, in Department of Computer Science, Trinity College Dublin: Dublin. 1997.

[59] Gowing, B. and V. Cahill. "Meta-Object Protocols for C++: The Iguana Approach". in *Proceedings of Reflection '96*. 1996. San Francisco

[60] Grace, P., G.S. Blair, and S. Samuel. "ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability". in *Proceedings of International Symposium on Distributed Objects and Applications (DOA 2003), (LNCS 2888)*. 2003. Catania, Sicily, Italy. Springer-Verlag.

[61] Haahr, M. "Supporting Mobile Computing in Object-Oriented Middleware Architectures", *Ph.D. Thesis*, in Department of Computer Science, Trinity College Dublin: Dublin. 2003.

[62] Haahr, M., R. Cunningham, and V. Cahill. "Supporting CORBA Applications in a Mobile Environment." in *Proceedings of the 5th International Conference on Mobile Computing and Networking (MobiCom '99)*. 1999. Seattle

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

[64] Haraszti, P. "Towards Automatic and Dynamic Meta-Object Protocol Composition in a Compiled, Reflective Programming Language", *Ph.D. Thesis*, in Department of Computer Science, Trinity College Dublin: Dublin. 2003.

[65] Hilsdale, E. and J. Hugunin. "Advice weaving in AspectJ". in *Proceedings of the 3nd International Conference on Aspect-Oriented Software Development (AOSD 2004)*. 2004. Lancaster, UK. ACM Press.

[66] IBM Research. (2004) [online]. Jikes RVM (Research Virtual Machine) (http://www.research.ibm.com/jikes/). 14 April 2005 [date accessed]

[67] Indulska, J., S.W. Loke, A. Rakotonirainy, and A. Zaslavsky. "Adaptive Enterprise Architecture for Mobile Computation". in *Workshop on Reflective Middleware, at Middleware 2000*. 2000

[68] Indulska, J., S.W. Loke, A. Rakotonirainy, and A. Zaslavsky. "An Open Architecture for Pervasive Computing". in *Proceedings of IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS 01)*. 2001. Krakow. Kluwer.

[69] Iona Technologies. (2004) [online]. Orbacus website (www.orbacus.com). 27 August 2004 [date accessed]

[70] Itoh, J., R. Lea, and Y. Yokote. "Using meta-objects to support optimization in the Apertos operating system". in *Proceedings of USENIX Conference on Object Oriented Technologies (COOTS 1995)*. 1995

[71] Jarir, Z., P.-C. David, and T. Ledoux. "Dynamic Adaptability of Services in Enterprise JavaBeans Architecture". in *The 7th International Workshop on Component-Oriented Programming (WCOP 2002) at ECOOP 2002,*. 2002. Malaga, Spain

[72] Jing, J., A.S. Helal, and A. Elmagarmid, "Client-server computing in mobile environments", in *ACM Computing Surveys*, 1999. **31**(2).

[73] Joosen, W., F. Matthijs, J.V. Oeyen, B. Robben, S. Bijnens, and P. Verbaeten, "CORRELATE: High-level Support for Travelling Agents" (Technical Report CW236). Dept. of Computer Science, Katholieke Universiteit Leuven: Leuven. 1996.

[74] Jørgensen, B.N., E. Truyen, F. Matthijs, and W. Joosen. "Customization of Object Request Brokers by Application Specific Policies". in *Proceedings of Middleware 2000 (LNCS 1795)*. 2000. New York, USA. Springer Verlag.

[75] Joseph, A.D., J.A. Tauber, and M.F. Kaashoek, "Mobile Computing with the Rover Toolkit", in *IEEE Transactions on Computers*, 1997. **46**(3).

[76] Joy, B., G. Steele, J. Gosling, and G. Bracha, *Java Language Specification*. 2 ed. Addison-Wesley. 2000.

[77] Jugel, M.L. and M. Meißner. (2003) [online]. The Java Telnet Application/Applet v.2.5 (http://javatelnet.org). 14 April 2005 [date accessed]

[78] Kagal, L., "Rei: A Policy Language for the Me-Centric Project" (Technical report: HPL-2002-270). HP Labs. 2002.

[79] Kagal, L., T. Finin, and A. Joshi. "A Policy Language for a Pervasive Computing Environment". in *Proceedings of The Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*. 2003. Lake Como, Italy. IEEE Computer Society.

[80] Kasten, E.P. and P.K. McKinley, "Adaptive Java: Refractive and Transmutative Support for Adaptive Software." (Technical Report MSU-CSE-01-30). Department of Computer Science and Engineering, Michigan State University: East Lansing, Michigan, USA. 2001.

[81] Keller, R. and U. Holzle, "Binary Component Adaptation" (TRCS97-20). University of California at Santa Barbara: Santa Barbara, CA, USA. 1997.

[82] Kiczales, G., "Beyond the Black Box: Open Implementation", in *IEEE Software*, 1996. **13**(1).

[83] Kiczales, G., J. des Rivieres, and D. Bobrow, *The Art of the Metaobject Protocol*. MIT Press. 1991.

[84] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming". in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97) (LNCS 1241)*. 1997. Jyväskylä, Finland. Springer-Verlag.

[85] Kniesel, G., P. Costanza, and M. Austermann. "JMangler - A Framework for Load-Time Transformation of Java Class Files". in *IEEE Workshop on Source Code Analysis and Manipulation (SCAM 2001)*. 2001. Florence, Italy. IEEE Computer Society Press.

[86] Kniesel, G., P. Costanza, and M. Austermann, "JMangler - A Powerful Back-End for Aspect-Oriented Programming." in *Aspect-oriented Software Development (To appear)*, R. Filman, et al., Editors. Prentice Hall. 2004.

[87] Kniesel, G., J. Noppen, T. Mens, and J. Buckley. "Unanticipated Software Evolution". in *ECOOP 2002 Workshop Reader (LNCS 2548)*. 2002. Malaga, Spain. Springer-Verlag.

[88] Kojo, M., K. Raatikainen, and T. Alanko, "Connecting Mobile Workstations to the Internet over a Digital Cellular Telephone Network", in *MOBILE COMPUTING*, T. Imielinski and H.F. Korth, Editors. Kluwer Academic Publishers. 1996.

[89] Kojo, M., K. Raatikainen, M. Liljeberg, J. Kiiskinen, and T. Alanko, "An Efficient Transport Service for Slow Wireless Telephone Links", in *IEEE Journal on Selected Areas in Communications*, 1997. **15**(7).

[90] Kon, F. "Automatic Configuration of Component-Based Distributed Systems", *PhD Thesis*, in Department of Computer Science, University of Illinois at Urbana-Champaign. 2000.

[91] Kon, F., R. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. "2K: A Distributed Operating System for Dynamic Heterogeneous Environments". in *Proceedings of 9th IEEE International Symposium on High Performance Distributed Computing*. 2001. Pittsburgh

[92] Kon, F., B. Gill, R.H. Campbell, and M.D. Mickunas. "Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents". in *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'2000)*. 2000. Zurich

[93] Kon, F., J.R. Marques, T. Yamane, R.H. Campbell, and M.D. Mickunas, "Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems", in *Software: Practice and Experience. To Appear*, 2005. **35**(7).

[94] Lafferty, D. and V. Cahill. "Language-independent aspect-oriented programming". in *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*. 2003. Anaheim, California, USA. ACM Press.

[95] Maes, P. "Computational Reflection (Tecnical Report VUB AI-Lab TR-87-02)", *PhD Thesis*, in Artificial Intelligence Laboratory, Vrije Universiteit: Brussels, Belgium. 1987.

[96] Maltz, D.A. and Pravin Bhagwat. "MSOCKS: An Architecture for Transport Layer Mobility". in *Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM '98)*. 1998

[97] Mansouri-Samani, M. "Monitoring of Distributed Systems", *PhD Thesis*, in Department of Computing, Imperial College, University of London: London. 1995.

[98] Marriott, D. "Policy Service for Distributed Systems", *PhD Thesis*, in Department of Computing, Imperial College, University of London: London. 1997.

[99] Mätzel, K. and P. Schnorf, "Dynamic Component Adaptation." (Ubilab Technical Report 97.6.1). Union Bank of Switzerland: Zurich, Switzerland. 1997.

[100] Microsoft Corporation. (1999) [online]. COM+ (http://www.microsoft.com/com). 14 April 2005 [date accessed]

[101] Microsoft Corporation. (2005) [online]..NET Framework SDK (http://msdn.microsoft.com/netframework/). 14 April 2005 [date accessed]

[102] Milojicic, D., F. Douglis, and R. Wheeler, eds. *Mobility: Processes, Computers, and Agents*. 1999. ACM Press Series

[103] Mockapetris, P. (1987) [online]. Domain Names: Concepts and Facilities (STD 13 / RFC 1034) (http://www.ietf.org/rfc/std/std13.txt). 9 September 2004 [date accessed]

[104] Moffett, J. and M. Sloman, "Policy Hierarchies for Distributed Systems Management", in *IEEE Journal on Selected Areas in Communications*, 1993. **11**(9).

[105] Morisio, M., C.B. Seaman, V.R. Basili, A.T. Parra, S.E. Kraft, and S.E. Condon, "COTS-Based Software Development: Processes and Open Issues", in *Journal of Systems and Software*, 2002. **61**(3).

[106] Object Management Group, "Common Object Request Broker Architecture: Core Specification (OMG Document formal/02-12-06)". 2002.

[107] Okoshi, T., M. Mochizuki, Y. Tobe, and H. Tokuda. "MobileSocket: Toward Continuous Operation for Java Applications". in *Proceedings of the 8th International Conference on Computer Communications and Networks*. 1999. Boston, MA, USA. IEEE Communication Society.

[108] Okoshi, T., M. Mochizuki, Y. Tobe, and H. Tokuda, "MobileSocket: Session Layer Continuous Operation Support for Java Applications", in *Transactions of the Information Processing Society of Japan (IPSJ)*, 2000. **41**(2).

[109] Oliva, A. "Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java", *Masters Thesis*, in Instituto de Computação, Universidade Estadual de Campinas, Brazil. 1998.

[110] Oliva, A. and L.E. Buzato, "Guaraná: A tutorial." (Technical Report IC-98-31). Instituto de Computação, Universidade Estadual de Campinas, Brazil. 1998.

[111] Oliva, A. and L.E. Buzato. "The Design and Implementation of Guaraná". in *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*. 1999. San Diego, California, USA

[112] Oreizy, P. "Open Architecture Software: A Flexible Approach to Decentralized Software Evolution", *Ph.D. Thesis*, in Information and Computer Science, University of California, Irvine: Irvine. 2000.

[113] Oreizy, P., M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software", in *IEEE Intelligent Systems*, 1999. **14**(3).

[114] Ossher, H. and P. Tarr. "Hyper/J: multi-dimensional separation of concerns for Java". in *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. 2000. Limerick, Ireland

[115] Parlavantzas, N., G. Coulson, M. Clarke, and G. Blair. "Towards a Reflective Component Based Middleware Architecture". in *Workshop on Reflection and Metalevel Architectures*. 2000. Sophia Antipolis and Cannes, France

[116] Podgurski, A. and L. Pierce, "Retrieving reusable software by sampling behavior", in *ACM Transactions on Software Engineering and Methodology*, 1993. **2**(3).

[117] Popovici, A., G. Alonso, and T. Gross. "Just-in-time aspects: efficient dynamic weaving for Java". in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*. 2003. Boston, Massachusetts. ACM Press.

[118] Popovici, A., T. Gross, and G. Alonso. "Dynamic weaving for aspect oriented programming". in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*. 2002. Enschede, The Netherlands. ACM Press.

[119] Prakash:, R., "Education: Mobile Computing." in *IEEE Distributed Systems Online*, 2001. **2**(6).

[120] Rakotonirainy, A., J. Indulska, S.W. Loke, and A. Zaslavsky. "Middleware for Reactive Components: An Integrated Use of Context, Roles and Event Based Coordination". in *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001, (LNCS Vol. 2218)*. 2001. Heidelberg, Germany. Springer Verlag.

[121] Raverdy, P.-G., R. Le Van Gong, and R. Lea. "DART: A Reflective Middleware for Adaptive Applications". in *Proceedings of the Workshop on Reflective Programming in C++ and Java, at OOPSLA 1998*. 1998. Vancouver, Canada

[122] Raverdy, P.-G. and R. Lea. "DART: A distributed adaptive run-time". in *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98) Work in Progress Session.* 1998. The Lake District, England

[123] Raverdy, P.-G. and R. Lea. "Reflection support for adaptive distributed applications". in *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*. 1999. Mannheim, Germany. IEEE.

[124] Redmond, B. (2003) [online]. Iguana/J Home Page (http://www.iguanaj.org/). 14 April 2005 [date accessed]

[125] Redmond, B. "Supporting Unanticipated Dynamic Adaptation of Object-Oriented Software", *Ph.D. Thesis*, in Department of Computer Science, Trinity College Dublin: Dublin. 2003.

[126] Redmond, B. 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 2000)*. 2000. Cannes, France

[127] Redmond, B. and V. Cahill. "Supporting Unanticipated Dynamic Adaptation of Application Behaviour". in *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002) (LNCS 2374)*. 2002. Malaga, Spain. Springer-Verlag.

[128] Robben, B. "Language Technology and Metalevel Architectures for Distributed Objects." *PhD Thesis*, in Department of Computer Science, Katholieke Universiteit Leuven: Leuven. 1999.

[129] Robben, B., W. Joosen, F. Matthijs, B. Vanhaute, and PierreVerbaeten. "A Metaobject Protocol for Correlate". in *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), at ECOOP '98 (LNCS 1543)*. 1998. Brussels, Belgium. Springer Verlag.

[130] Robben, B., W. Joosen, F. Matthijs, B. Vanhaute, and P. Verbaeten, "Building a Meta-level architecture for distributed applications (Technical Report CW 265)". Department of Computer Science, Katholieke Universiteit Leuven: Leuven. 1998.

[131] Robben, B., B. Vanhaute, W. Joosen, and P. Verbaeten. "Non-Functional Policies". in *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection*. 1999. Saint-Malo, France. Springer-Verlag.

[132] Román, M., F. Kon, and R. Campbell. "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case". in *Proceedings of the Workshop on Middleware at 19th International Conference on Distributed Computing Systems (ICDCS'99)*. 1999. Austin, Texas. IEEE Computer Society.

[133] Sadjadi, S.M. and P.K. McKinley. "ACT: An adaptive CORBA template to support unanticipated adaptation." in *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*. 2004. Tokyo, Japan

[134] Sadjadi, S.M. and P.K. McKinley. "Transparent self-optimization in existing CORBA applications". in *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*. 2004. New York, NY, USA

[135] Sadjadi, S.M., P.K. McKinley, R.E.K. Stirewalt, and B.H.C. Cheng, "TRAP: Transparent reflective aspect programming." (Technical Report MSU-CSE-03-31). Computer Science and Engineering, Michigan State University: East Lansing, Michigan, USA. 2003.

[136] Sadjadi, S.M., P.K. McKinley, R.E.K. Stirewalt, and B.H.C. Cheng. "Generation of Self-Optimizing Wireless Network Applications". in *Proceedings of the International Conference on Autonomic Computing (ICAC-04)*. 2004. New York, NY, USA. IEEE Computer Society.

[137] Sandia National Laboratories. (2003) [online]. Jess, the Rule Engine for the Java Platform (http://herzberg.ca.sandia.gov/jess/). 14 April 2005 [date accessed]

[138] Sato, Y., S. Chiba, and M. Tatsubori. "A Selective, Just-in-Time Aspect Weaver". in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, (GPCE 2003), (LNCS 2830)*. 2003. Erfurt, Germany. Springer-Verlag.

[139] Schäfer, T. "Supporting Metatypes in a compiled, reflective programming language", *PhD thesis*, in Department of Computing Science, Trinity College Dublin: Dublin. 2001.

[140] Schmidt, D.C. (2002) [online]. Real-time CORBA with TAO (The ACE ORB) (http://www.cs.wustl.edu/~schmidt/TAO.html). 14 April 2005 [date accessed]

[141] Senra, R. (2001) [online]. Guaraná Development Kit Home Page (http://www.ic.unicamp.br/~921234/gdk.html). 20 August 2004 [date accessed]

[142] Silva, F.J.S., M. Endler, and F. Kon. "Developing Adaptive Distributed Applications: a Framework Overview and Experimental Results". in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2003) (LNCS 2888)*. 2003. Catania, Sicily, Italy. Springer Verlag.

[143] Sloman, M., "Policy Driven Management For Distributed Systems", in *Journal of Network and Systems Management*, 1994. **2**(4).

[144] Smith, B.C. "Reflection and Semantics in a Procedural Language (Technical Report MIT-LCS-TR-272)", *Ph.D. Thesis*, in Department of Electrical Engineering and Computer Science, MIT: Cambridge, Massachusetts. 1982.

[145] Sun Microsystems. (2000) [online]. Java Native Interface (JNI) (http://java.sun.com/j2se/1.3/docs/guide/jni/index.html). 14 April 2005 [date accessed]

[146] Sun Microsystems. (2001) [online]. Java Platform Debugger Architecture Enhancements (http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html#hotswap). 28 August 2004 [date accessed]

[147] Sun Microsystems. (2001) [online]. Java Platform Debugger Architecture: Java Virtual Machine Debug Interface Reference (http://java.sun.com/products/jpda/doc/jvmdi-spec.html). 28 August 2004 [date accessed]

[148] Sun Microsystems. (2002) [online]. Java 2 Platform, Standard Edition (J2SE) (http://java.sun.com/j2se/). 14 April 2005 [date accessed]

[149] Sun Microsystems. (2002) [online]. Java Naming and Directory Interface (http://java.sun.com/products/jndi/). 28 August 2004 [date accessed]

[150] Sun Microsystems. (2002) [online]. Java Remote Method Invocation Specification (http://java.sun.com/products/jdk/rmi/). 14 April 2005 [date accessed]

[151] Sun Microsystems. (2003) [online]. HotSwap Client Tool: (http://developers.sun.com/dev/coolstuff/hotswap). 28 August 2004 [date accessed]

[152] Tatsubori, M., S. Chiba, M.-O. Killijian, and K. Itano. "OpenJava: A Class-Based Macro System for Java". in *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering (OORaSE 1999) (LNCS 1826)*. 2000. Denver, CO, USA. Springer-Verlag.

[153] Truyen, E., B. Vanhaute, and W. Joosen. "Integrating flexible middleware solutions with applications through non-functional policies". in *Proceedings of OOPSLA Workshop on Reflection and Software Engineering (OORaSE '99)*. 1999. Denver, USA

[154] Vanhaute, B., E. Truyen, W. Joosen, and P. Verbaeten. "Composing non-orthogonal meta-programs". in *Proceedings of the 1st Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems (at OOPSLA '99)*. 1999

[155] Vasseur, A. "Java Dynamic AOP and Runtime Weaving - How does AspectWerkz address it?" in *Dynamic Aspects Workshop (DAW04)*. 2004. Lancaster, UK.

[156] Wall, T. "Mobility and Java RMI", *M.Sc. Thesis*, in Department of Computing Science, Trinity College Dublin: Dublin. 2000.

[157] Wall, T. and V. Cahill. "Mobile RMI: Supporting Remote Access to Java Server Objects on Mobile Hosts". in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*. 2001. Rome, Italy. IEEE Computer Society.

[158] Welch, I. and R.J. Stroud, "Dalang - A Reflective Extension for Java. (CS-TR: 672)". Department of Computing Science, University of Newcastle. 2000.

[159] Welch, I.S. and R.J. Stroud. "Kava - A Reflective Java Based on Bytecode Rewriting,". in *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering (OORaSE 1999), (LNCS 1826)*. 1999. Denver, CO, USA. Springer-Verlag.

[160] Welch, I.S. and R.J. Stroud. "Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code". in *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000), (LNCS 1895)*. 2000. Toulouse, France. Springer-Verlag.

[161] Welch, I.S. and R.J. Stroud. "Kava - Using Bytecode Rewriting to add Behavioural Reflection to Java,". in *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*. 2001. San Antonio, Texas, USA. USENIX Association.

[162] Xerox PARC. (2004) [online]. The AspectJ Project (http://aspectj.org). 17 August 2004 [date accessed]

[163] Yeong, W., T. Howes, and S. Kille. (1995) [online]. Lightweight Directory Access Protocol (RFC 1777) (http://www.ietf.org/rfc/rfc1777.txt). 9 September 2004 [date accessed]

[164] Yokote, Y. "The Apertos reflective operating system: The concept and its implementation". in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1992)*. 1992. ACM Press.

[165] Yokote, Y. "Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach". in *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS)*. 1993

"Well, let's away, and say how much is done."

William Shakespeare (1564 - 1616), *Macbeth (III, iii), ~1605*